A DOMAIN-SPECIFIC DESIGN TOOL FOR VERIFYING SPACECRAFT SYSTEM BEHAVIOR

by

Sravanthi Venigalla

A thesis submitted in partial fulfillment
of the requirements for the degree

of

MASTER OF SCIENCE

in

Computer Engineering

Approved:

Dr. Brandon Eames Major Professor	Dr. Edmund Spencer Committee Member
Dr. Jacob Gunther Committee Member	Dr. Byron R. Burnham Dean of Graduate Studies

UTAH STATE UNIVERSITY
Logan, Utah

2009

Abstract

A Domain-Specific Design Tool for Verifying Spacecraft System Behavior

by

Sravanthi Venigalla, Master of Science

Utah State University, 2009

Major Professor: Dr. Brandon Eames
Department: Electrical and Computer Engineering

In this report we present a graphical tool, Behavioral Analysis of Spacecraft Systems (BASS), that can be used by spacecraft designers to perform system-level behavioral analysis of small satellites. The domain-specific spacecraft meta-model is created in the visual modeling tool Generic Modeling Environment (GME) such that spacecraft designs created using the meta-model appear familiar to the spacecraft designers. Users can model scenarios that are to be verified for the design in BASS. The graphical models are assigned formal semantics facilitating the creation of formally verifiable spacecraft models. The C++ application that translates the modeling objects to equivalent mathematical representation of interest is called BASS Interpreter and is bound to the meta-model. BASS Interpreter that generates Communicating Sequential Processes (CSP) semantics for the visual spacecraft models is supported in the current work. The model-checker for CSP called Failures Divergences and Refinement (FDR) is run to explore the state-space of the spacecraft process model to comment on the design. We demonstrate the feasibilty and advantage of incorporating BASS into initial design phases of small satellite development by successfully verifying the design of Tomographic Remote Observer of Ionospheric Disturbances (TOROID).

(165 pages)

To my loving family and Sharath

Acknowledgments

Pursuing research and compiling this thesis has been a memorable experience I will cherish throughout my life. Accomplishing this would not have been possible without the constant guidance and encouragement of Dr. Brandon Eames. Dr. Eames has been my mentor since I began the masters program here at Utah State University, and he has been a tremendous influence throughout the two years of my academic career. The learning experiences from all of his graduate courses, and the research collaboration have been extremely enriching and enjoyable, and I would like to thank him profusely for all his time and support.

I would like to sincerely thank my committee members, Dr. Jacob Gunther and Dr. Edmund Spencer, who have encouraged my work throughout this project. Special thanks to Dr. Charles Swenson for taking time from his busy schedule to review and provide important feedback on our work. His course, Spacecraft Systems Engineering, was highly instrumental in developing a good understanding of spacecraft systems.

I am extremely grateful to Space Dynamics Lab for funding this project.

I would like to acknowledge all my friends here at Logan, especially Chandana, Miti, Masuyo, Varsha, and Padmashri, with whom I have spent some great times, helping me take my mind off school. Many thanks to my colleagues at the ECE Department: Rohit Saraswat and Atul Kumar Shah, for helping me on multiple occasions with setting up softwares and giving some good advice.

Sharath and my brother Sandeep managed to keep up my spirits during stressful times and deserve a special note of thanks. Last, but by no means least, I would like to thank my mother, Sudha Rani, who is my constant source of inspiration, and my father, whose blessings are with me always.

Sravanthi Venigalla

Contents

List of Figures

Chapter 1

Introduction

1.1 Background

1.1.1 Spacecraft Systems

Spacecraft play an extensive, but often overlooked, role in our day to day lives. Satellites represent a crucial part in communications, remote-sensing, reconnaissance, entertainment, scientific research, etc. Their ability to see the earth as a whole, while in a microgravity environment, offers a unique advantage over terrestrial instruments. However, this advantage is offset by the harshness of space: the space environment is not completely understood and access is extremely limited. Consequently, satellite developers invest significant design-time effort in minimizing operational errors, especially considering the costliness of post-deployment repair. Further, only a finite amount of remote control on the spacecraft can be afforded due to limited power budgets to support ground communication. All these factors render spacecraft engineering as hard, long, and expensive when compared to other terrestrial systems engineering.

The basic design process adopted for building spacecraft has not changed significantly over the years. This is due to the inclination to reuse architectures and technology that has successfully flown in earlier missions, and the fact that for similar mission scenarios, typically only the payload of the spacecraft changes to any significant degree. The several aspects of spacecraft design include attitude control, propulsion and launch, orbital mechanics, communications, hardware and software, power systems, etc. Experts from each of the various areas identified contribute to spacecraft design, identifying and finalizing the interdependencies between dissimilar areas as the design proceeds.

Development within each "subsystem" area can be quite involved, with engineers drawing heavily from prior flight designs, discipline-specific engineering and mathematics, and subsequent simulations in tools such as MATLAB to confirm system operation. Once developed, these individual modules are run through several test cases to verify whether the requirements are satisfied, and are then integrated and re-tested. Errors uncovered during system integration can be costly to repair, and can impact the project time line to a significant degree. This is due to the fact that fixing such errors can necessitate revisiting the designs of multiple subsystems in order to properly accommodate the necessary subsystem interactions. While carefully planning subsystem interfaces early in the design process can reduce these costly iterations to a large degree, subsystems can often interact in non-obvious ways, which under some circumstances can lead to unexpected behavior. These scenarios arise due to the large number of internal states in each subsystem. When subsystems interact, integration-level testing cannot possibly cover the cross product of the state spaces of multiple complex components. Formal methods can be employed to explore the state spaces comprehensively in order to establish that a system meets its requirements and is deadlock free.

1.1.2 Spacecraft Composition

A system as complicated as a spacecraft is typically composed of multiple components, each trying to accomplish a specific task. These components are referred to as subsystems. A spacecraft subsystem is a group of hardware and/or software which accomplishes a set of functional requirements. These subsystems are generally shown as blocks in a system-level diagram, which communicate either via the system buses or through custom interfaces, shown as connections. The system buses carry power and data between subsystems. An example spacecraft block diagram taken from Elements of Spacecraft Design [1] is shown in fig. 1.1. Spacecraft systems are partitioned in this fashion so as to allow each subsystem to be designed, developed, and tested by experts in each respective subsystem area.

We define in general terms the role of each of the subsystems shown in fig. 1.1. The exact number and kind of subsystems can vary from spacecraft to spacecraft, and mission

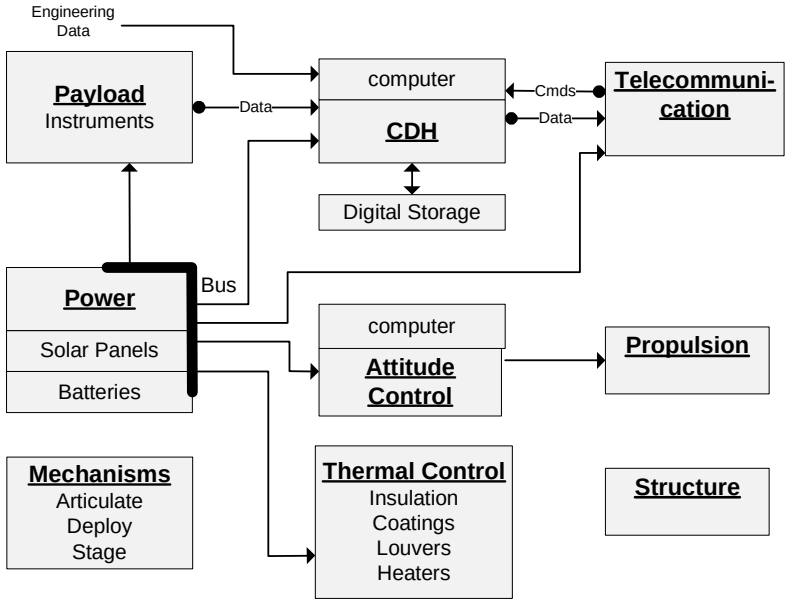

Fig. 1.1: Example spacecraft block diagram.

to mission, but the following tend to appear commonly in small satellites.

- **ADCS:** This subsystem controls the attitude, which indicates "how a spacecraft is oriented in space" [2]. The Attitude Determination and Control Subsystem (ADCS) takes care of estimating the current attitude and makes attitude adjustments as needed. This ability is very important for a spacecraft because through out its lifetime, it will need to assume various attitudes for supporting (e.g., pointing the payload in a specific direction, angling the solar panels toward the sun, pointing cameras toward a specific location, etc.)

- **CDH:** The command and data handling (CDH) subsystem is responsible for commanding the rest of the subsystems, collecting telemetry data (spacecraft health and scientific data) from them, and managing interactions with the ground station. It typically consists of a processor with software running on it and memory to store both the telemetry data and sequences of commands to be issued.

- **Power:** This subsystem takes care of generating, storing, and distributing the power required for the spacecraft to carry out its operations. Power generation can be done

through primary batteries, solar panel powered batteries, nuclear fuel, etc., depending on the design. If solar panels are used, the current produced from them is used to store charge into a battery. Most recent small spacecraft use solar powered batteries, which implies that the available power can fluctuate depending on the solar flux available to the panels.

- **Telecommunications:** A spacecraft that operates in isolation without communicating with a ground station serves little use. The telecommunications subsystem takes care of the uplink and downlink communications with the satellite, making use of multiple antennas, encoding and decoding systems. Typically, the uplink channels carry commands to the spacecraft and the downlink channels carry data acquired by the satellite's instruments and on-board status monitoring.

- **Payload:** This subsystem is mission-specific and typically has a science-based instrument or a communication device. For instance, the payload for a remote-sensing satellite could be a camera. Some spacecraft payloads host multiple devices, depending on the mission.

- **Thermal Control:** The hardware used in the spacecraft and the batteries that store power, all have their own operating temperature ranges which must be maintained. Monitoring the temperature and regulating it via active and/or passive methods is important to ensure the functioning of the spacecraft hardware. The thermal control subsystem manages these functions.

- **Propulsion:** This subsystem deals with the fuel and thrusters needed to deploy and maintain the spacecraft in its orbit.

- **Structure and Mechanisms:** This subsystem provides physical support for the rest of the subsystems and facilitates the deployment of booms, antennas, science instruments, and solar arrays.

1.1.3 Formal Methods

Formal methods refer to the techniques and languages used to mathematically specify systems. This discipline of software engineering has tremendous application potential in the field of systems requirements, design, and verification. The relevance of formal methods is more fully established because of the increasing complexity of systems, since with increased complexity comes higher likelihood of errors. These errors typically are a result of the fact that the number of internal states of a system grows combinatorially with system complexity. Since scenario-based testing cannot comprehensively cover the entire state space of such systems, such systems mandate the use of formal methods to assure the requisite level of reliability.

Formal methods can be defined as *"mathematically rigorous techniques and tools for the specification, design, and verification of software and hardware systems"* [3]. While there is value in merely expressing a system and its requirements in a formal, unambiguous way, the power of formal methods can be enhanced by using specialized analysis tools called model checkers. A model checker is a computer program which accepts as input the system specification, conforming to a predefined set of formal rules or notations (referred to as the "modeling language"). Different model checkers perform different tasks, depending on the modeling language and application. However, typically, their role is to analyze the provided system specification for consistency with a set of captured requirements or rules. The model checker either exhaustively examines the state space of the application to prove the absence of inconsistencies, or alternatively, provides the user with a counter-example, showing a valid system state, which is inconsistent with the specified rules.

It is not uncommon to have multiple formal models for the same system that address different system properties [4]. Some aspects of system behavior may require timing specifications, while for others, a simple sequencing specification may suffice. The choice of modeling language and supporting verification tool depends on the system properties and behaviors of interest. Various modeling languages exist, including PROMELA [5], CSP, Petrinets [6], ASM [7], etc. Different model checkers support different modeling languages;

however, the verification approach is consistent for all formal languages. The system and its associated requirements are expressed in the modeling language. The corresponding model-checker then performs state-space exploration to search for any scenario that does not conform to the given requirements.

Unfortunately, even a formally verified system can result in erroneous behavior. This is due to the fact that the formal specification is not the actual implementation specification. The modeling language and the implementation language belong to completely different domains. Further, the modeling of a system in a formal language typically involves abstraction, simply because modeling every part of the system may not be useful and there are limits to the size of state space the verification tools can handle. There are no absolute rules that dictate the appropriate level of abstraction to employ for system modeling. Identification of the appropriate set of system properties, and the creation of an appropriate system model corresponding to those properties is a highly subjective process. Despite all these factors, it is worthwhile to employ verification because given the proper model and the proper specifications, a model-checker can ensure that those specifications hold for the system.

1.2 Motivation

In recent years, a class of spacecraft much smaller in size, but which can still perform meaningful science operations, are gaining popularity. These are called small satellites and have generated a great amount of interest in the academic, industrial, and government satellite design communities due to the potential expected of these devices. Small satellite design generally consists of the application of various sizing procedures, definition of high-level interfaces, and a detailed design of each of the subsystems. The design and testing within each subsystem has traditionally received greater attention as compared to the interactions between the subsystems. Though such an approach appears reasonable, given well-designed high level interfaces, it unfortunately fails to recognize the importance given to system-level behavior. Often system-level behavior is not composable in nature, and cannot be easily discerned through analysis conducted at the subsystem-level. The spacecraft

is thus more vulnerable to discrepancies in the system level behavior which could be the result of a deadlock, an untested scenario, livelock, etc. Due to this, informal spacecraft designs when tested only after the implementation phase can result in tedious design changes late in the design process. The ensuing process consists of identifying the problem, making the necessary design changes and going through rounds of system-testing. Small satellites cannot afford such lengthy development processes. Further, even after all these steps there could be undetected errors in the system. Formally defining the subsystem interfaces, along with their associated behavior, and subsequently verifying the resulting system-level design could potentially reduce design integration errors significantly. This idea is a major shift from the current practices and mandates the employment of formal methods in the design and verification process.

Employing formal methods involves a translation of the system and its requirements into some modeling language, whose semantics are grounded in mathematics. Consequently, the general trend is to have someone proficient with the modeling language develop the system model, which requires significant interaction between the system designer and system modeler. Unfortunately, this leads to confusion on the part of the systems engineers, due to lack of expertise in formal languages and system modeling, and may introduce specification errors on the part of the formal methods expert due to lack of understanding of the application domain. This disparity tends to hinder the adoption of formal methods by the spacecraft design community. The goal of this research is to develop domain-specific automated graphical tools that make formal modeling accessible to spacecraft systems engineers. The graphical tool presented in this thesis, BASS (Behavioral Analysis of Spacecraft Systems), provides a domain-specific visual modeling tool for representing small spacecraft behavior. The visual modeling language is strongly typed and is based on formal semantics, facilitating the automatic translation of the captured spacecraft system into a formal modeling language, to which model checking can be applied.

1.3 BASS: Behavioral Analysis of Spacecraft Systems

This thesis claims that it is possible to create a domain-specific visual modeling tool

targeting the design and verification of system-level small satellite behavior, thus enabling earlier detection of design errors in the development life-cycle. We offer BASS as a tool to realize this goal. This thesis documents the BASS toolflow, which is organized into the following chapters:

1. Related Work and BASS Toolflow: This chapter discusses similar work done on earlier occasions, introduces the software tools utilized in creating BASS, followed by an end-to-end toolflow explaining its usage.

2. Modeling Generic Subsystem Behavior: This chapter identifies the behavior common to all the subsystems and introduces modeling constructs (as GME meta-models) that can adequately describe them.

3. Modeling Subsystem Specific Behavior: This chapter delves into the modeling constructs that are needed to describe the custom behavior of each of the spacecraft subsystems. The ADCS, CDH, Power, Payload, Uplink, and Downlink subsystems are visited in this section.

4. Spacecraft System Verification: This chapter initially discusses how a composite spacecraft system is built using the modeling constructs introduced in earlier chapters. An approach to requirements modeling and the meta-model for the same are also discussed.

5. BASS Interpreter and CSP Semantics: This chapter discusses how and what kind of CSP semantics are assigned to the various visual modeling constructs of BASSMP. Relevant examples to demonstrate how the BASS interpreter does the translation are also presented.

6. Case-Study for BASS: This chapter explains how few parts of TOROID's design are translated into BASS models. Useful requirement checks that verify the system behavior are composed in BASS and their verification results in FDR are discussed.

Chapter 2

Related Work and BASS Toolflow

Significant work has been done in the area of spacecraft behavior verification and the development of graphical modeling tools that support system verification. In this chapter, we examine work done in the areas of spacecraft verification and automated graphical tools which is relevant to BASS. The academic modeling tool GME and the process algebra CSP are introduced following which the end-to-end toolflow for BASS is presented.

2.1 Related Work

2.1.1 Formal Methods Applied to Spacecraft

Several researchers have examined the use of formal methods for verifying spacecraft systems. The most relevant work on this front at the system-level has been done by McInnes [8]. His approach was based on CSP (Communicating Sequential Processes) semantics and concentrated on developing a CSP library and generic constructs that can define system-level interactions. An FFBD approach to specifying and verifying high-level spacecraft behavior was also developed [9], which provides a generic visual modeling environment coupled with semantics rooted in CSP.

Verification of flight software running on NASA's Deep Space 1 brought to light a few design level problems [10]. The model-checker SPIN was used to verify the models derived from a copy of the code running on the spacecraft. The software had been running for years without ever exposing the detected problems. This effort reiterates the necessity of formal methods. In another effort, the health monitoring system on Deep Space 1, called Livingstone, was formally verified using the model-checker SMV [11]. An automated translator program was developed to convert the MPL declarative formal definitions used

to describe Livingstone into CTL, which was then verified by SMV.

Formal methods were applied to requirements engineering in the fault-protection area for the International Space Station by Easterbrook et al. [12]. This was carried out using the PVS theorem prover with the goal of making the requirements less ambiguous. Though this project was initiated post-launch, it provided important guidelines as to how formal methods can be utilized in the early design phases to resolve ambiguities earlier and steer the design/implementation in the right direction.

Autonomous planners flown in missions like Earth Orbiter 1 and Deep Space 1 were verified by Smith et al. [13] using the SPIN model checker. The plan of action in these spacecraft is decided dynamically on-board based on the occurrence of faults or unexpected execution results. The planning system was modeled using resources, activities, states, and requests in PROMELA (PROcess MEta Language). The resulting verification process succeeded in finding a few problems with the planner.

Most of the earlier attempts at incorporating formal methods into spacecraft development life-cycle concentrated on isolated functional blocks, with little emphasis on system-level analysis.

2.1.2 Graphical Modeling Tools

Graphical tools that can automatically generate formal models have been developed and documented. Hilderink [14] developed a graphical modeling tool that has constructs for representing system behavior, and generates machine-readable CSP. The generated CSP can be model-checked in FDR. The language constructs target generic system behavior and are CSP-specific, mandating a deep knowledge of CSP. Another graphical language-to-CSP translation is introduced by Muan and Butler [15]. They assigned CSP semantics to the widely used UML Statecharts, thus allowing for the formal verification of finite state machines used in any system.

A graphical interval logic for specifying concurrent systems, developed by Dillon et al., has been used to develop a system verification tool called the GIL toolset [16]. It aims at providing an intuitive notation for expressing system specifications using intervals of time

delimited by conditions. The proof checker and model generator associated with the logic helps in later verification. The tool framework has no means of specifying actions that occur in systems, which are important for describing spacecraft behavior.

Specification Description Language (SDL) is another graphical specification language which uses formal methods [17]. SDL is based on Finite State Machines (FSM) and can be used to describe system behavior. However, it is more widely used for telecommunication systems and to our knowledge, has not been applied widely to spacecraft. However, it has been applied to the validation of fault tolerance in the design of autonomous spacecraft, examining in particular the Data Management System [18].

In general, the graphical languages developed to date which support formal methods do not offer a domain-specific modeling syntax familiar to spacecraft systems engineers.

2.2 GME

The academic tool GME (Generic Modeling Environment), developed at Vanderbilt University represents a meta-programmable toolkit for creating domain specific visual modeling languages. GME also supports the development of interpreter tools, analogous to a compiler for a textual programming language, for supporting the extraction and manipulation of information captured in the models.

To construct a custom modeling environment using GME, the first step is to identify all the modeling entities that can adequately express the system to be modeled. In GME, these are classified into one of the three types: atoms, models, and connections [19]. Atoms are the basic parts which cannot contain any other entity within them, models are compound entities made up of atoms, models, and connections. Connections are used to express association between atoms and models. Relationships between these entities are defined through the use of aggregation and inheritance. Attributes and connections between them to indicate sequencing can also be defined. A collection of similar parts can be included as part of a library. A structured collection of these modeling entities along with the relationships between them is called a meta-model. A meta-model is a model of the modeling language itself. It defines the syntax (relationships between all the modeling objects) for

the visual model a user can make. The meta-models created in GME are similar to UML class diagrams, with some added features like hierarchy and aspects. Hierarchy within a meta-model is an effort to reduce the visual clutter and make the model more manageable by allowing objects to contain other objects. This introduces the obvious problem of how to connect objects at different depths in the hierarchy. References, which are similar in concept to C "pointers," represent a placeholder for another object, and can be used to support cross-hierarchy connectivity. Aspects in GME are used as a visual partitioning for a model. They can be used to realize the concept of separation of concerns while modeling. They not only reduce the visual clutter but also make make the modeling environment more meaningful by allowing the user to address one aspect of modeling at a time.

Interfaces to the modeling entities can be represented by categorizing a subset of its contained components as ports. Any model, atom, or reference can act as a port. Ports are visualized as parts on the edges of the container when viewed from the perspective of the container's parent. Connections can terminate on, or emanate from, either the object or the object's ports, depending on the syntax specified in the meta-model.

All the modeling rules in a GME meta-model are embedded into a configuration file called the paradigm. GME creates the paradigm from the meta-model with a tool called the MetaGME Interpreter. Once a paradigm is available, the user can create models that conform to the rules defined by the meta-model. The power of GME comes from the fact that we can have an interpreter corresponding to a paradigm. GME offers multiple APIs or interfaces that facilitate the creation of interpreters. GME allows an interpreter to access the information captured by the user when drawing models. Interpreters apply semantic translations, performing such tasks as code generation, model-to-model transformations, or model analysis. Multiple language bindings, including C++ and Java, are supported.

2.3 CSP

Communicating Sequential Processes is a process algebra conceived by Hoare [20]. It was developed specifically to describe systems that have concurrent components communicating synchronously/asynchronously among themselves and with their environment. CSP

offers a rich set of expressive operators, which have well defined execution semantics. The basic entities used in CSP are processes, channels, and events. CSP considers processes as "independent self-contained entities with particular interfaces through which they interact with their environment" [21]. These interfaces are called channels. An event can be described as a particular action that can be performed by a process or an environment. It could be the occurrence of an interrupt, reception of a message, generation of an error, etc. Related events can be grouped together to form a datatype and a channel can be associated with a datatype. Channels which are not accompanied with a datatype or a listing of events can be considered as events themselves. The concept of an event is important in CSP because behavior is expressed as a sequence of events. In the following sections, we examine some of the basic features of CSP. A comprehensive accounting of CSP is beyond the scope of this thesis. Interested readers are referred to textbooks by Roscoe [22] and Schneider [21] for more information.

2.3.1 Classifying Events

A process is always associated with an alphabet, a set of events associated with the process. Events are classified either as external events (given by the environment), internal events (which are generated by a process and are not visible to the rest of the processes), or a termination event (indicating a successful termination of a process). It is common practice to specify an interface set for each process that consists of all the externally observable events. Any events that need to be suppressed because they are of no interest to the rest of the processes are made internal using the hiding operator "\." For instance, the process $Proc \backslash \{internal\_channel\}$ implies that all the occurences of *internal_channel* events are replaced by the internal event τ.

2.3.2 Choice Between Events/Processes

Future evolution of a process can be controlled by itself or by another process or the environment. From a system modeling point of view, this implies that a component is controlled by the behavior of another component or by its own behavior. CSP supports two

choice operators for these two cases: internal choice ⊓ and external choice □. In the case of internal choice, control lies within the process and not with any other entity. Hence, it is often called nondeterministic choice.

2.3.3 Combining Processes

Processes can be compositional; two or more processes can always be combined to form a larger process. This support for hierarchical composition of systems is very useful as one can build models of a large system out of smaller, simpler processes. When processes are composed together, there can be situations wherein a process needs to synchronize with others or proceed independently. CSP provides four kinds of parallel combination of processes to manipulate the events composed in a process. They are Synchronous Parallel, Alphabetized Parallel, Generalized Parallel, and Interleaved combinations.

2.3.4 Example

We provide an example CSP snippet here which has defintions for three processes. They are StudentProcess, ParentProcess, and an alphabetized parallel combination of the two, StudentAndParent, which synchronizes on the intersection of the sets StudentInterface and ParentInterface. Study or play actions is performed by the student based on the selection made by parent due to the presence of external choice. However, whether the student reads English or science is up to him, owing to the internal choice between these two events.

```
channel : study, play
datatype :readType = english | science
channel read : readType

StudentProcess = study -> StudyProcess
                 []
                 play -> STOP
StudyProcess = read.english -> ReadEnglish
               |~|
               read.science -> ReadScience
```

```
ParentProcess = study -> STOP
                |~|
                play -> STOP
StudentInterface = {study, play}
ParentInterface = {study, play}

StudentParentSet = {(StudentInterface, StudentProcess),
                    (ParentInterface, ParentProcess)}
StudentParent = ||((Interface,Process):StudentParentSet
                    @ [Interface] Process)
```

2.3.5 Verification Support for CSP - FDR

A system representation is not very useful unless it can verify user-modeled requirements or constraints. These are modeled at a higher level of abstraction compared to the system and generally look at one behavioral requirement at a time. The most important feature of CSP that supports verification is refinement. The process that represents the system, called implementation, can be checked for certain requirements called specifications. If all the runs of the implementation process adhere to the specification process, we say the implementation refines the specification. Aside from the user-specified checks, the model checker for CSP FDR can check for generic deadlock and livelock situations.

$$Specification \sqsubseteq Implementation$$

2.4 BASS Toolflow

The end-to-end toolflow of BASS is shown in fig. 2.1. GME is chosen for the front-end of BASS because its features enable creation of complex system models and generation of semantics out of them. We call the spacecraft meta-model created in GME as BASSMP (Behavioral Analysis of Spacecraft System Modeling Paradigm). The visual model created in BASSMP needs to be translated into a modeling language so that the spacecraft design can be verified. CSP is chosen initially for this purpose because of its inherent ability to represent concurrent systems like spacecraft. The work performed by McInnes [8] to verify

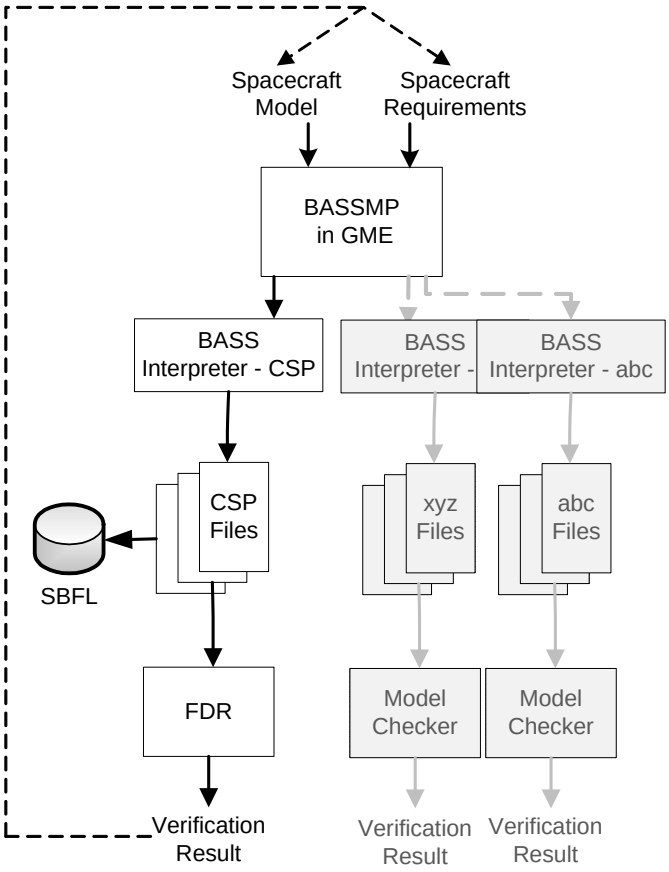

Fig. 2.1: BASS toolflow.

system-level behavior of spacecraft reiterates this claim. The CSP library and the case studies provided in his dissertation provide a good idea of the desired semantics and a good start to implement BASS.

The information captured as part of the user model in BASSMP is utilized to create the formal modeling language description of the spacecraft system via the BASS Interpreter. BASSMP is intended to be specific to the domain of spacecraft system behavior for small satellites, but independent of any particular formal language. However, the interpreter that carries out the translation must be specific to a formal language. We have selected the process algebra CSP as a target formal language for the BASS model interpreter, shown below. The dashed flow output from BASSMP indicates that other interpreters, targeting different formal languages could be constructed in support of this toolflow, but are not considered in this thesis.

Spacecraft requirements represent a second input to the tool, as shown in fig. 2.1. These are the high-level requirements specifications that are again captured in BASSMP, mostly reusing the constructs defined in the spacecraft model. The model-checker for CSP, FDR (Failures Divergences and Refinement) is used for verification. The output of the tool is the verification result, which says whether the requirements are met in the current design. In case of failure, FDR provides a counter-example containing the trace of events that leads to a failure. A failure implies problems with the current design or with the requirements specification. Once the failure is discovered, the user can go back and change the models and/or requirements specification until the requirements are met.

Chapter 3

Modeling Generic Subsystem Behavior

Systems can be visualized in two aspects according to Oliver, et al. [23]. The structure aspect captures how the system is composed and the behavior aspect represents what the system does. Capture of behavior in system models involves the identification of the cause and description of effect of all the conditions that bring rise to some reaction in the system. A model is "a pattern of something to be made" [23]. By this definition, block diagrams of a system found in design documents qualify as system models. Modeling a system may involve the development of only the structural models or only the behavior models or a mix of both. In order to intuitively decompose and represent system behavior for a spacecraft in the visual developed syntax, we adopt a mix of both structure and behavior modeling.

BASS facilitates the capture of a spacecraft as a collection of subsystems, similar to the decomposition represented in fig. 1.1. We limit the specification of spacecraft structure in BASS to the identification of the subsystems. While the individual subsystems have structural concerns which could be captured, such detail is unneeded for system-level behavioral modeling. Oliver et al. [23] discuss what constitutes behavior and what elements are required to describe behavior: "Behavior for a system describes what the system is to do, independent of how the system will do it." We follow the same approach when modeling behavior for each of the subsystems in the spacecraft, by focusing on the representation of what a subsystem does without concentrating on how such behavior is achieved.

Since BASS is intended to be a domain-specific tool that has modeling elements recognizable by a spacecraft designer, as opposed to a software engineer, appropriate spacecraft-specific visual modeling constructs need to be developed that cover all the basic elements of behavior. To create such a custom spacecraft modeling environment, appropriate meta-models for all subsystems need to be made. When representing spacecraft behavior on a

subsystem-by-subsystem basis, a handful of concepts and constructs are employed in several of the subsystem models. We proceed by identifying and modeling those interfaces and functions which are common to multiple subsystems, followed by the capture of constructs which are exclusive to each subsystem. In this chapter, we focus on modeling the constructs which are shared between subsystems. These constructs are categorized based on functionality into areas of interfaces and behaviors for control, power, and fault-handling.

3.1 Subsystem Control Interfaces

The control interfaces generally found in spacecraft design documents include commands, messages, continuous data streams, global variables, etc. McInnes [8] characterizes these interfaces as explicit interfaces, since they are explicitly mentioned in the design documents. He also defines implicit interfaces as "... a point of interaction which results when the behavior of a subsystem depends upon a physical state that is under the control of another subsystem, or of the environment" [8], and uses as an example of an implicit interface the attitude of the spacecraft. The spacecraft generates power depending on the orientation of the solar panels with respect to the sun. Such a dependency is based on a state of the spacecraft, and are not typically communicated explicitly during operation. We factor the representation of implicit interfaces into the meta-models. In this subsection, we cover explicit interfaces, and later define shared state objects which are used to represent implicit interfaces. BASS supports four classes of explicit interfaces for spacecraft control: Commands, Telemetry Data Streams, Messaging, and Shared Values.

3.1.1 Commands

A command is a specific type of message sent either from a ground station to a spacecraft, or between spacecraft subsystems, which mandates some sort of action. The set of commands employed depends to a large extent on the nature and design of the the mission. During subsystem design, developers produce an exhaustive list of all the commands that each subsystem can accept. These commands can be discrete (e.g., to pulse or activate relays), or serial, consisting of a sequence or group of commands [24]. Serial commands

stored in memory that get triggered in response to a timer or in response to some event implement a form of autonomy, allowing the spacecraft to react and make decisions without direct interaction with the ground station.

The representation of a discrete command is straightforward, simply requiring the definition of an object representing the command. Modeling serial commands, in contrast, requires provisions for describing the sequence of commands that makes up the serial command, a mechanism to load and unload the sequence from memory, and a means of pausing the sequence execution. Sets of similar commands, regardless of type, can be grouped to form "parametrized" commands. For example, a command to switch on the ADCS and another to switch it off can be grouped to form a single command that has two parameters named "on" and "off." Such parameterization is accomplished using Symbols within the command.

In BASSMP, each subsystem has a CommandSet which contains definitions of all the commands accepted by that subsystem. Such commands include discrete commands (which are objects of type Command), and serial commands (captured as objects of type CommandSequence). The CommandSet contains both commands and references to commands (CmdRefs). CmdRef objects are used in the context of Uplink subsystem's CommandSet to indicate that issue of a few subsystem commands is controlled by the Uplink which inturn is dependent on their receipt from a ground station. Connections between command objects are used to define CommandSequences. Commands are further categorized into SimpleCommands, CommandSequences, and CommandSeqCmds as shown in fig. 3.1. These are analogous to the discrete commands, serial commands, and the commands which control the serial commands. SimpleCommands can have custom Symbols the user defines, in addition to the predefined "on" and "off" symbols. Such custom symbols are used to define parametrized commands. The regulating command for a CommandSequence, CommandSeqCmd, contains four predefined symbols load_seq, run_seq, stop_seq, and unload_seq. These Symbols indicate the type of action to be taken with the corresponding CommandSequence. Each CommandSequence command is associated with exactly one CommandSeqCmd object.

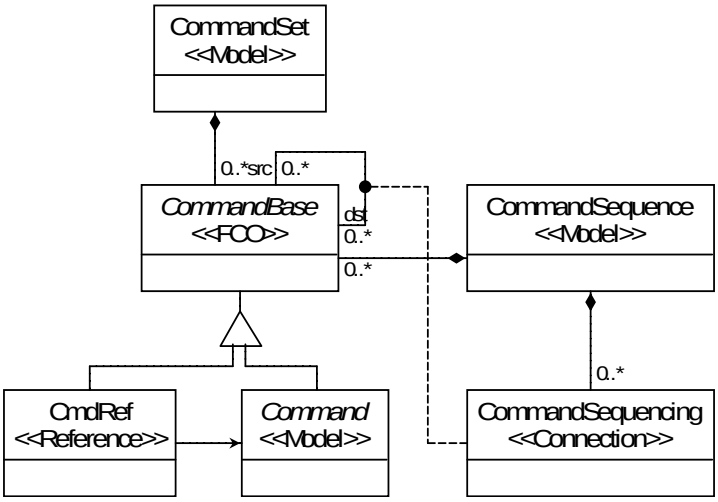

Fig. 3.1: Meta-model for the CommandSet.

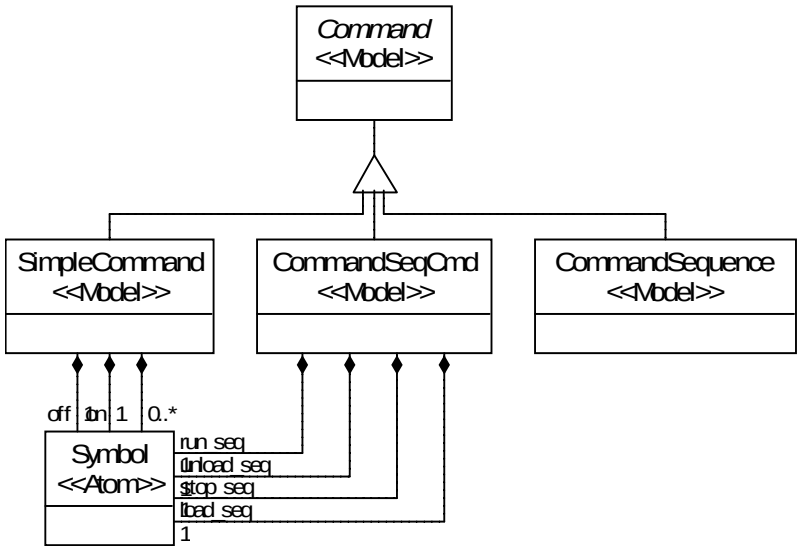

Fig. 3.2: Meta-model for the Command.

Example: The commands defined within a CDH CommandSet are shown in fig. 3.3. SwitchADCS is a parametrized SimpleCommand used to switch the ADCS on and off. StandbyCmdSeq is a CommandSequence that is controlled by the StandbySeqCmd CommandSeqCmd object.

Figure 3.4 shows the internal definition of the StandbyCmdSeq CommandSequence, consisting of two commands which occur in succession. The first, PayloadStandbyCmd, is issued to the Payload subsystem to stop any science operations by moving the Payload into

Fig. 3.3: Example CDH subsystem CommandSet.

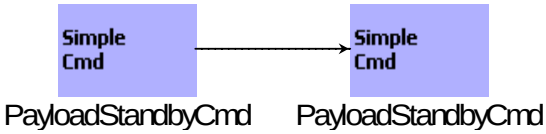

Fig. 3.4: Definition of StandbyCmdSeq.

a standby state. The second is issued to the ADCS subsystem, and brings the ADCS to a standby attitude. Such a sequence can be issued when a science attitude pointing error occurs, thereby stopping the science operations.

3.1.2 Telemetry Data Streams

Telemetry data in a spacecraft includes any continuous information produced by the spacecraft, which may be of interest to other subsystems or the ground station. Such data is typically collected and managed by the CDH subsystem. Generally, this data is processed and stored by the CDH for subsequent downlink transmission upon a ground station contact. A comprehensive modeling of all kinds of data and subsequent transmission is not attempted as discussed in sec. 4.1. In BASS, we discretize continuous data streams by capturing only changes in stream state between user-defined levels, as discussed in sec. 4.1. Telemetry data streams in the context of BASSMP represent the continuous flow of state information, which is stored in a SharedState object (defined in sec. 3.2.2), transmitted from one subsystem to another. Subsystems that generate/utilize a telemetry data stream should create a reference to it and register with the appropriate interface. As indicated by fig. 3.5, TelDataStreamLib represents a collection of a set of Telemetry Data Stream objects. Each TelDataStream object is associated with a SharedState object, which holds the system state information streamed by the data stream, via containment of a SharedStateRef object. TheSharedStateRef is a reference to a SharedState object. The TelDataStream contains

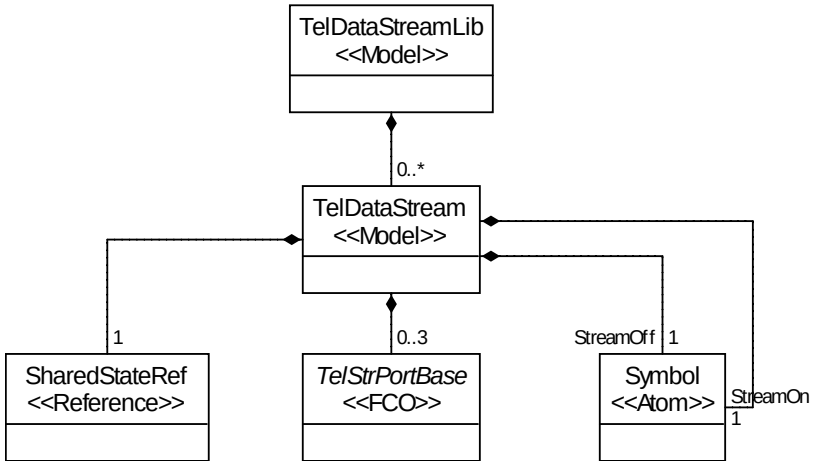

Fig. 3.5: Meta-model of TelDatastream.

two Symbols StreamOff and StreamOn, which are used to command the stream to start and stop, respectively, streaming data. These typically correspond to the "on" and "off" symbols that are part of the power interface of the subsystem whose state is being relayed. Up to three ports of type TelStrPortBase are available on the TelDataStream to facilitate interfacing to the stream. Figure 3.6 defines the three types of data stream ports: TelStrGet is used to query the current stream value, TelStrSet is used to set the current stream value, and TelStrTrans is used to receive notification of when the data stream changes state.

3.1.3 Inter-Subsystem Messages

Messages that flow from one subsystem to another are common in spacecraft design.

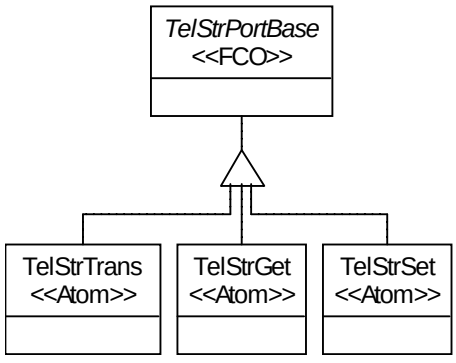

Fig. 3.6: Meta-model of the TelDataStream ports.

These messages allow for asynchronous/synchronous communication between subsystem modules, depending on the design. In order to model the occurrence of certain non-deterministic behaviors, implying which can occur at any time or are dependent on environmental factors, and to model scenarios when inter-subsystem communication occurs, messages are generally used. BASS defines such inter-subsystem messages as PointToPointMsg objects.

PointToPointMsg objects are atoms that can be contained in a subsystem. They model messages which are used to acknowledge a command received or communicate to other subsystems a deviation from normal behavior. Some of these messages can have additional semantics associated with them based on the subsystem in which they are defined and are discussed in the next chapter.

3.1.4 Shared Values

Values shared between subsystems, stored as status-indicating flags in memory, are common in spacecraft design, but are treated differently from explicitly passed messages. Shared values generally imply a state or the status of an activity and restrict or enable the progress of some behavior of a subsystem. These shared values mostly are owned by a subsystem and any changes to these are communicated through messages over the data bus. Access to Shared Values is typically restricted via semaphores. We do not model the synchronization details as that leads to too may states to be handled for the model checker. In BASS, we refer to these Shared Values as Memory_Units. When modeling the Memory_Units, details of ownership are modeled by enabling a specification of the subsystem that writes to it. The message-passing is abstracted using the "set" and "get" channels of the Memory_Unit as we see in the meta-model below.

The meta-model for the Memory_Unit is as shown in fig. 3.7. This shows the interfaces to the Memory_Unit Read, Write, and Trans. The Read and Write interfaces are self-explanatory; the Trans interface becomes active whenever the value in the Memory_Unit changes. AllowedValues, defined in more detail in fig. 3.8, embodies all possible values (represented as Symbols) the Memory_Unit can take on. The default value assigned to a

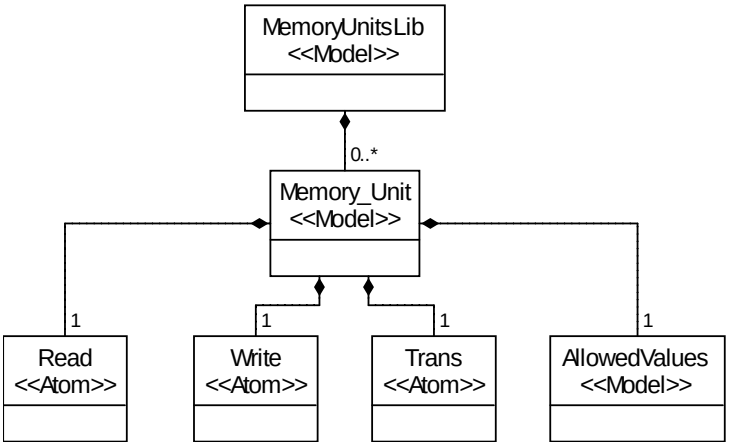

Fig. 3.7: Meta-model of Memory_Unit.

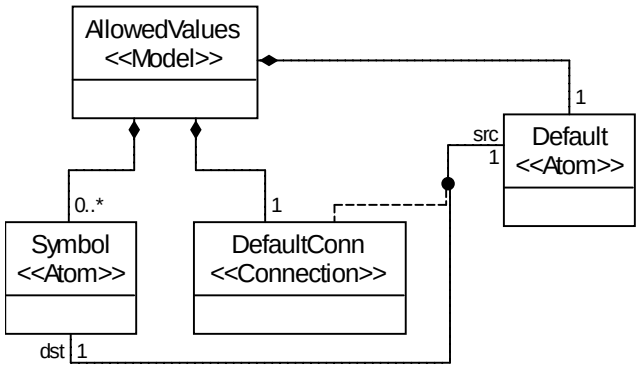

Fig. 3.8: Meta-model for AllowedValues.

Memory_Unit object is specified by the DefaultConn connection, which connects a Symbol to the Default atom.

3.2 Subsystem Control Behaviors

In addition to the control interfaces described above, subsystems also define constructs modeling control behaviors. Mode-based behaviors for a subsystem or one of its components are commonly found in spacecraft systems. A behavioral mode change can cause changes in certain states internal to the subsystem, or alternatively, a change in certain system-level shared values can result in a change in the mode of operation. In this section, we describe modeling constructs that enable the user to define different kinds of subsystem behavior.

3.2.1 Functions

A function represents a relationship which maps an element from an "input" set onto an element from an "output" set. Such relationships occasionally must be defined when creating spacecraft models in BASS. Functions may be used to describe an interdependency or simply to assign values to some qualitative ranges. The MapFunction construct enables the description of one-to-one or many-to-one relations.

The MapFunction, defined in fig. 3.9, can contain a set of inputs, fIn, and a set of outputs, fOut. Valid members for these sets include Symbol objects, and in the case of fIn, reference objects, which refer to Modes of a Mode Transition System (defined in sec. 3.2.3). The mapping between inputs and outputs is defined using the SymbolMappingConn and ModeRefToSymConn connections, represented in fig. 3.10.

3.2.2 Shared States

Internal spacecraft behavior depends to a large degree on the physical spacecraft state (e.g., attitude, temperature, power level, etc.). Subsystem behaviors can both be governed by spacecraft state, and in some circumstances, can control spacecraft state. Often, the interactions between spacecraft state and a subsystem depends on the subsystem being considered. Such interactions are covered in Chapter 4. However, the general structure of shared state objects is defined here, along with the facilities for accessing a shared state

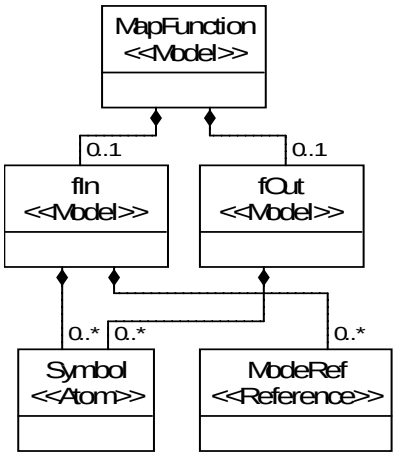

Fig. 3.9: Meta-model of a MapFunction showing its input and output contents.

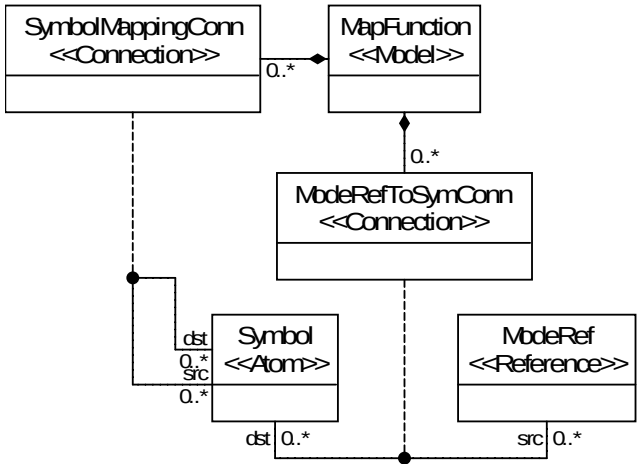

Fig. 3.10: Meta-model of a MapFunction showing connections between inputs and outputs.

object.

Figure 3.11 shows the the model of a SharedState which contains well defined interfaces (via the SSSet, SSGet, and SSTransports) for accessing the state, as well as the AllowedValues model, representing the set of values a SharedState object is allowed to take on. Similar to the Memory_Unit, SharedState objects support "set" operations, where the state value is updated, "get" operations, where the state value is queried, and "trans" operations, through which notifications of changes to the shared state object are broadcast.

3.2.3 Mode Transition System

Spacecraft behavior is modal in nature. Modes represent stages in the execution of the spacecraft where distinct behaviors are exhibited. The spacecraft remains in a mode

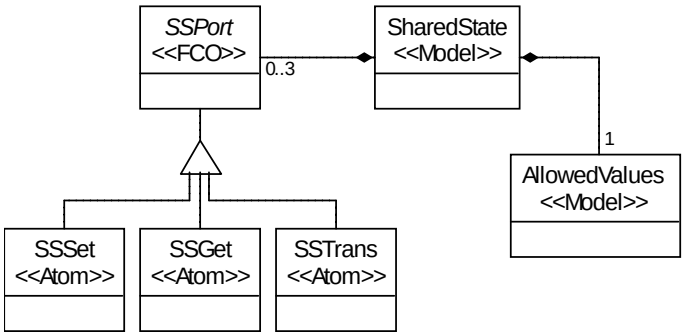

Fig. 3.11: Meta-model of SharedState.

until some conditions are met, which cause it to transition to a different mode. Typically, systems engineers model spacecraft modality and mode transitions using some form of finite state machine. An FSM representing TOROID's ADCS manager is presented in Chapter 7. They consist of modes/states connected by arrows that model the transitions. Labels on the transitions represent transition events, or conditions which must occur in order for the transition to be taken.

The meta-model for ModeSystem, shown in fig. 3.12, is designed to appear similar to those depicted in typical spacecraft documentation. It contains Modes, transition objects (ControlTransitions) which define the rules for a transition and a special mode called CommonMode which embodies any transitions common to all modes. Transition connections are used to connect modes to transition objects and vice versa. At times it is desired to associate a SharedState object with a Mode Transition diagram. This allows for representing updates to a SharedState, if any, caused by the actions performed in the previous mode. To enable the capture of this association, Symbols from the AllowedValues of a SharedState object can be associated with appropriate modes using ValueToModeMap connections, as shown in fig. 3.13.

A Mode is analogous to a state whose corresponding actions are performed as soon as

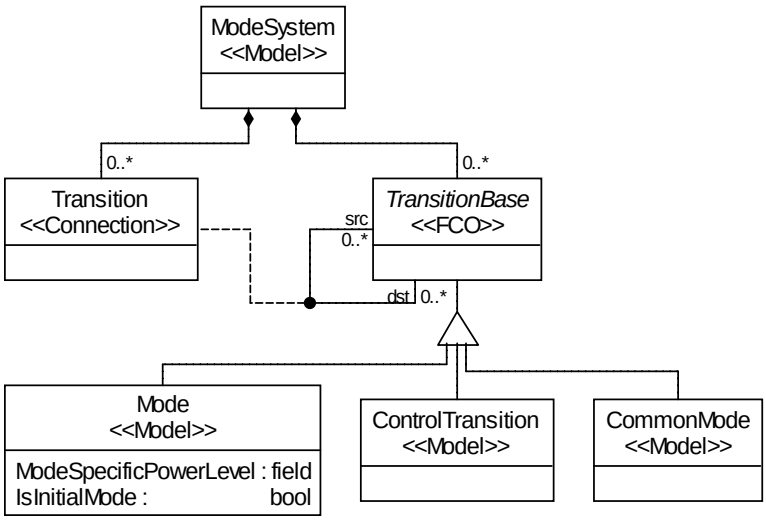

Fig. 3.12: Meta-model of the Modesystem showing Modes and ControlTransitions.

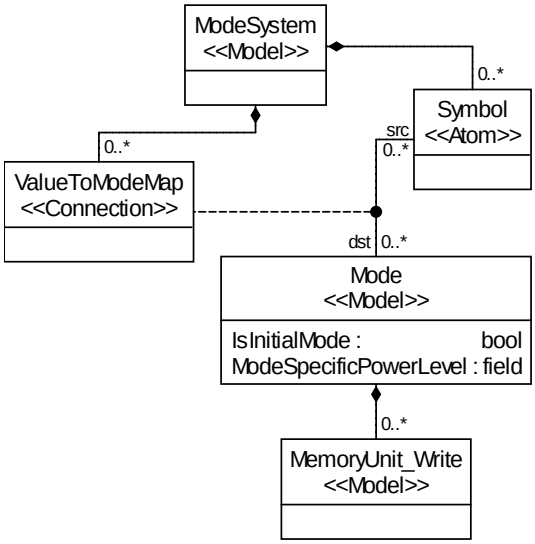

Fig. 3.13: Meta-model for ModeSystem showing contents of Mode and association with SharedStates.

it is entered; ControlTransitions capture the conditions required for mode transition. Such conditions could include waiting for a power switching event, a transition in a SharedState, Memory_Unit, or TelemetryDataStream, or the occurrance of an internal fault generated within the subsystem. The ControlTransition construct is also employed in the definition of the CDH subsystem, and is further refined in sec. 4.1.4. Meeting the condition represented by a ControlTransition is a necessary, but not sufficient condition for the transition to be taken; a transition will only be taken if sufficient power is available. All transitions share an implicit interface to the Power subsystem, which is used to access the current power level. The resulting value is compared against the desired mode's ModeSpecificPowerLevel attribute value, which represents the minimum power required to enter a particular mode.

A ModeSystem typically transitions either in response to an internally generated event, or due to the receipt of a command from the CDH subysystem. BASS supports the use of "response messages," which are messages that a ModeSystem sends to the CDH when it transitions to a new mode. While the CDH module could issue a query to a SharedState object which records the ModeSystem's current mode in order to determine if a transition occurs, the use of a response message more explicitly mimics the actual behavior typically implemented on spacecraft. Each ModeSystem must define a set of RspMsg

objects, representing three particular types of response messages. ModeCmd_Ack signifies that a mode transition command received by the ModeSystem has successfullly resulted in a mode transition, ModeCmd_Rej implies that a transition to a mode requested by a received command is not possible because there is no available transition between the two modes. Power_Insufficient specifies that the system cannot transition to a mode indicated by a received command, due to lack of available power.

Example: Figure 3.14 provides an example ModeSystem corresponding to a Payload subsystem containing three modes: Unpow, indicating the unpowered state; Science_Idle, indicating that the Payload hardware is ready but data is not currently being collected; and Collect_Data, indicating that the hardware is actively collecting data. Unpow is the initial mode for this ModeSystem and is indicated so by setting the IsInitialMode flag to "true" (not shown in the figure). The boxes internally labeled Transition represent ControlTransitions. switch_on waits until an event is received which indicates the Payload subsystem has been switched on; coll_data waits for the receipt of a command from the CDH mandating a transition to the Collect_Data mode; idle waits for the receipt of a command from the CDH mandating the cessation of data collection and return to the Science_Idle mode. The syntax defining these ControlTransitions is discussed in sec. 4.1.4.

In addition to the transitions depicted in fig. 3.14, the Payload needs to transition to the Unpow mode whenever a "switch off" event occurs. This transition applies regardless of what mode the ModeSystem is in currently. Figure 3.15 shows the switch_off transition connecting the CommonMode construct to to the Unpow Mode. The figure also shows the spec-

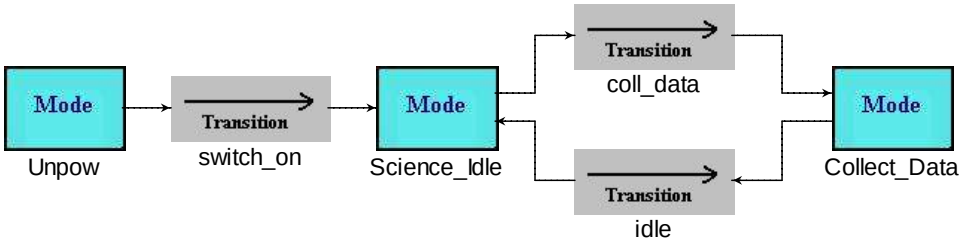

Fig. 3.14: Example Payload ModeSystem.

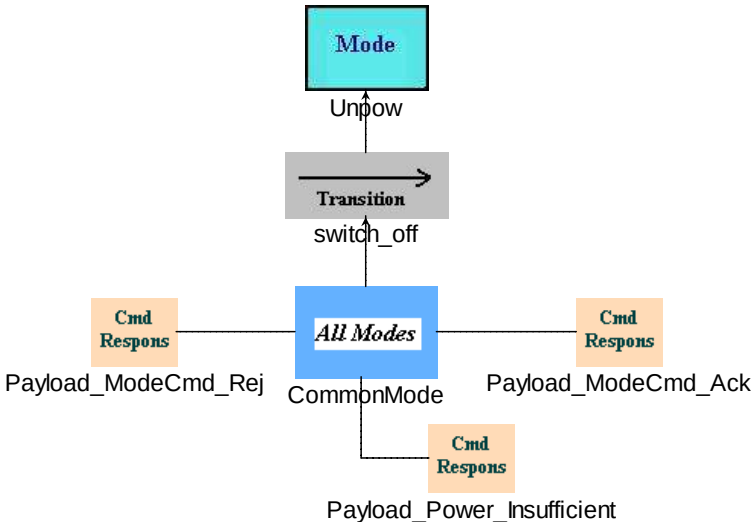

Fig. 3.15: Example CommonMode and common transitions part of the Payload ModeSystem.

ification of the three response messages Payload_ModeCmd_Rej, Payload_ModeCmd_Ack, and Payload_Power_Insufficient. For these response messages to be propogated to the CDH, they need to be associated with PointToPointMsg objects that get sent to the CDH. This is achieved by connecting the RespMsg objects, which appear as port objects of the ModeSystem, three corresponding PointToPointMsgs defined in the subsystem.

3.3 Subsystem Power Interfaces

Any subsystem that requires power must instantiate an interface to the Power subsystem. The subsystems considered in this thesis are all powered; however subsystems like Structure, if modeled, will typically not require a power interface. Depending on the design, a subystem may need a constant amount of power to support all its behaviors, or it may have varying power requirements depending on its modes. For both cases, BASS supports modeling the power requirements and facilitates the verification that sufficient power is available for supporting the spacecraft mission.

Figure 3.16 shows the BASS meta-model that defines the power switching interface SubSysPowerIf that is instantiated in each powered subsystem. The interface contains “on” and “off” symbols which when used in other behavioral constructs like the ControlStates

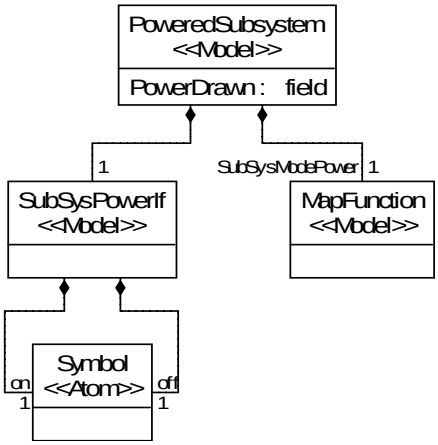

Fig. 3.16: Meta-model for power-related interface and behavior in a PoweredSubsystem.

or ControlTransitions (described in sec. 4.1.4), represent the power switching actions for a subsystem.

3.4 Subsystem Power Behaviors

The SubSysModePower construct, shown in fig. 3.16, is part of a powered subsystem, and enables users to specify the amount of power consumed for each mode of operation in the subsystem. SubSysModePower is a MapFunction whose input set consists of references to modes and whose output set defines different power levels. If a subsystem draws a constant amount of power, this construct need not be used.

3.5 Subsystem Fault-Handling Interfaces

All deployed spacecraft will encounter faults at some point, if allowed to execute for a sufficiently long period of time. A fault can be defined as a discrepancy in the internal working of a subsystem. Faults are generated, not only due to hardware failures due to age or defects, but can also occur because of random or near-random changes in hardware state caused by the space environment. To avoid further hardware damage or to prevent working on erroneous data, faults need to be handled in a timely fashion. This can involve some actions within the subsystem and/or in other subsystems that are affected by the occured fault. Consequently, upon detection, information on some faults must be propogated to the

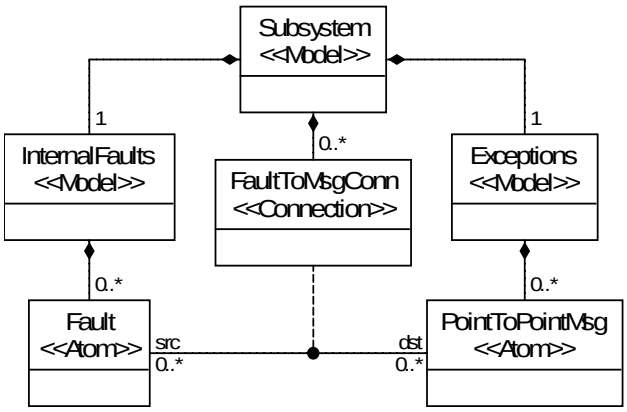

Fig. 3.17: Meta-model for Faults and Exceptions in a subsystem.

CDH so it can inform other subsystems as appropriate. The next immediate concern is how to model the generation of faults when the hardware-level behavior is not modeled in the first place. The approach we follow is outlined by McInnes [8], where the fault-occurence is regulated by an external process. By abstracting the details of how the fault occurs, and simulating the fault via BASS itself through a requirement specification, we verify fault-handling behavior. The faults specified in any of the subsystems do not occur when model-checking the design, unless a requirement specifies their occurence. This approach provides an additional benefit that the model-checker will only consider the occurence of faults, which trigger complete different behavior, if the explicitly examines, via specification in the requirements, fault-handling behaviors. The verification of normal behavior scenarios is thus unencumbered by faults and fault management behavior.

In BASSMP, we define two constructs Faults and Exceptions, together with connections that map faults to respective exceptions. An exception really is a PointToPointMsg object, which is communicated by the subsystem to the CDH. Exceptions have system-level visibility, while faults have visibilty within a subsystem only. All the faults which can be encountered during subsystem execution are enumerated within the InternalFaults model, while the exceptions they generate, which need to be handled by the CDH, are defined within the Exceptions construct. A connection between the two is the FaultToMsgConn as shown in fig. 3.17. This is used to indicate which faults result in the raising of an exception.

Chapter 4

Modeling Subsystem Specific Behavior

This chapter defines additional constructs, specific to individual types of subsystems, for capturing subsystem behavior. In order to more accurately support the modeling of individual subsystems, BASS explicitly identifies and supports constructs for six types of subsystems commonly included in spacecraft designs: Command and Data Handling (CDH), Attitude Determination and Control (ADCS), Payload, Power, Uplink, and Downlink. We discuss the constructs specific to each subsystem individually.

4.1 Command and Data Handling

The CDH subsystem represents the central nervous system for the spacecraft. It is responsible for issuing commands to the other subsystems and collecting telemetry data from them. From a hardware perspective, it typically consists of one or more processor boards and data storage elements [24]. The processor boards typically execute some type of system software. Command handling by the CDH entails the dispatching of commands received from the ground station, as well as the issue of commands or stored command sequences in response to some onboard event. Data Handling refers to the collection, buffering, and relay of telemetry data to the ground station. In some applications, the CDH is tasked with monitoring the health of spacecraft, as indicated by a set of on-board sensors. On the discovery of some problem, the CDH can react by shutting off non-critical operations or initiating corrective action. Such on-board fault-handling behaviors tend to complicate the design making formal verification prior to deployment all the more important.

Telemetry data typically consists of housekeeping data that is needed to monitor the health and status of the spacecraft, attitude data collected from the various attitude sensors used to determine spacecraft position and orientation, and data collected by the spacecraft

payload [25]. BASS supports data handling through a level of abstraction, where only the data that signifies an important change in the internal state of the spacecraft is relayed to the ground station via the downlink. This downlink data is not a snapshot of the spacecraft health and status in terms of numbers, but is qualitative data indicating important internal transitions in the spacecraft.

CDH Meta-Model: The CDH is typically the most complicated subsystem on a spacecraft from a behavior perspective, because it coordinates the activities of the other subsystems. Further, its complexity depends on the level of autonomy employed in the spacecraft design. BASS supports the representation of the CDH subsystem by providing modeling constructs that facilitate:

- Enumeration of all incoming and outgoing commands and command sequences, and specification of rules governing command dispatch;
- Enumeration of continuous streams of data of interest to the CDH via telemetry data stream constructs;
- Enumeration of the data and data handling to support the relay of information to ground station;
- Capture of subsystem interaction and its governance by the CDH using high level control flow constructs;
- Management of subsystem faults communicated to the CDH as exception messages.

Figure 4.1 shows the high-level constructs available in BASS for modeling the CDH. ControlFlow is a construct to represent subsystem interaction. CDHCmdDispatch is used to model command handling and dispatch. DLData and DLMessage objects are used to model data and messages that are relayed to the ground station via downlink. The ExceptionHandlingPort is an interface to the CDH through which exceptions raised by other subsystems are communicated to the CDH.

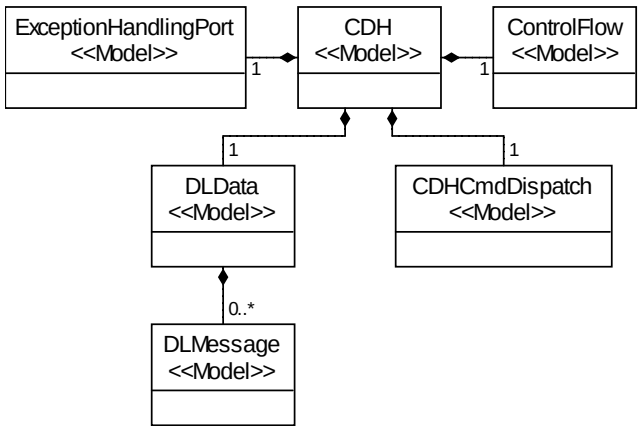

Fig. 4.1: CDH meta-model.

4.1.1 CommandSet

The CommandSet construct was initially presented in sec. 3.1.1. In contrast to the CommandSet employed in different subsystems, the CDH CommandSet consists of commands it issues to other subsystems to direct spacecraft operations, as well as commands it receives from the ground station via the Uplink. The CommandSets defined in other subsystems consist only of the commands that a subsystem can receive.

Example: Figure 4.2 shows an example CDH CommandSet that defines four commands: Switch-ADCS, SetADCSActive, SetADCSStandby, and SetPayloadMode. The SetADCS-Active, SetADCSStandby, and SetPayloadMode commands are issued to other subsystems at appropriate times as dictated by the CDH internal control flow specifications. The SetADCSActive command, in contrast, is issued by the ground station and relayed to the CDH by the Uplink subsystem. The distinction is not depicted visually in the diagram; rather, it is defined through the sharing of a command definition with the Uplink subsystem's CommandSet.

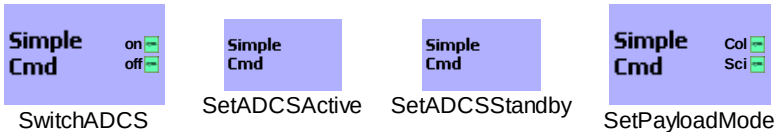

Fig. 4.2: Example contents of the CDH CommandSet.

4.1.2 CDHCmdDispatch

When the CDH receives a command from the ground station, it must determine the appropriate reaction to that command. These reactions can take the form of issuing commands to different subsystems. The CDHCmdDispatch construct, shown in fig. 4.3, is used to define the relationships between commands received by the CDH and commands/command sequences raised by the CDH, targeting other subsystems. These relationships are represented as a series of connections, of type DispatchConn, connecting a Trigger object to a Target object. A Trigger is a Reference to command, and models a command which the CDH can receive. A Target can either be another reference to a Command, modeling a command defined in a CommandSet of some other subsystem, or a Symbol, representing a parameter defined within a parameterized command.

Any command issued by the CDH is defined in its own CommandSet and is associated with another subsystem's command using the constructs defined within the CDHCmdDispatch construct. As opposed to supporting the issue of subsystem commands directly from the CDH, the approach taken was adopted in order to support a consistent separation of command definitions between subsystems. It is possible to embed this dispatch functionality into the CDH ControlFlow (sec. 4.1.4), but this separate modeling entity prevents the ControlFlow from getting more complicated.

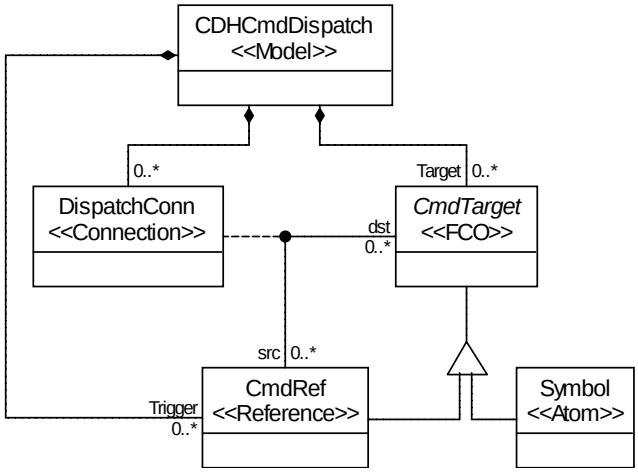

Fig. 4.3: Meta-model of CDH CDHCmdDispatch.

Example: An example CDHCmdDispatch object is shown in fig. 4.4. Boxes to the left represent references to commands defined in the CDH CommandSet, while boxes to the right are references to commands defined in the ADCS subsystem. The SetADCSStandby is defined as a CDH command and not an Uplink command like the SetADCSActive in order to allow the CDH to issue it whenever it needs to move to a safe mode as part of fault-handling. SetADCSActive and SetADCSStandby CDH commands map onto different Symbols within the ADCS ModeCmd, implying the mandate to change the ADCS ModeSystem to a specific mode corresponding to the CDH command.

4.1.3 DLData

The DLData construct is used to model a set of messages sent by the CDH to the ground station. Such messages are used to convey the current status of the spacecraft, to downlink data captured by the payload, or other mission-specific data. Each message is of type DLMessage. Similar to the specification of Commands, DLMessages can be parameterized using Symbols. All DLMessages are defined in the CDH. Since the Downlink subsystem is responsible for the actual transmission of the DLMessages, references to CDH DLMessages are defined within the Downlink subsystem. The CDH control flow structures capture the decision logic used to trigger the issue of a DLMessage to the Downlink subsystem, and its

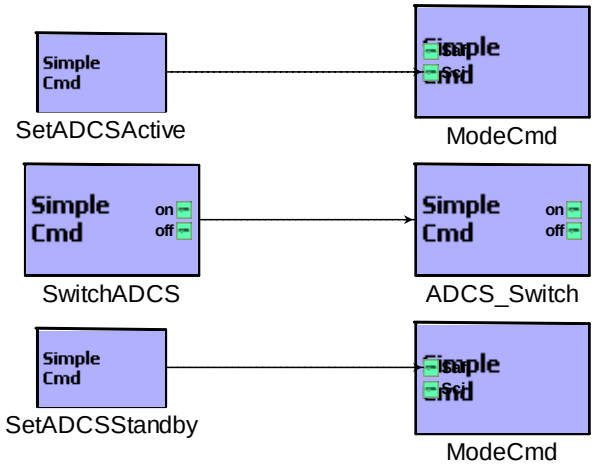

Fig. 4.4: An example CDHCmdDispatch.

corresponding transmission to the ground station.

4.1.4 ControlFlow

The CDH subsystem acts as the central nervous system for the spacecraft. BASS supports the specification of high-level control flow constructs which allow the specification of decision criteria such as the triggering logic for issuing commands to various subsystems and the triggering of recording and downlink of spacecraft data. The ControlFlow construct within the CDH allows the user to capture such specifications. Commands can be issued in response to a timer event, as part of error correction on the receipt of an exception, after sensing transitions in spacecraft data streams (e.g., due to a drop in power during eclipse, etc.). Some spacecraft missions require the specification of dynamic behavior in the CDH control logic, where multiple responses to a given situation are possible and decisions must be made at runtime, possibly involving the triggering of simultaneous responses. BASS offers a construct conceptually similar to StateCharts [26], though syntactically different, for modeling CDH control flow.

The use of state machines as a means of formally defining system behavior is not a new idea, and other projects have examined the coupling of formal semantics to finite state machines. Statemate statecharts were assigned CSP semantics by A.W.Roscoe and Z.Wu [27]. An approach for formalizing UML state diagrams into CSP has also been developed [15]. Initially we considered the adoption of such earlier published approaches formalizing state machines. However, with the goal of specializing the visual syntax to more closely represent spacecraft ideas and concepts, BASS introduces a variant of Statecharts, coupled with formal CSP semantics.

Figures 4.5 and 4.6 depict the BASS ControlFlow construct, consisting of ControlStates, ControlTransitions, AND_Connectors, and OR_Connectors. ControlConnections are used to represent both precedence and information flow. ControlStates model states in the state machine, wherein actions are performed. ControlTransitions are used to represent the criteria for transitioning between states. Such criteria include waiting for events to occur. ControlStates can be hierarchically refined, through the containment of other

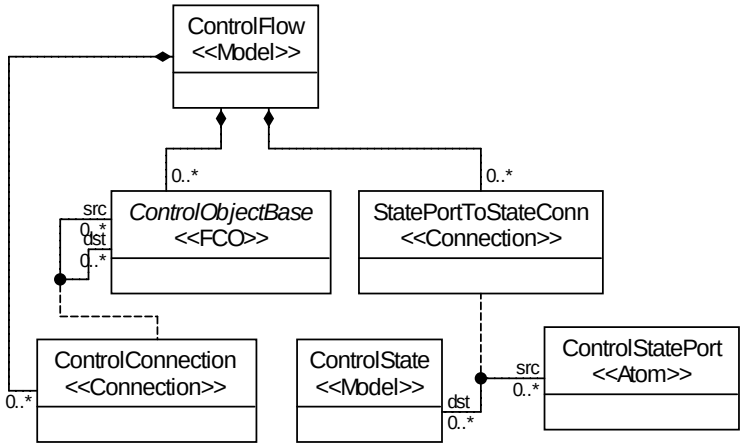

Fig. 4.5: Meta-model of ControlFlow.

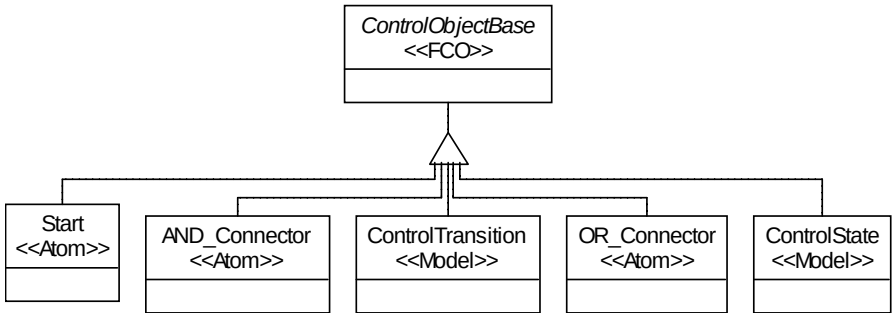

Fig. 4.6: Objects derived from ControlObjectBase.

ControlFlow objects. AND and OR connectors are used to specify concurrency and choice, respectively, within the ControlFlow.

Example: Figure 4.7 shows an example ControlFlow model. Control begins at Start, which immediately transitions through the AND_Connector to both the Attitude_Sun_Safe and powerDropMsgTrans ControlTransitions. Both transitions indicate criteria (not shown on the diagram) for waiting to transition to their respective connected states. Each transition can be taken independently.

ControlTransition:

ControlTransitions are defined similar to the transitions in a StateChart [26]. In a StateChart, the transition takes the form e[c]/a, where e is an event or combination of events which triggers the transition, c is a Boolean guard condition which must hold in order for

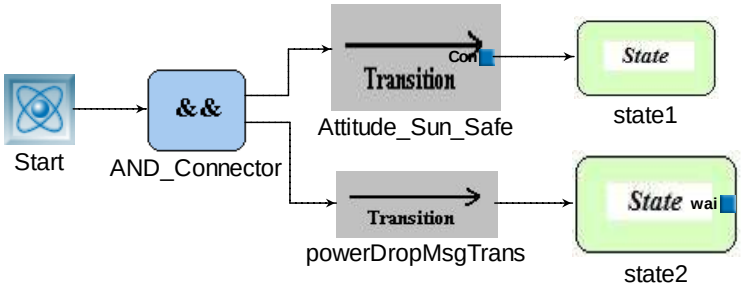

Fig. 4.7: Example ControlFlow with AND_Connector, ControlStates and ControlTransitions.

a transition to be taken, and a is an action which occurs once the transition is taken. In BASS, actions are modeled as part of a state, leaving only the trigger and guard conditions for each ControlTransition. The condition is specified via the ControlDecision construct, which behaves similar to an OR_Connector object. A ControlTransition can support up to two output connections. The first is followed when the transition is successfully triggered. The second is followed only when the transition criteria fails to be satisfied.

All the objects that can be used in a ControlTransition are derived from the ControlTransitionBase class, shown in fig. 4.8. The controlTransConn can be used to connect these objects in a specific order. The AssociatedSymbolConn is used to associate Symbols with ControlTransitionBase objects in order to model parametrization.

Figure 4.9 shows the objects that can be used as part of a ControlTransition. Each will be examined in more detail below.

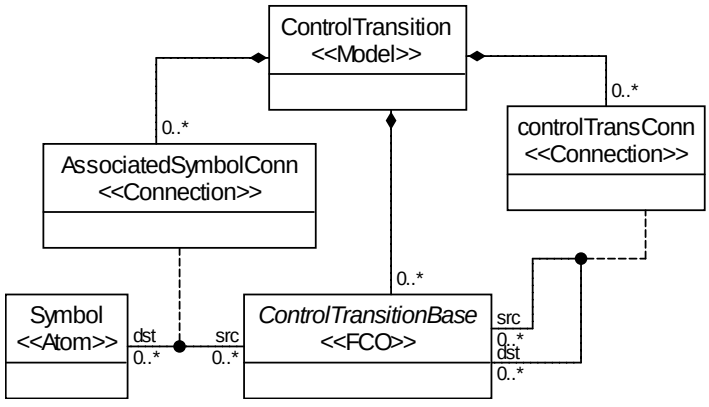

Fig. 4.8: Meta-model of ControlTransition.

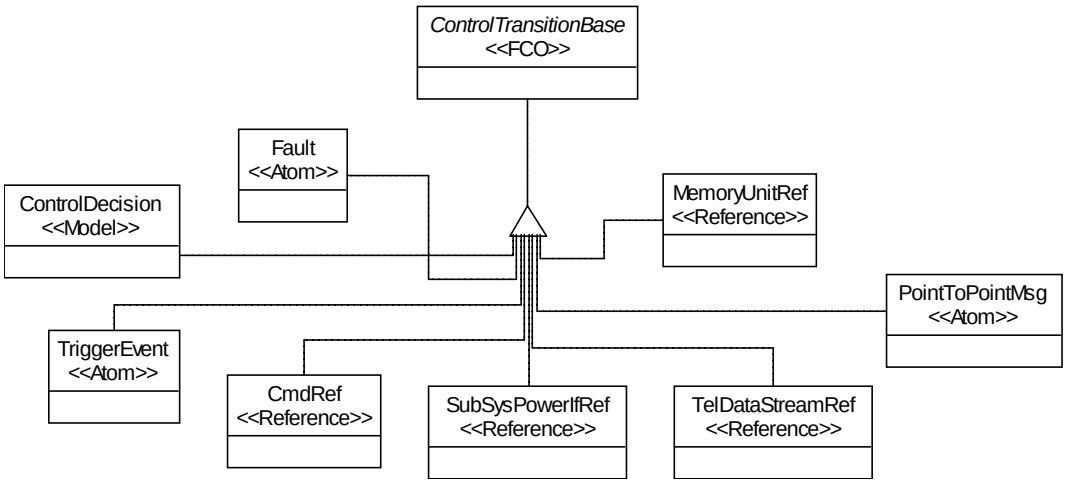

Fig. 4.9: Objects derived from ControlTransitionBase.

TriggerEvent: The TriggerEvent models the event that triggers the ControlTransition.

PointToPointMsg: The receipt of a PointToPointMsg from another subsystem can trigger a reaction by the CDH. For example, the CDH can wait for the receipt of a mode response message, sent as a PointToPointMsg object, from a subsystem which was commanded to change modes.

CmdRef: The CDH can wait for a command from the ground station which is relayed to the CDH by the Uplink subsystem. Such a scenario is modeled by instantiating a reference to the CDH command within a ControlTransition. The reference is of type CmdRef. If the command is parametrized, connections to one of the Symbol objects, acting as ports on the CmdRef, can be instantiated, facilitating the selection of a particular command from the parameter-based command family.

ControlDecision: The ControlDecision model acts as the guard condition for a ControlTransition and is connected in sequence with some other construct, which evaluates to a value. The value to which it evaluates is modeled as a Symbol. The ControlDecision construct defines two sets, SuccessSet and FailSet (fig. 4.10), each of which contain a collection of Symbol objects. The SuccessSet contains all symbols which, upon receipt by the ControlDecision, cause the decision to evaluate to true. Conversely, the FailSet contains a set of symbols, which if received, cause the decision to evaluate to false. Users need to

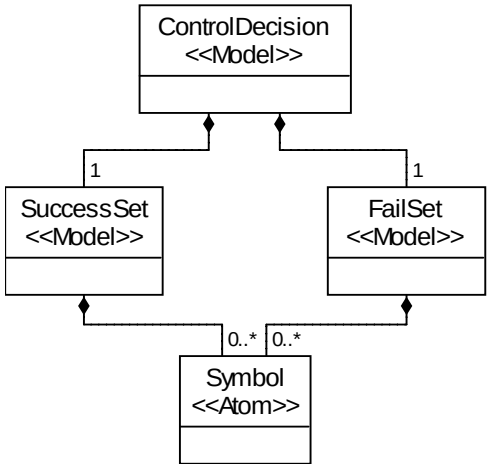

Fig. 4.10: Meta-model of ControlDecision.

define symbols for only one of these two sets, the other set is evaluated as a negation of the defined set. These two sets can be connected to a ControlStatePort, which can be connected to a ControlState, indicating the path of control flow after the decision is evaluated. ControlTransitions that do not contain a ControlDecision are connected to a ControlState using the ControlConnections of fig. 4.5.

A single connection from the SuccessSet represents a blocking transition for the desired value while both the connections are used to model scenarios where failure to find the desired value is possible. The two constructs that can precede the ControlDecision are TelDataStreamRef, modeling a reference to a Telemetry Data Stream object, and MemoryUnitRef, modeling a reference to a Memory_Unit object. A connection from a ControlDecision's SuccessSet, as shown in left figure of fig. 4.11, is required, and indicates where control proceeds once a symbol in the SuccessSet is received. When a ControlDecision has no output connection from its FailSet, upon evaluation of a ControlDecision, the CDH must wait until such a symbol is received by the ControlDecision. Alternatively, if a connections from both the FailSet and SuccessSet are defined as in right figure of fig. 4.11, control proceeds once a symbol in either set is received, continuing along the output path corresponding to the received symbol's set.

TelDataStreamRef and ControlDecision: A change in a Data Stream can cause the

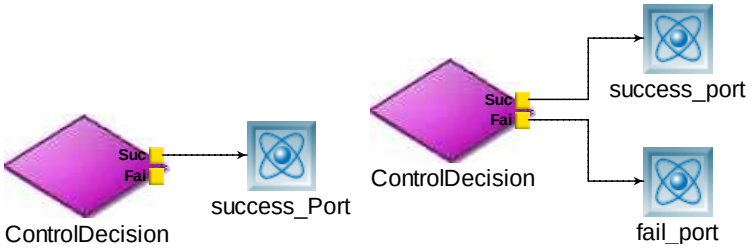

Fig. 4.11: Possible connections out of a ControlDecision.

CDH to transition to a new state. The dependence of a ControlDecision on the value of a Data Stream is captured by including a reference to a Data Stream in a ControlTransition, and associating it with a ControlDecision object. This association indicates that the evaluation of the ControlDecision begins with the examination of the Data Stream's current value, which is compared against the Symbols in the ControlDecision object's SuccessSet and FailSet. In the case where the FailSet is defined and connected to a port, the ControlDecision evaluates immediately, taking one of the two available paths (success or failure). In the case where the FailSet is not connected to a port, the ControlDecision waits until the corresponding data stream takes on a value contained in the ControlDecision's SuccessSet.

An identical combination of a query on the Data Stream followed by a ControlDecision is possible within the ControlTransition. The first case would be then sampling the Data Stream continuously till the desired value is observed. The other case would be a query followed by two possible lines of behavior based on whether the queried value belongs to the SuccessSet or not.

Example: Figure 4.12 shows the internals of a ControlTransition in which the CDH waits for the Attitude_DataStream to take on the value of Sun_Safe_Attitude. The Sun_Safe_Attitude is contained in the ControlDecision's SuccessSet.

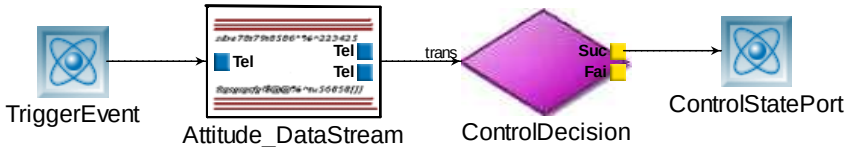

Fig. 4.12: Example ControlTransition containing a TelDataStreamRef and ControlDecision.

MemoryUnitRef and ControlDecision: Some control transitions can depend on values or changes to values stored in MemoryUnits. The syntax for capturing such dependences is identical to that used wtih Data Streams.

SubSysPowerIfRef: Control flow in the CDH can also depend on whether power has been provided to a particular subsystem. Consequently, a ControlTransition can contain a reference to a SubSysPowerIf object, along with an associated "on" or "off" symbol, indicating the switching on or off, respectively, of the corresponding subystem. The switching of all powered subsystems except the CDH are controlled via commands sent by the CDH to the Power Subsystem, therefore only the CDH's SubSysPowerIf can be used within the ControlFlow.

Example: Figure 4.13 shows the SubSysPowerIf of the CDH subsystem in a ControlTransition implying that this transition is triggered when power is initially provided to the CDH.

ControlState:

A ControlState models a sequence of actions performed by the CDH. The system remains in a particular ControlState until the transition criteria for leaving the state are met. ControlStates can be defined hierarchically, through the containment of ControlFlow objects. The ControlState meta-model is presented in figs. 4.14 and 4.15. All the objects that can be used in a ControlState are derived from the ControlStateBase class. The ControlStateConns connections can be used to connect these objects, representing the sequence of actions to be carried out in the ControlState. The AssociatedSymbolConn is used to associate Symbols with ControlStateBase objects, modeling parametrization. Each class inheriting from

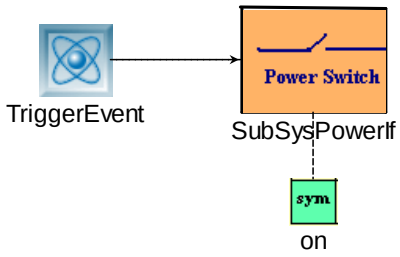

Fig. 4.13: Example ControlTransition containing a SubSysPowerIf.

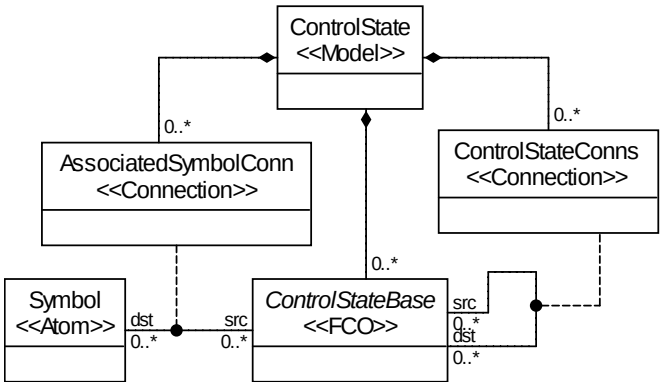

Fig. 4.14: Meta-model of ControlState.

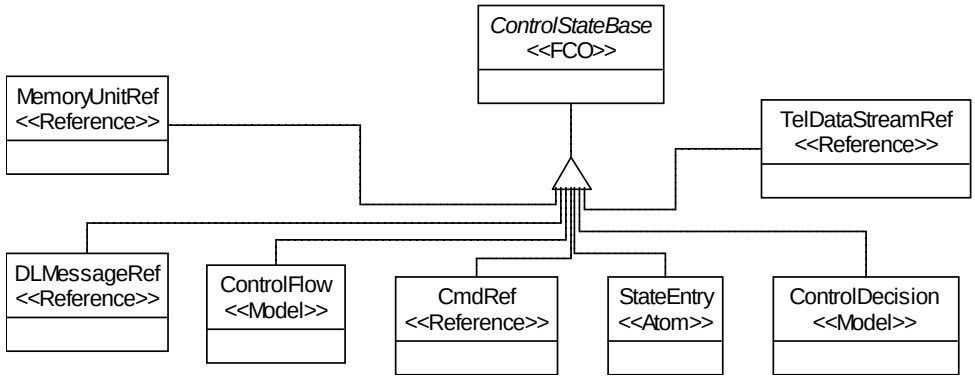

Fig. 4.15: Objects derived from ControlStateBase.

ControlStateBase is discussed below.

StateEntry: The StateEntry object represents the beginning point of the sequence.

CmdRef: The inclusion of a CmdRef object inside a ControlState models the issuing of the corresponding command.

DLMessageRef: The CDH can issue messages to the Downlink subsystem in order to transmit them to the ground station. References to DLMessages defined in a ControlState model the issue of such messages.

TelDataStreamRef and ControlDecision: A query on the Data Stream can be done in a ControlState and based on the current value of the stream, control proceeds in two different ways. We do allow modeling of any data stream transitions because they are blocking and we donot wait in a ControlState. The semantics generated for this combination of modeling constructs is similar to that in a ControlTransition.

MemoryUnitRef and ControlDecision: Similar to a query of the Data Stream, query of some stored values could decide the flow of control within CDH and can be included within a ControlState.

ControlStatePort:

ControlStatePorts are used to model the transfer of control from one state to another state, where the second state is not necessarily located adjacent to the source state in the model. This facilitates a scalable environment for modeling large, hierarchical control flows.

Example: In fig. 4.16, we can see flow of control from the state Is_Science_Attitude to the ControlStatePort Wait_PowerDropMsg. This port, visible on the ControlFlow, shown in fig. 4.16, can be used to connect to another ControlstatePort. Control can be propogated recursively in this fashion upto the level containing the desired ControlState is reached.

OR, AND Connectors:

BASS deviates from traditional Statechart-like syntax when representing mutual exclusion and concurrency. Instead of the Statechart-like AND-state and OR-state approach, BASS leverages constructs from the domain of Functional Flow Block Diagrams (FFBDs), in the form of OR_Connectors and AND_Connectors

OR_Connector: The OR_Connector is used to model alternative execution paths. The OR_Connector connects to two or more ControlTransition objects, each of which models transition criteria. When control arrives at the OR_Connector, the first output ControlTransition whose trigger and guard criteria are satisfied is taken. The representation of non-determinism is also supported, where the OR_Connector is connected to a set of Con-

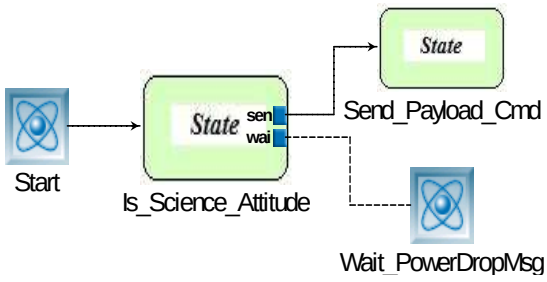

Fig. 4.16: Example ControlFlow involving ControlStatePorts.

trolStates instead of ControlTransitions. When the model is evaluated, the model checker will evaluate all possible paths of execution, in order to determine if the system behaves as specified.

AND_Connector: The AND_Connector is used to partition control flow into mutiple paths, each of which occur simultaneously. An AND_Connector can connect to either a set of ControlStates or a set of ControlTransitions.

4.2 ADCS

The Attitude Determination and Control Subsystem is responsible for ascertaining and maintaining the orientation of the spacecraft. The set of all attitudes the spacecraft is expected to achieve is determined at design time. Attitude sensors, actuators, and control hardware and software are all required for determining and acquiring particular attitudes. While significant design effort is spent on analyzing and implementing the hardware and control algorthims for the ADCS, the complexity of attitude control does not impact other spacecraft subsystems. Other subsystems simply depend on the ADCS achieving appropriate attitudes as required by the mission. The attitude of a spacecraft corresponds to physical state. That state is modeled in BASS with a SharedState object named Attitude. Those subsystems whose behavior is dependent directly on the phyiscal attitude of the spacecraft can access the current attitude via the SharedState object. In BASS, Spacecraft attitudes are defined as a set of discrete states, such as Earth Pointing or Sun Pointing.

Rest of the ADCS modeling in BASS will utilize the power and control interfaces discussed in Chapter 3. Mode based behavior commonly found in ADCS design can be described using ModeSystem along with the power based variations it causes.

4.3 Payload

The Payload subsystem typically consists of one or more mission-specific instruments which collect data of interest to the spacecraft stakeholders. The Payload subsystem is responsible for managing the corresponding instrumentation, and for properly invoking it, collecting data, and forwarding that data to the CDH for buffering and downlink. The

behavior of each payload instrument can be modeled by its own ModeSystem. Commands and PointToPointMsgs are defined as part of the Payload so as to allow the CDH to control the Payload's operation.

4.4 Power

The Power subsystem manages the amount of power generated, stored, and utilized on-board the spacecraft. Estimating the power requirements and sizing the sources of power (fuel, batteries, or solar panels) accordingly, in order to meet power requirements throughout the mission duration is given major emphasis during spacecraft design. However, for modeling system-level interactions, the low-level details of power generation and conversion are abstracted, in favor of verifying that amount of power determined to be available meets the power required by normal spacecraft operations in all possible modes. The Power subsystem is generally designed to satisfy the average and peak electrical load requirements through the end of mission. It is not uncommon for small satellites to have a solar-array powered battery which may not meet the peak power requirements at all the times. Whether such a condition is acceptable or not depends on the spacecraft mission and its design. Irrespective of peak power requirement analysis, a modeling tool such as BASS should support the description of load-shedding behavior if any, when peak power requirement is not met.

Another important modeling requirement for the Power subsystem is the propagation of changes in power consumption that are a direct result of switching on/off spacecraft subsystems and variations in the modes of subsystem behavior. The amount of power generated from solar panels is affected by the attitude of the spacecraft, hence there is an implicit interface between the attitude and power generated at any instant. Modeling this dependence in terms of numbers that represent the power generated in each attitude is important. BASS takes the general approach of supporting the modeling of power-related changes in each subsystem behavior, and determines whether the power available is greater than the power consumed over all possible combinations of behavior.

Meta-Model: The PowerPorts shown in fig. 4.17 model the power interfaces to a powered

subsytems. The Power Subsystem is responsible for switching on or off each PowerPort. The switching of each powered subsystem's PowerPort is controlled by corresponding commands defined in the commandSet of the Power subsystem. This one-to-one correspondence is represented using the PowerSwitchConnections that connect a SimpleCommand to a PowerPort. Initially, when the Power subsystem comes up, all the subsystems are switched on in a sequence indicated by the PowerInitSequence, shown in fig. 4.18. This is a command sequence, containing all the commands that control the PowerPorts. Any further controlling of the PowerPortscan be done only through commands issued by the CDH. Any changes in power consumption by a subsystem are communicated to the Power subsystem through PowerPorts. As mentioned earlier, attitude changes of the spacecraft affects power generation. The relationship between power generation and current attitude is specified using the AttitudeSpecificAvailablePower MapFunction.

Figure 4.18 indicates the presence of a SharedState object in the role PowerState, which holds the current power level available on the spacecraft. "Power level" is discretized into different qualitative levels, each of which is represented with a Symbol. Each such level is associated with some percentage Depth of Discharge (DOD) of the battery as part of the design. The association of relative numerical values to the qualitative Power Level values can be specified using the PowerStateToNumMap MapFunction. For spacecraft which do not have qualitative levels of power, a single power level can be used within the PowerState.

The next step would be modeling behavior that causes variations in the power level

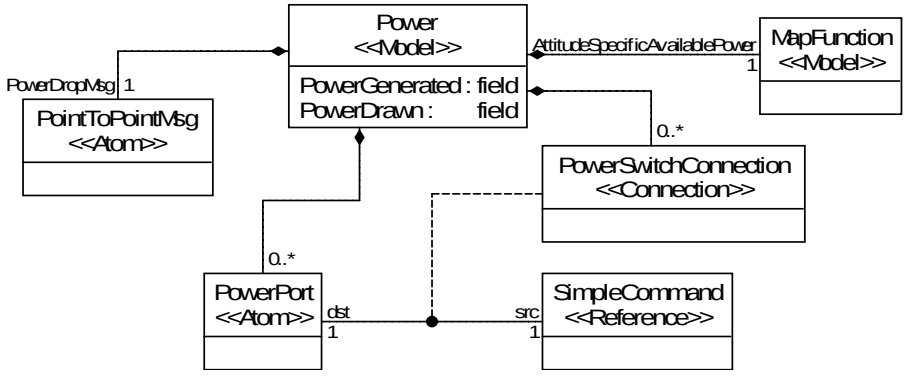

Fig. 4.17: A subsection of the Power subsystem meta-model.

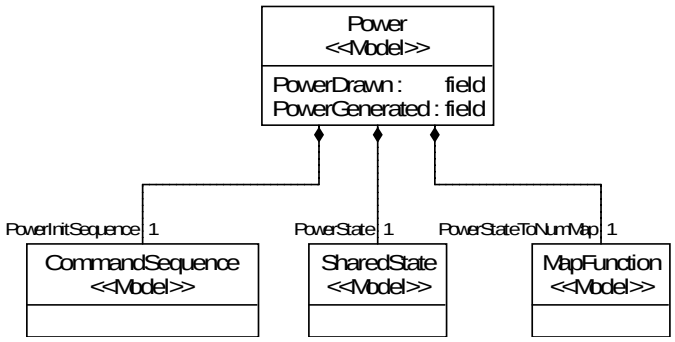

Fig. 4.18: Rest of the modeling constructs in Power subsystem.

for a solar-array-based battery (as this is almost always the source of power in recent small satellites). This involves modeling the build up of charge in the battery which depends not only on the charge rate and duration but also on the attitude which controls the solar flux incident on the solar panels. These time-durations are non-deterministic, depending either on the sequence of modes the spacecraft goes through or the variable eclipse durations during each orbit. Consequently, we model the PowerState as an internally controlled object whose implementation details are covered in Chapter 6.

A steep fall in power level generally triggers behavioral changes in several of the subsystems. To enable the user to specify such behavior, BASSMP defines a type of a PointToPointMsg named PowerDropMsg that is generated whenever the amount of charge in the battery decreases to the next threshold value. This message can be received by the CDH and subsequenly handled as per the design indicated in its ControlFlow. Typical operations on receiving the PowerDropMsg include shutting down the science operations and modifying the attitude.

Example: Figure 4.19 shows two power ports, one for the CDH and the other for the ADCS subsystems. The triggering commands used to control these ports are defined in the Power subsystem's CommandSet. These triggering commands are also utilized in the PowerInitSequence to model the system startup sequence.

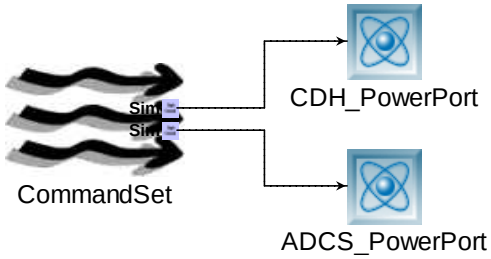

Fig. 4.19: Example showing the association between PowerPorts and SimpleCommands.

4.5 Uplink

The Uplink subsystem does not require any additional constructs for modeling its behavior. Its main function is decoding the commands received from the ground station through the antennas. All the decoded commands are then relayed to the CDH for further action. Since this subystem does not affect the functioning of other subsystems aside from CDH, it is relatively simple to model. The Uplink subsystem supplies inputs to the spacecraft as commands, which are specified within its CommandSet. The Uplink CommandSet actually contains references to commands defined in the CDH CommandSet. The presence of a CDH command within Uplink implies that the particular CDH command is issued by the ground station.

4.6 Downlink

The Downlink subsystem is responsible for encoding and relaying data of interest to the ground station. The structure of the qualitative data that needs to be sent and its sequencing is controlled by the CDH. References to DLData constructs defined in the CDH are used in this subsystem to indicate the data that it needs to encode and transmit.

Chapter 5

Spacecraft System Verification

This chapter considers the composition of the constructs discussed in Chapter 3 and Chapter 4 to create a complete specification of various subsystems in a spacecraft. Additionally, we examine the composition of subsystem models to form a composite spacecraft system model, reflective of block diagrams used in traditional spacecraft design documents. System level models in BASS can also contain specifications of requirements or checks to be carried out against spacecraft models. The BASS approach for modeling such assertions or queries is discussed in this chapter as well.

When building the subsystem-level and system-level meta-models in GME, it is useful to utilize aspects. Aspects can be used to create multiple views of a spacecraft model. Since spacecraft designs are generally complex, the separation of the design into distinct views aids in design comprehension through separation of concerns. BASSMP supports three aspects DataCommAspect, PowerAspect, and ExceptionsAspect, each of which are discussed below. All components of the spacecraft system and its constituent subsystems are assigned to one or more aspects.

5.1 Aspects in a Subsystem

5.1.1 DataCommAspect of a Subsystem

The DataCommAspect shows constructs which model Command and Data Handling, whether they form part of a subsystem which interacts with the CDH or the CDH itself. Such constructs include, as shown in fig. 5.1, CommandSet, ModeSystem, PointToPointMsgs, SharedStates, and TelDataStreamRefs, which are used in a variety of subsystems. In addition to these constructs common to many subsystems, CDHCmdDispatch,

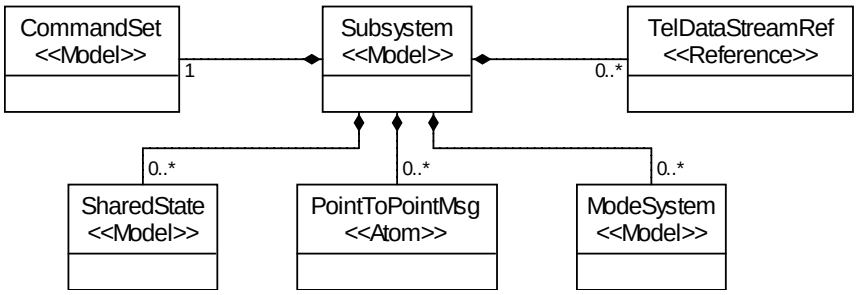

Fig. 5.1: Subsystem meta-model involving the constructs mapped onto DataCommAspect.

ControlFlow, and DLData in the CDH subsystem, the Attitude state of the ADCS subsystem, PowerDropMsg of the Power subsystem, and DLMsgRefs of the Downlink subsystem, are also mapped onto this aspect.

Meta-Model: In the DataCommAspect, the CommandSet and TelDataStreamRefs contain or define constructs which involve the transfer of data to and from the system bus; hence they are declared as ports of their containing subsystem, allowing them to be visualized on the edges of the subsystem when viewed from the perspective of the subsystem's parent Their inclusion as ports facilitates the specification of inter-subsystem connectivity, as will be discussed in more detail below.

5.1.2 PowerAspect of a Subsystem

Unlike control and data handling activities, which are performed by all the subsystems, power-related activities are executed only by subsystems that draw power. Consequently, only those subsystems which are subclasses of PoweredSubsystems, defined in sec. 5.2, offer a power aspect. Each powered subsystem, as shown in fig. 5.2, contains a power interface of type SubSysPowerIf, a function that specifies the power drawn for each of the subsystem's modes, and the subsystem's CommandSet, each of which is visualized in the subsystem's PowerAspect view. Almost all constructs defined in the Power subsystem are assigned to the PowerAspect view, including AttitudespecificAvailablePower, PowerPorts, PowerSwitchConnections which connect commands to PowerPorts, and the PowerState object which signifies the qualitative threshold charge levels in a battery.

Meta-Model: Both the power interfaces, PowerPorts and SubSysPowerIfs, involve physi-

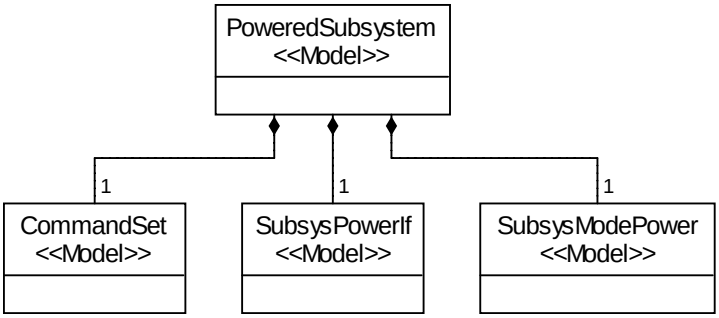

Fig. 5.2: Subsystem meta-model involving the constructs mapped onto PowerAspect.

cal connections between subsystems and therefore are declared as ports of the subsystem, visualized in the system-level PowerAspect.

5.1.3 ExceptionsAspect of a Subsystem

The ExceptionsAspect view shows the faults that a subsystem can generate, manifested as subsystem-level exceptions. The communication of exceptions to the CDH occurs through the exception-handling port.

Meta-Model: Figure 5.3 shows the Subsystem-level containment of constructs defined for managing exceptions. The Exceptions model is defined as a port on the subsystem so as to permit connections to the SystemBus.

5.2 Composite Spacecraft Model

The top-level SpacecraftSystem model, which defines how subsystems compose to form a spacecraft system, is shown in fig. 5.4. A SpacecraftSystem is composed of Subsystems. Different types of subsystems are defined in BASS. The classification is based on power

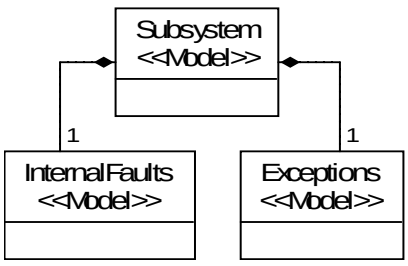

Fig. 5.3: Subsystem meta-model involving the constructs mapped onto ExceptionsAspect.

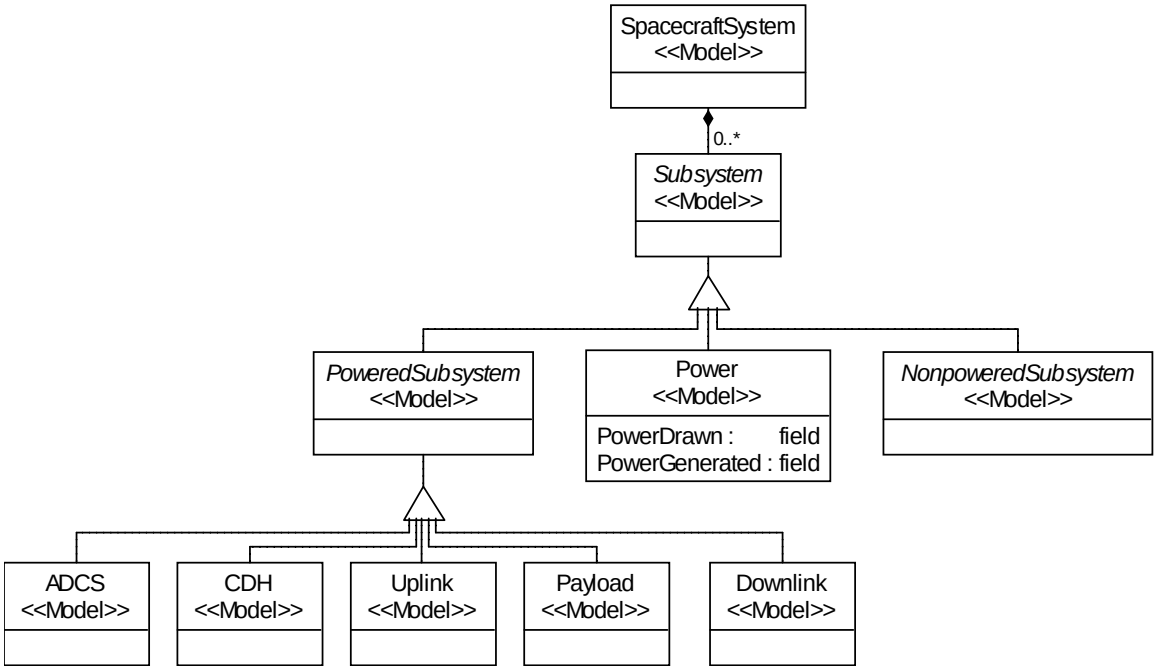

Fig. 5.4: Hierarchical model of a spacecraft in BASSMP.

consumption: PoweredSubsystems consume power, while NonpoweredSubsystems do not. Additionally, the Power subsystem provides all required power to the spacecraft. All subsystems currently supported in BASS fall into the PoweredSubsystems category. Subsystems like Structure, if not consuming power, would fall under the NonpoweredSubsystem category.

5.3 Aspects in SpacecraftSystem

Subsystem intercommunication and interfacing is modeled using connections between subsystem ports and a system bus. Such connections are partitioned into the three aspects supported by BASS.

5.3.1 SpacecraftSystem DataCommAspect

Connections are used in the DataCommAspect to indicate the flow of data and commands via the system bus. SystemBus, defined in fig. 5.5, is the physical carrier of data in the spacecraft. All TelDataStreamRefs must connect to the SystemBus with a connection

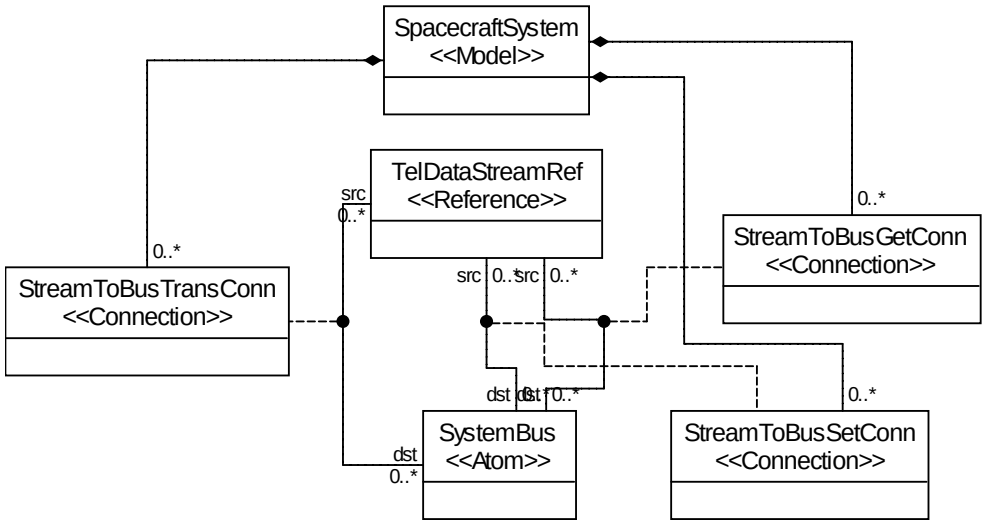

Fig. 5.5: Meta-model showing telemetry streams to system bus connectivity.

type that indicates the interface employed set, get, or trans. The three connection types supported are StreamToBusGetConn, StreamToBusSetConn, and StreamToBusTransConn.

In addition to these, connections indicating command transfer from CDHCmdDispatch to rest of the subsystems CommandSets can be indicated using appropriate connections to and from the SystemBus.

Example: Figure 5.6 shows DataCommAspect of an example spacecraft system. The ports labeled "Att" and "Pow" present in ADCS, Payload, Power, and CDH represent ports corresponding to TelDataStreamRef objects. Connections between ports labeled "Com" correspond to the transfer of commands between CommandSets.

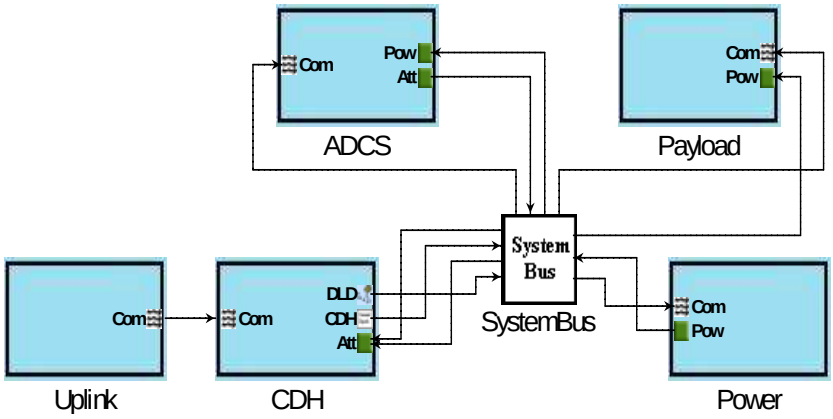

Fig. 5.6: DataCommAspect of an example spacecraft.

5.3.2 SpacecraftSystem PowerAspect

The PowerAspect view at the system level addresses power considerations. As discussed previously, the power ports to each subsystem and the mode-based power function are part of the PowerAspect for each PoweredSubsystem. Within the Power subsystem, PowerState, modeling the qualitative power levels of the spacecraft, PowerInitSeq, PowerstateToNumMap, and AttitudeSpecificAvailablePower are mapped onto the PowerAspect. In the system-level view, connections are defined which model the conveyence of power between the Power subsystem and the other powered subystems. As shown in fig. 5.7, the PowerPorts present within the Power subsystem and the SubSysPowerIf of a PoweredSubsystem can be connected in the PowerAspect using the PowerConnections.

Example: Figure 5.8 shows the PowerConnections between the Power and the rest of the subsystems in the PowerAspect. The ports labeled as "Sub"are the SubSysPowerIf objects defined within each of the powered subsystems.

5.3.3 SpacecraftSystem ExceptionsAspect

The system-level connections used in the ExceptionsAspect view indicate the path that

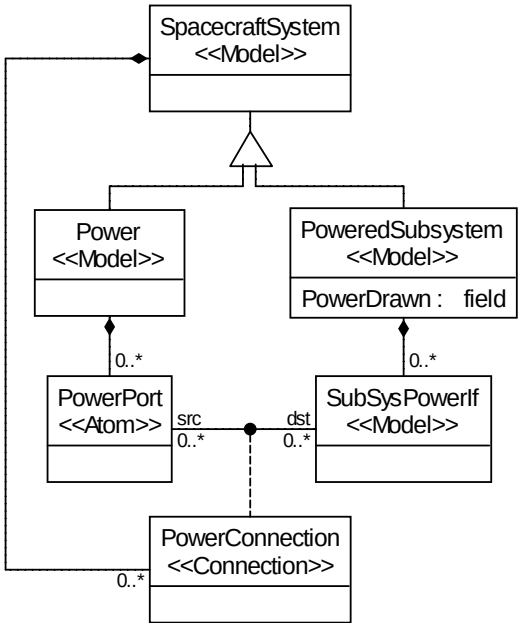

Fig. 5.7: Subsystem meta-model involving the constructs mapped onto PowerAspect.

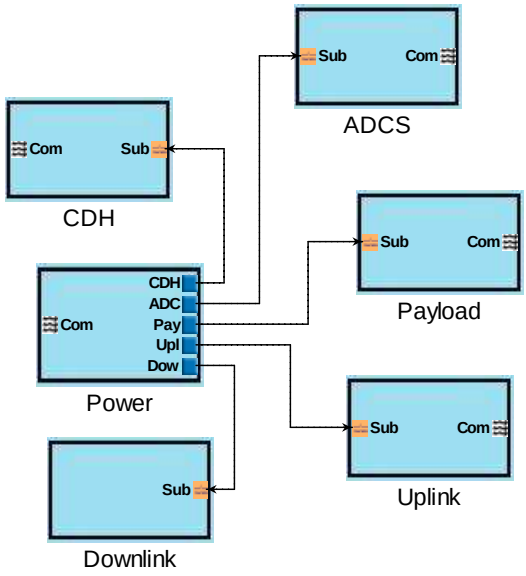

Fig. 5.8: PowerAspect of an example spacecraft.

generated exceptions take to the CDH. Specifically, connections are defined which emanate from a subsystem's Exceptions port and terminate on the SystemBus, and emanate from the SystemBus and terminate on the CDH's ExceptionHandlingPort .

Example: Figure 5.9 shows an ADCS exception being propagated to the CDH via the SystemBus.

5.4 Spacecraft System Verification

BASS supports the capture of requirements for a spacecraft system. A requirement in this context is a desired property of the system behavior, and is abstracted as a sequence of observable events in the execution of the spacecraft behavior model. Such observable events

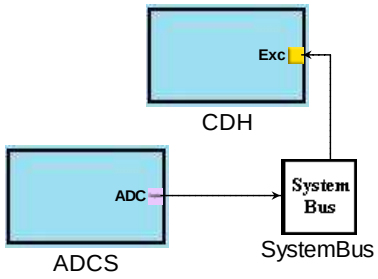

Fig. 5.9: ExceptionsAspect of an example spacecraft.

in BASS consist of system inputs and outputs. Inputs consist of commands issued from the ground station, captured as commands contained in the Uplink's CommandSet. Outputs consist of messages that are downlinked to the ground station modeled as DLMsgRefs in the Downlink subsystem, signifying qualitative information about spacecraft behavior. In addition to inputs and outputs, subsystem faults can be modeled as system inputs, allowing users to model the insertion of a fault during system operation.

An assertion generated from the modeling of requirements and fault injections, is applied during verification, comparing all execution paths of the modeled system to ascertain consistency with the the assertion. Variations in PowerState discussed in sec. 4.4 are non-deterministic in the BASS behavior model, modeling changes in spacecraft attitude, leading to a reduction in current production from the solar panels or situations where the spacecraft is in an eclipse. The non-determinism allows the checking of power levels without requiring the inclusion of a model of orbital mechanics. However, some spacecraft requirements may specify desired behavior when adequate solar flux is not available in a certain attitude or mode. For such requirements, variations in solar flux must be made explicit through their inclusion in the requirement definition in BASS. Summarizing, the requirement specification should be able to drive the power level available to the system, if required.

Meta-model: Assertions on spacecraft behaviors are captured as ModelChecks, as shown in fig. 5.10. BASS defines ModelCheckLib, which defines a folder that can contain a collection of ModelChecks. Each ModelCheck has CheckBaseClass objects which are connected in a sequence using the CheckBaseClassSequencer connections. Some of the CheckBaseClass objects can be associated with Symbols using AssociatedSymbol connections in order to model parameterization.

Several types of objects can be included in a ModelCheck, as shown in fig. 5.11. CmdRefs indicate particular commands to be sent to the spacecraft from the ground station and DLMessageRefs model messages communicated from the spacecraft to the ground station. In addition, "Faults" model faults injected into the system and SolarFlux objects can be used to specify the availability of SolarFlux to the spacecraft to support explicit changes

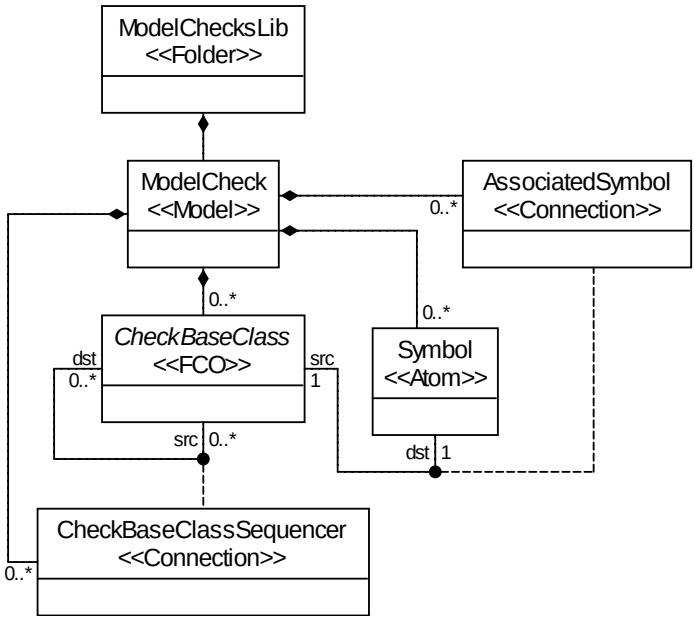

Fig. 5.10: Meta-model of ModelCheckLib.

to the PowerState. StartCheck is a part which indicates the beginning of a ModelCheck sequence.

Example: Figure 5.12 presents an example ModelCheck which translates a spacecraft requirement into BASS. To separate the ground station and spacecraft domains, we model the ground station-issued commands to the left and the spacecraft generated information messages onto the right. The sequence begins with the StartCheck atom at the top of

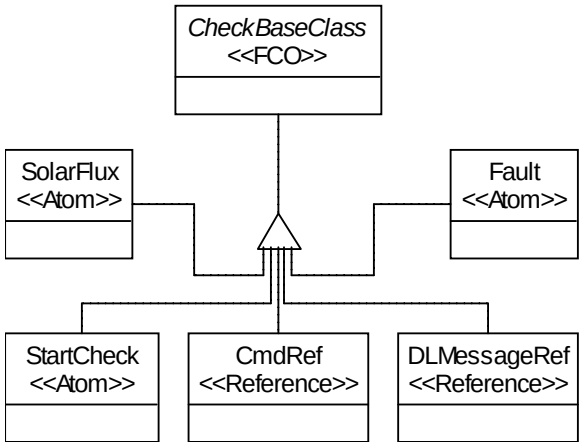

Fig. 5.11: Objects derived from CheckBaseClass.

the right column. The check first waits for the arrival of a downlink message, of type ADCSModeStatusMsg, of which the Safehold parameter is selected. This indicates that the check makes sure that the spacecraft starts up with theADCS in the Safehold mode. Once that message is received by the ground station, it issues the SetADCSActive command, and then waits for the spacecraft to react. The check asserts that the spacecraft should next send an ADCSModeStatusMsg, indicating that the ADCS has entered the Sci_Active mode. If, during verification, this sequence is not adhered to by the spacecraft behavior, an error is raised and the user is notified.

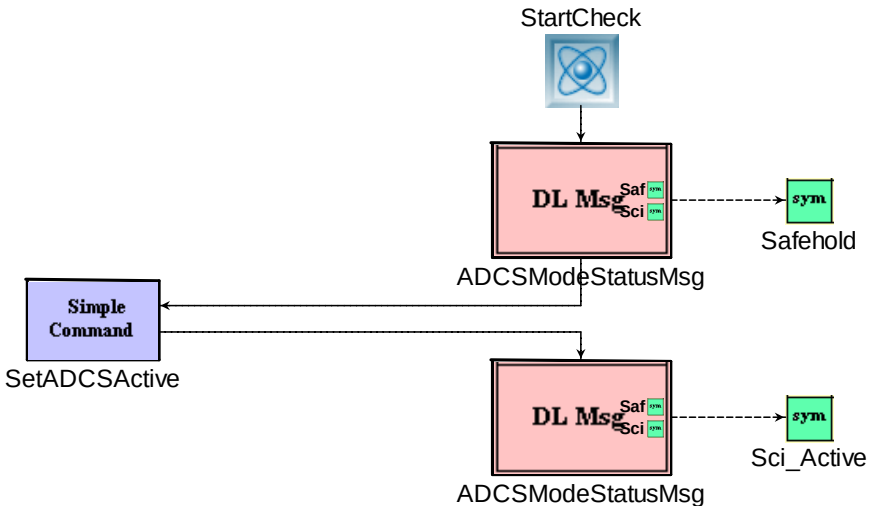

Fig. 5.12: An example spacecraft requirement composed as a ModelCheck.

Chapter 6

BASS Interpreter and CSP Semantics

Chapters 3, 4, and 5 focus on the description of the visual language for modeling spacecraft system behavior. Without an interpreter that translates the graphical models into appropriate semantics, BASS is reduced to a tool for creating visual documentation. The interpreter framework provided with GME enables the mapping of visual models to hierarchically traversable data models that can be utilized in creating artifacts of interest. This chapter describes the development of BASSI, the BASS Interpreter tool, which extracts information captured in the visual diagrams and generates an equivalent CSP model of the spacecraft. The interpreter leverages a library of CSP constructs previously developed by McInnes [8]. The generated CSP code can be accepted by a commercially-available model checking tool in order to verify consistency and correctness of the specification, and therefore the correctness of the user-specified visual models.

The BASS Interpreter (BASSI) traverses models and derives appropriate semantics from them. The semantics currently supported by the BASS Interpreter is captured as machine-readable CSP. All the modeling constructs mentioned earlier have their corresponding CSP process models, most of which are leveraged from the ideas and constructs introduced by McInnes [8]. BASSI makes liberal use of his SBFL (Spacecraft Behavior Framework Library). In this chapter, the semantics for the modeling constructs introduced or modified as part of this thesis are presented. Those constructs which are equivalent to structures and concepts defined as part of the SBFL are omitted. The interested reader can find the CSP model for a complete system in Appendix A. The focus in this section is on demonstrating the correlation between the modeling constructs and their equivalent CSP, as opposed to the algorithm used by the BASS Interpreter to implement the mapping from models to CSP. The mapping algorithm primarily consists of traversals of the model

objects in order to access relevant information. The traversals themselves are not viewed as a fundamental contribution of the thesis.

BASSI follows the hierarchical composition of the spacecraft when generating the process models. The behavioral components of each subsystem, are composed using an appropriate CSP parallel composition to define a system-level CSP process. In a similar fashion, subsystem processes are composed using the alphabetized parallel combination. In order to be able to use the alphabetized parallel combination, each process involved should have an interface associated with it. The interface set associated with each subsystem process includes a set of all the events that are of interest to the rest of the subsystems, or which may be referenced in the system requirements specifictaions.

The discussion of the CSP binding follows a structure similar to that applied to the discussion of the metamodel. We first discuss subsystem constructs that are employed in a variety of subsystems, followed by a discussion of subsystem-specific constructs. Again, only constructs that were added to the set that McInnes developed are presented here.

6.1 CSP for Generic Subsystem Modeling Constructs

6.1.1 PointToPointMsgs

PointToPointMsgs are transmitted as an event over a channel. Each such message is associated with a telemetry data stream for the message's source subsystem. For instance the PowerDropMsg of the Power subsystem is mapped onto a CSP event *SystemBus.PowerTlm.PowerDropMsg*. In this construct, *SystemBus.PowerTlm* represents a channel and *PowerDropMsg* is an event that is transmitted via that channel.

6.1.2 MemoryUnitsLib

The context in which Memory_Units and SharedStates are used is different, but their CSP semantics are identical, as both are data storage elements. However, there is one check performed by BASSI when creating a CSP process for a Memory_Unit: if the *trans* channel of a Memory_Unit is never used, it is not added to the interface set for the process containing

it. If the *trans* channel is not omitted in such circumstances, following an update to the value stored by the Memory_Unit, the process will wait indefinitely trying to synchronize on a non-existent *trans* event. All the Memory_Units in the system model are defined in the special library referred to as MemoryUnitsLib. We present an example MemoryUnitsLib and present step by step CSP translations BASSI performs.

Figure 6.1 shows all possible values the Memory_Units ADCSModeVal (which stores current mode of ADCS) and PayloadModeVal (which stores current mode of Payload) can take, to the left and right sides, respectively. The Symbols contained in AllowedValues are aggregated into CSP "datatypes" *ADCSModeValDType* and *PayloadModeValDtype* of fig. 6.2. Another datatype called *StateIF* is used from the SBFL which defines the specific action carried over a Memory_Unit: *set*, *get*, or *trans*. Combination of these two datatypes *StateIF* and *ADCSModeValDType*/*PayloadModeValDType* results in the creation of compound datatypes. The Memory_Units are converted into CSP channels *ADCSModeVal* and *PayloadModeVal* characterized by their corresponding compound datatypes.

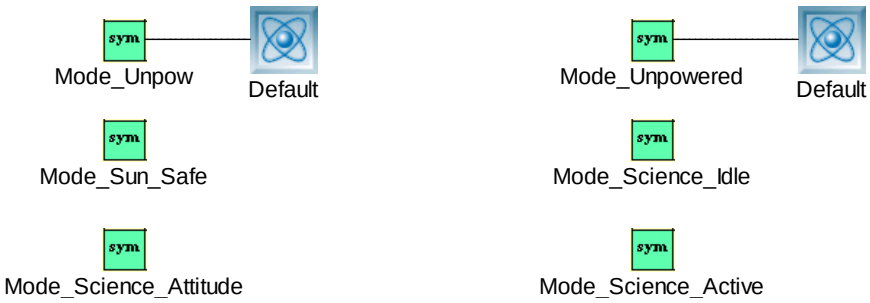

Fig. 6.1: Contents of AllowedValues of Memory_Units ADCSModeVal and PayloadModeVal.

```
datatype ADCSModeValDType = Mode_Unpow | Mode_Sun_Safe | Mode_Science_Attitude
datatype PayloadModeValDType = Mode_Unpowered | Mode_Science_Idle |
                               Mode_Science_Active

channel ADCSModeVal: StateIF.ADCSModeValDType
channel PayloadModeVal: StateIF.PayloadModeValDType
```

Fig. 6.2: CSP channels and datatypes generated for ADCSModeVal and PayloadModeVal.

The state-bearing portion of the Memory_Units are defined as aliases to the *AssignableState* process, which is defined in SBFL as a construct for storing a value. Each of the two processes (fig. 6.3) are parameterized by their respective *set*, *get*, and *trans* channels, and they are each supplied with an initial value as the parameter to the process during its invokation.

As Memory_Units are used between subsystems, and not owned by a particular subsystem, all of them are composed into a single process *MemoryUnits*1, shown in fig. 6.4. Since each Memory_Unit defined in MemoryUnitsLib is an independent entity, the interleaving operator "|||" is used to compose them.

In order to avoid synchronization issues when more than one subsystem reads the Memory_Units's value, all the *getval* channels in *MemoryUnits*1 process are renamed to subsystem-specific channels using CSP's renaming operator "⟦ ⟧," resulting in the *MemoryUnits* process. *Memory_Units* is the system-level process that encapsulates behavior of all the Memory_Units used within the system and *MemoryUnitsIF* is its corresponding interface set. Figure 6.5 shows definitions for these two. *PayloadModeVal.getval* is mapped onto two different channels one for CDH (*CDHPayloadModeValMemoryGet*) and the other for ADCS (*ADCSPayloadModeValMemoryGet*) and are included in the process alphabet

```
ADCSModeValState(init) = AssignableState(ADCSModeVal.setval, ADCSModeVal.getval,
                         ADCSModeVal.trans, init)
PayloadModeValState(init) = AssignableState(PayloadModeVal.setval, PayloadModeVal.
                            getval, PayloadModeVal.trans, init)
```

Fig. 6.3: Processes defining behavior of ADCSModeVal and PayloadModeVal.

```
MemoryUnits1 =
let
MemoryUnitProcesses = {(PayloadModeValState(Mode_Unpowered)),
                       (ADCSModeValState(Mode_Unpow))}
within (||| x:MemoryUnitProcesses @ x)
```

Fig. 6.4: Interleaved combination of all Memory_Unit processes, MemoryUnitProcess.

```
MemoryUnits = MemoryUnits1[[PayloadModeVal.getval <- Payload_State.getval,
                            PayloadModeVal.getval <- CDHPayloadModeValMemoryGet,
                            PayloadModeVal.getval <- ADCSPayloadModeValMemoryGet]]

MemoryUnitsIF = {|ADCSModeVal.setval, ADCSModeVal.trans, PayloadModeVal.setval,
                  CDHPayloadModeValMemoryGet, ADCSPayloadModeValMemoryGet|}
```

Fig. 6.5: System-level *MemoryUnits* process and its interface *MemoryUnitsIF*.

MemoryUnitsIF. *PayloadModeVal.trans* is not included in this set as there is no behavior associated with this transition, unlike the case with *ADCSModeVal.trans*.

6.1.3 ModeSystem

ModeSystem has complex semantics due to the number of checks and activities to be performed on a mode transition. All the modes are translated into a set of parameterized CSP processes. An example ADCS ModeSystem is shown in fig. 6.6. It consists of three modes and three transitions with Unpowered as the default mode. Rules embedded into the ModeSystem for each Mode related to its set of allowed modes are mapped onto a set of CSP functions *AllowedADCSModeSystem*() as in fig. 6.7. A complementary set of modes which indicates the disallowed modes is also defined using the function *DisAllowedADCSModeSystem*().

If the trigger for the transition is a command, a response message for it needs to be

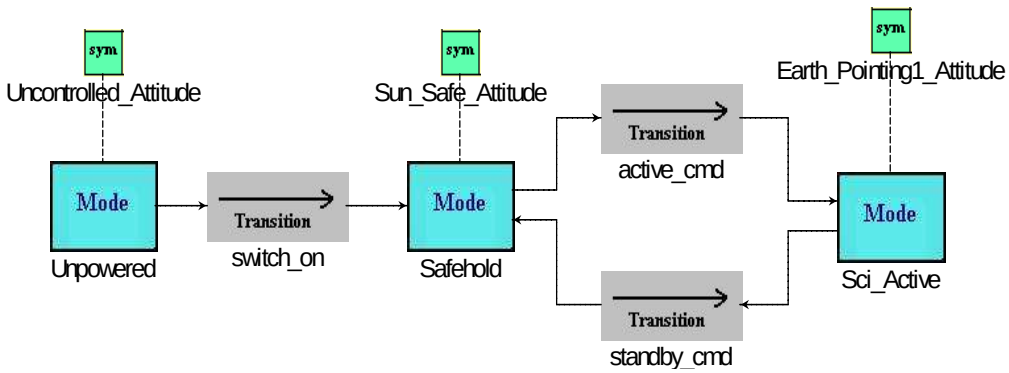

Fig. 6.6: An example ADCS ModeSystem.

```
AllowedADCSModeSystem(Safehold) = {Sci_Active}
AllowedADCSModeSystem(Sci_Active) = {Safehold}

DisAllowedADCSModeSystem(x) = diff(CommandableADCSModeCmd,AllowedADCSModeSystem(x))
```

Fig. 6.7: CSP corresponding to allowed and disallowed modes of ModeSystem of fig. 6.6.

generated. All three types of messages possible (success, fail, and insufficient power) are sent over the SystemBus as part of the ADCS Telemetry stream *ADCSTlm*. For trigger events other than commands, these messages are not generated. For any kind of transition trigger, a few common actions need to be taken as described by the *PreADCSModeSystemState*(*next*,-*curr*, *SendRsp*) process shown in fig. 6.8. First argument to this process is the mode entered when the transition succeeds, second argument is the current mode of ADCSModeSystem, and the last argument is a flag that dictates if a response message has to be generated for the mode transition. *PreADCSModeSystemState* is called from all the processes that represent Modes by passing the appropriate arguments as we see later in this section. Following is the list of checks/actions initiated as part of a mode transition.

- A check is made to determine if the required amount of power for the next mode is available. The PowerLevel telemetry stream *ADCSPowerStateTlmGet* is queried for the current power level and we make sure that it is part of the power level values defined in the PowerState SharedState (refer *PowerStateEnum* defined in sec. 6.2.4). This value is compared against the required power level derived from the mode to power level mapping function, *ADCSModePowerLevelFn*(*next*). *fPowerStateEnumFn* is a MapFunction, defined for the PowerLevel SharedState that is used for the comparison.

- If enough power is available to transition into the next mode, the *ADCSAcceptProcess* is called. This process, defined in fig. 6.9, checks if a positive response message *SystemBus.ADCSTlm.ADCSModeCmd_Accept* is to be generated based on the argument *SendRsp*. Next, attitude is set to the new value that corresponds to *next* mode using the mode to attitude map function, *ADCSModeSystemMapFn*(*next*). Finally,

the mode-specific actions are defined in the *ADCSModeSpecificSetVars* process. The change in power consumption resulting from change in mode is also communicated to the Power subsystem. A channel to indicate the begin and end of mode transitions is associated with each ModeSystem for this purpose, which in this example is the *ADCSModeSystemChannel*. Transitions on this channel signal the process representing SubSysModePower function to trigger the power consumption modifications. This communication is possible due to a parallel combination of these two processes that synchronize on *ADCSModeSystemChannel*.

- If there is not enough power to transition into the next mode, control stays in the current mode as defined by the $ADCSRejectProcess(next, curr, SendRsp)$ as shown in fig. 6.10. Based on the SendRsp flag, the message indicating insufficient power condition, *SystemBus.ADCSTlm.ADCS_Power_InsufficientMsg*, is generated.

We consider equivalent CSP for one of the ADCS modes, Safehold mode, which has an outgoing transition active_cmd to the mode Sci_Active. As can be seen from fig. 6.11, the process has external choice between three events. This external choice reflects all possible transitions out of the Safehold mode. Choice of the transition is dynamic, depending on which of the events in the external choice occurs first. The event for the first two choices is the same; the command present within active_cmd, which is the ModeCmd to command ADCS into the Sci_Active mode. Given this behavior of ADCS, if the CDH tries to command the ADCS to Unpowered mode, the ADCSModeCmd falls into the $DisAllowedADCSModeSystem(Safehold)$ and the message *SystemBus.ADCSTlm.ADCS_Cmd_Rej* indicating failure to move into the commanded mode is communicated to

```
PreADCSModeSystemState(next,curr,SendRsp) = ADCSPowerStateTlmGet?x:
{d|(d,r) <- PowerStateEnum} -> if fPowerStateEnumFn(x) >=
fPowerStateEnumFn(ADCSModePowerLevelFn(next)) then powerChange.ADCS
-> ADCSAcceptProcess(next,curr,SendRsp) else powerNoChange.ADCS
-> ADCSRejectProcess(next,curr,SendRsp)
```

Fig. 6.8: Definition of PreADCSModeSystemState process.

```
ADCSAcceptProcess(next,curr,SendRsp) = if (SendRsp == true) then
SystemBus.ADCSTlm.ADCSModeCmd_Accept!next -> ADCSModeSystemChannel.
begin!next -> Attitude.setval!ADCSModeSystemMapFn(next) ->
ADCSModeSystemChannel.end!next -> ADCSModeSpecificSetVars(next);
ADCSModeSystem(next) else ADCSModeSystemChannel.begin!next ->
Attitude.setval!ADCSModeSystemMapFn(next) -> ADCSModeSystemChannel.end!next
-> ADCSModeSpecificSetVars(next);ADCSModeSystem(next)
```

Fig. 6.9: Definition of ADCSAcceptProcess process.

```
ADCSRejectProcess(next,curr,SendRsp) = if (SendRsp == true) then SystemBus.
ADCSTlm.ADCS_Power_InsufficientMsg!m -> ADCSModeSystem(curr)
else ADCSModeSystem(next)
```

Fig. 6.10: CSP for ADCSRejectProcess.

the CDH. However, if appropriate command to move to Sci_Active is given, it falls into the $AllowedADCSModeSystem(Safehold)$ set. Subsequently, the $PreADCSModeSystemState$ process defined earlier is called. Behavior common across all the modes is indicated by the ControlTransition switch_off emanating from CommonMode as in fig. 6.12. The event generated for this common transition $power.load\_switch.ADCS.off$ is the third external choice event for the Safehold mode process.

The Safehold mode has a MemoryUnit_Write within it whose contents are shown in

```
ADCSModeSystem(Safehold) = (SystemBus.allADCSCmds.ADCSModeCmd?m1:
AllowedADCSModeSystem(Safehold) -> PreADCSModeSystemState(m1,Safehold,true))
[]
(SystemBus.allADCSCmds.ADCSModeCmd?m1: DisAllowedADCSModeSystem(Safehold) ->
SystemBus.ADCSTlm.ADCS_Cmd_Rej!m1 -> powerNoChange.ADCS ->
ADCSModeSystem(Safehold))
[]
(power.load_switch.ADCS.off -> powerChange.ADCS -> ADCSModeSystemChannel.begin!
Unpowered -> Attitude.setval!ADCSModeSystemMapFn(Unpowered) ->
ADCSModeSystemChannel.end!Unpowered -> ADCSModeSystem(Unpowered))

ADCSModeSpecificSetVars(Safehold) = PhotometerVal.setval!Spinning -> SKIP
```

Fig. 6.11: CSP corresponding to the Safehold mode of the ADCS ModeSystem of fig. 6.6.

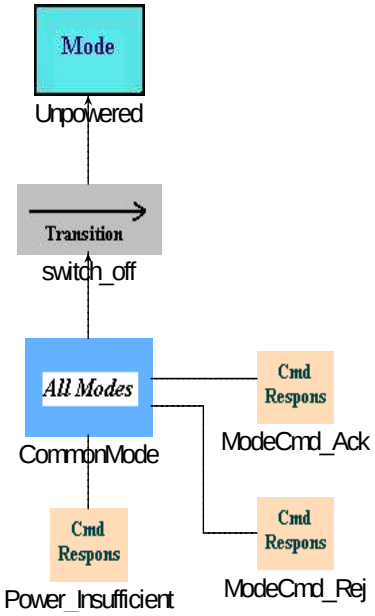

Fig. 6.12: Common transitions in the ADCS ModeSystem of fig. 6.6.

fig. 6.13. Event sequence representing this action is performed when this mode is entered in the process *ADCSAcceptProcess* (fig. 6.9), after the event *ADCSModeSystemChannel.end*!-*next*. In general, *ADCSModeSpecificSetVars*(*next*) has a sequence of all the Memory_Units that are set in the mode being considered. The order in which these are set is not important. Definition for Safehold mode process *ADCSModeSpecificSetVars*(*Safehold*) is in the last line of fig. 6.11. In this process, PhotometerVal is set to the value Spinning after which the process ends performing the termination event SKIP.

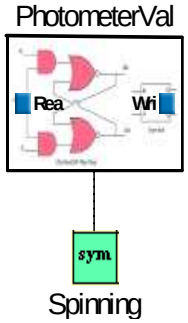

Fig. 6.13: Contents of the MemoryUnit_Write within Safehold mode.

6.1.4 Faults and Exceptions

All the generated faults are mapped onto exceptions (PointToPointMsgs) that are transmitted to CDH via subsystem telemetry streams. Their occurrence can be defined only in ModeSystems as part of a ControlTransition and the CSP involves specifying the PointToPointMsg that is to be handled by CDH.

6.2 CSP for Subsystem Specific Modeling Constructs

6.2.1 CDH

The ControlFlow related CSP semantics only are discussed in this section. CSP for rest of the constructs is identical to that mentioned in McInnes's dissertation [8].

ControlFlow:

Control Flow is CDH's FSM like construct that has ControlStates, ControlTransitions, AND_Connectors, OR_Connectors, and ControlStatePorts. CSP semantics for both ConrolStates and ControlTransitions are identical, resulting in an event or a sequence of events. The events used in a ControlTransition are controlled by another process (containing the transition events), by placing it in parallel with the CDH process. This leads to a wait-like scenario for ControlTransition. On the other hand, events in the ControlState are initiated by CDH, resulting in no wait time in CDH.

ControlTransition:

PointToPointMsg: A PointToPointMsg translates to a single CSP event over the subsystem telemetry stream. A PowerDropMsg present within a ControlTransition results in the event *SystemBus.PowerTlm.PowerDropMsg* where PowerTlm is the telmetry stream associated with Power subsystem.

CmdRef: Similar to the PointToPointMsgs, a CmdRef within a ControlTransition generates a single CSP event. However, based on whether the CmdRef is included in the Uplink CommandSet as well, different channels are used. *cmdin* channel is used when the

CmdRef is present in Uplink otherwise *SystemBus* is used. For example, a CmdRef SetADCSMode in the ControlTransition can result in the CSP event *cmdin.UplinkSetADCSMode* or *SystemBus.ADCSSetADCSMode*.

TelDataStreamRef and ControlDecition: A TelDataStreamRef followed by a ControlDecision has different semantics depending upon the number of connections coming out of it. A ControlTransition that has contents, as in fig. 6.14 (with the Symbol *Nadir_Pointing* within the SuccessSet), is a blocking transition that is mapped onto the event *SystemBus.ADCSTlm.AttitudeDataStream.trans?x : Nadir_Pointing*. If the connection between the telmetry stream and the ControlDecision were a "get" instead of "trans," the corresponding CSP event is *CDHAttitudeDataStream TlmGet?x : Nadir_Pointing*.

For the case where there are two transitions coming out of the ControlDecision, control proceeds to two different states based on whether the transition value is Nadir_Pointing or not. The CSP then is the sequence of events *SystemBus.ADCSTlm.AttitudeDataStream.trans?x → if x : Nadir_Pointing then State1Process else State2Process*. The two states are mentioned as State1Process and State2Process.

Memory_Unit and ControlDecision: The CSP semantics for this case are identical to the TelDataStreamRef and ControlDecision combination. Here, the SystemBus and telemetry stream channel combination is replaced with the name of the Memory_Unit. As these values are not relayed as streams in a spacecraft, they are not relayed onto the SystemBus.

SubSysPowerIfRef: A transition that waits on the switching on or switching off actions to a subsystem contains a SubSysPowerIf construct. The CSP for this involves communication over the power channel that is defined by a composite datatype to carry back and forth power related information within the spacecraft. BASSI finds out the subsys-

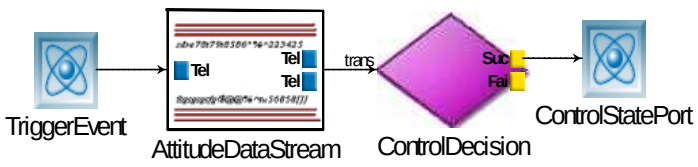

Fig. 6.14: Example ControlTransition waiting on an AttitudeDataStream transition.

tem the SubSysPowerIf belongs to and generates the CSP event accordingly. Switching on/off a subsystem causes the power consumption to vary and therefore, another event to indicate this change to the Power subsystem, power.load_delta, is appended after the switch event. CSP for a power switching ControlTransition targeting ADCS would be $power.load\_switch.ADCS.on \rightarrow power.load\_delta.ADCS!4$.

ControlState:

The sequence of actions that can be defined within a ControlState are are mapped onto a sequence of CSP events. These events are added to the CDH process interface set because CDH controls other subsystems through them. ControlStates which have multiple entry points or which are entered as a result of the branch of a ControlDecision result in the definition of a new CSP process. Rest of the ControlStates cause the event sequence to be appended to the process definition. Semantics for CmdRef, TelDataStreamRef, MemoryUnitRef, and SubSysPowerIfRef donot differ when used in a ControlState or ControlTransition, and hence are not discussed here.

ControlFlow: A controlFlow included in a ControlState causes the definition of a new process that has the name of the ControlFlow.

DLMessageRef: The DLMessages generated within a ControlState are put onto the SystemBus so that the Downlink subsystem can access them. An example DLMessage message ADCSModeStatusMsg associated with a Symbol Safehold to convey the ADCS mode is interpreted as $SystemBus.dl\_cmd!DLADCSModeStatusMsg.DLSafehold$.

ControlStatePort:

These ports facilitate flow of control beween constructs at different levels in the hierarchy and do not have much semantic significance. They are interpreted to a CSP process which has the name of the ControlState on which the connection from ControlStatePort finally terminates.

AND_Connector:

AND_Connectors are interpreted as the interleaving operator |||. The CSP semantics for an AND_Connector which has N outgoing connections from it is as shown below. *Next_Events* is the set of N events in the ControlTransition/ControlState on which the connections terminate and $P(x)$ is the set of N CSP processes that represent the flow of control following the each *Next_Event*.

$$ANDProcess = ||| \; x : \{Next\_Event1, Next\_Event2, ...Next\_EventN\} \bullet x \rightarrow P(x)$$

Figure 6.15 shows a portion of a ControlFlow that does command and control (Attitude_Sun_Safe transition branch) and simultaneously handles any PowerDropMsg (Dummy state branch) from the Power subsystem. CSP corresponding to this portion of the model is shown in fig. 6.16.

OR_Connector:

The CSP equivalent to OR_connector is the external choice operator "[ ]." The CSP semantics for an OR_Connector which has N outgoing connections from it is shown below.

$$ORProcess = \Box \; x : \{Next\_Event1, Next\_Event2, ...Next\_EventN\} \bullet x \rightarrow P(x)$$

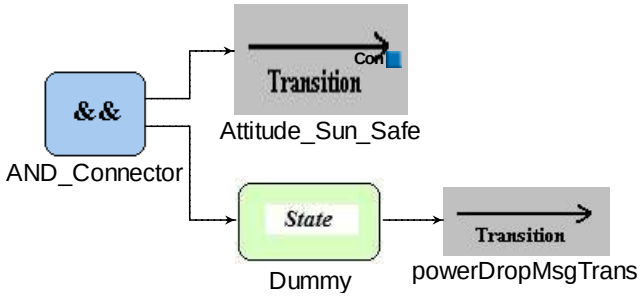

Fig. 6.15: Example ControlFlow containing AND_Connector.

```
Attitude_Sun_SafeProcess = SystemBus.ADCSTlm.Attitude_tlm_stream.trans?x:
                           {Sun_Safe_Attitude} -> ...
powerDropMsgTransProcess = SystemBus.PowerTlm.PowerDropMsg -> ...

AndProc0 =
let
Processes = {Attitude_Sun_SafeProcess, powerDropMsgTransProcess}
within(||| x: Processes @ x)
```

Fig. 6.16: CSP corresponding to the ControlFlow of fig. 6.15.

Figure 6.17 is an example ControlFlow where the OR_connector is used to indicate the possibility of occurence of any of the two messages; command accept or command reject. The transitions accept and reject have PointToPointMsgs Payload_Cmd_Accept and Payload_Cmd_Rej which signify success or failure of the command issued in the preceding ControlFlow. The corresponding CSP is shown in fig. 6.18.

6.2.2 ADCS

The ADCS does not involve interpretation of modeling constructs other than the generic

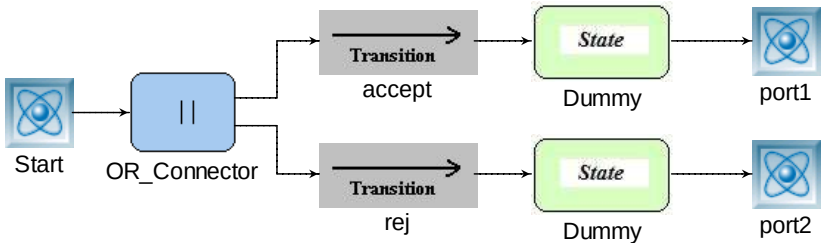

Fig. 6.17: Example ControlFlow containing OR_Connector.

```
Sample_Cmd_ResponseProcess = SystemBus.PayloadTlm.Payload_Cmd_Accept.CollectData
                             -> DL_SamplingProcess
                             []
                             SystemBus.PayloadTlm.Payload_Cmd_Rej.CollectData ->
                             Payload_SampleProcess
```

Fig. 6.18: CSP corresponding the ControlFlow of fig. 6.17.

constructs used in a subsystem. BASSI uses intermediate data structures to store process name and associated event interface set of each modeling entity used in the subystem so that the subsystem process can be built. The data structures are classes used to store strings or lists of simple information nodes, and hence their structural details are skipped. The subsystem process *ADCSProcess* and the interface set *ADCS_IF* are built according to the guidelines provided by McInnes [8].

6.2.3 Payload

Payload subsystem's modeling constructs are identical to those of ADCS, and hence BASSI behaves the same way as for ADCS subsystem.

6.2.4 Power

System-level power interfaces for power distribution and updation are generated by BASSI as the CSP channels *power.load_switch* and *power.load_delta*, respectively. CSP generated for the constructs PowerInitSequence, AttitudeSpecificAvailPower, and the CSP process that checks for power insufficiency (communicates power insuffciency by generating the *eps_exception* event) are generated by BASSI according to the ideas presented by McInnes [8]. Interpretation of the constructs introduced as part of BASS: PowerLevel, PowerDropMsg, and PowerStateToNumMap only are discussed in this section.

The PowerState is of type SharedState and is interpreted into an AssignableState process defined in the SBFL. The CSP datatype *PowerStateValues* encapsulates all the values PowerState can assume. AllowedValues of a SharedState is interpreted similar to the AllowedValues construct of Memory_Unit discussed in sec. 6.1.2. An example PowerState containing LOW and HIGH values is considered whose equivalent CSP is as in fig. 6.19. *PowerStateToNumMap* is the function whose domain is an enumeration of PowerState AllowedValues and range is a number that indicates the relative power threshold in the battery that supplies power. *fPowerStateEnumFn* is the MapFunction that takes power level as input and gives the number as output.

```
datatype PowerStateValues = LOW | HIGH
channel PowerState: StateIF.PowerStateValues

PowerStateEnum = {(LOW, 1), (HIGH, 2)}
fPowerStateEnumFn(c) = apply(PowerStateEnum, c)

PowerStateState(init) = AssignableState(PowerState.setval, PowerState.getval,
                        PowerState.trans, init)
```

Fig. 6.19: CSP generated for the SharedState PowerState.

Transitions between various levels of power in a PowerState are modeled as random as discussed in sec. 4.4. Hence, CSP processes to control behavior of the PowerState are also described as part of Power subsystem CSP and is shown in fig. 6.20. The *PowerStateTransitions* process gets the default PowerState value and creates a transition on the *PowerState_tlm_stream* so that all the interested subsystems are indicated of default power level. *PowerStateProcess*(*prev*) is the random process whose first channel *changePower* accepts any PowerLevel event that is either one level above or one level below the current value. This restriction aptly models how charge in a battery can either rise or fall only to the next immediate threshold level. If the power level falls, the PointToPointMsg PowerDropMsg is transmitted over the Power telemetry stream so that CDH can handle it appropriately. Subsequently, the power level is updated to the new value and subsystems affected by this variation are notified of this change over the *PowerState_tlm_stream*. *changePower* event is added to the Power interface set in order to make it externally visible. It can be included in the specification, and in that case its occurence is controlled by the SolarFlux events of the specification (ModelCheck). In the absence of a specification, FDR explores the system behavior for all possible random variations in the PowerState.

These processes are composed appropriately with rest of the modeling construct processes in order to obtain the subsystem composite process, *PowerProcess*.

6.2.5 Uplink and Downlink

The Uplink and Downlink subsystems perform limited functions as compared to the rest

of the subsystems. The Uplink subystem maps all the incoming commands on the *uplink* channel to the *cmdin* channel that feeds commands to the CDH. CSP for the Downlink subsystem maps all the messages arriving on *SystemBus.dl_cmd* channel onto the *downlink* channel. The *downlink* channel is characterized by its datatype which is a composition of all the DLMsgRefs CDH can generate. CSP processes for both Uplink and Downlink are reused from McInnes work [8].

6.3 Spacecraft System and ModelChecks

The individual subsystem processes, the MemoryUnits process, and their corresponding interface sets are composed in an alphabetized parallel fashion, to make up the Spacecraft-System as in fig. 6.21.

The desired system behavior, error events occurrence, deadlock-freedom, livelock-freedom can be checked by composing appropriate refinement specifications in CSP and loading them into FDR. FDR is an exhaustive state-space exploration tool that can verify the system requirements expressed in CSPM, the machine-readable dialect of CSP. It makes use of the denotational models of CSP and labeled transition systems in its operation. The utility of FDR is proved by its prior successful efforts in verifying security protocols [28, 29], asynchronous hardware design [30], and defense-applications [31].

```
channel changePower:StateIF.PowerStateValues

PowerStateTransitions = PowerState.getval?x ->SystemBus.PowerTlm.
                        PowerState_tlm_stream.trans!x -> PowerStateProcess(HIGH)

PowerStateProcess(prev) =
changePower.setval?new:{d|(d,r) <- PowerStateEnum, r==fPowerStateEnumFn(prev)+1
or r == fPowerStateEnumFn(prev)-1} -> if(fPowerStateEnumFn(prev) >
fPowerStateEnumFn(new)) then powerDropProcessDone -> PowerState.setval!new ->
SystemBus.PowerTlm.PowerState_tlm_stream.trans!new -> SystemBus.PowerTlm.
PowerDropMsg ->PowerStateProcess(new) else PowerState.setval!new ->
SystemBus.PowerTlm.PowerState_tlm_stream.trans!new -> PowerStateProcess(new)
```

Fig. 6.20: CSP processes modeling the PowerState.

```
SpacecraftSystem =
let
Subsystems = {(Power_IF, PowerProcess),
              (ADCS_IF, ADCSProcess),
              (Downlink_IF, DownlinkProcess),
              (Payload_IF, PayloadProcess),
              (CDH_IF, CDHProcess),
              (Uplink_IF, UplinkProcess),
              (MemoryUnits_IF, MemoryUnits)}
within (|| (IF, Subsys):Subsystems @ [IF] Subsys)
```

Fig. 6.21: CSP for the composite spacecraft system.

The important feature of CSP, refinement, is used to check the system properties. In BASS, refinement over traces, stable failures, and failures/divergence models is used. The traces model is used when we want to find out if an event or a set of events occur under all possible execution paths of the given process. The traces model is kind of objective and becomes inadequate in situations where there is non-determinism in the process, which is a result of the use of internal choice or internal events which are not controllable by the environment. The stable failures model is used in such scenarios and this model checks if a given sequence of events holds good for all possible execution paths. An additional feature added to the failures model results in the failures/divergence model which tests the process for divergence as well.

6.3.1 Inbuilt ModelChecks

The BASS Interpreter by default, builds a few a specifications based on the error events that have been defined as part of the design. Example for this is the resource overflow event *eps_exception* which is generated when power in the spacecraft is not adequate. The equivalent refinement check for the same is in fig. 6.22.

```
check1 = STOP
assert check1 [T= SpacecraftSystem \ diff(Events, {|eps_exception|})
```

Fig. 6.22: CSP refinement check that checks for power overflow.

6.3.2 User-Made ModelChecks

The user-made model checks discussed in sec. 5.4 are interpreted into refinement checks over the stable failures and failures/divergence models. The constructs used in the ModelCheck are mapped onto a simple sequence of events and in the process, each of the events is added to the corresponding interface set. Figure 6.23 shows an example specification process check2's definition. This process is in sequential composition with the event success. If the process behavior is as per the design requirements, the success event will occur, resulting in the satisfaction of the check in FDR. This approach is called "must-testing" and is adopted from McInnes work [8].

```
check2 = Check2Proc0

Check2Proc0 = downlink.ADCSModeStatusMsg.DLSafehold -> Check2Proc1
Check2Proc1 = downlink.ADCSModeStatusMsg.DLSci_Active -> SKIP

assert (success -> STOP) [F= ((check2;success -> STOP)
                              [|check2IF|]
                              SpacecraftSystem)\diff(Events,{success})

assert (success -> STOP) [FD= ((check2;success -> STOP)
                               [|check2IF|]
                               SpacecraftSystem)\diff(Events,{success})
```

Fig. 6.23: Example ModelCheck translation into CSP.

Chapter 7

Case Study for BASS: TOROID

The utility of a tool is proven only when it solves real-world problems. In order to demonstrate the capabilities of BASS, in this chapter we develop and verify a BASS model of a real-world small-satellite design. USU's small-satellite under construction, TOROID, is chosen for this task. TOROID is USU's entry into the University Nanosatellite Competition hosted by the Air Force Research Laboratory's (AFRL), Space Vehicles Directorate (VS), and the Air Force Office of Scientific Research (AFOSR) [32]. This competition provides students with an opportunity to design and build small-satellites that can potentially be flown in space. TOROID is not a design from scratch and heavily reuses the USUSAT1 and USUSAT2 small-satellite designs developed in earlier rounds of the competition. TOROID is not fully implemented yet, and hence has a volatile design. Our purpose is to illustrate the practical usability of BASS as applied to an actual spacecraft; consequently the design considered here does not necessarily correspond to the latest TOROID design.

7.1 TOROID Mission

TOROID stands for Tomographic Remote Observer of Ionospheric Disturbances, and is being developed to perform measurements of UV emissions in the ionosphere [33]. Its design is composed of seven subsystems: Spacecraft Configuration and Mechanical, ADCS, Power, Control and Management Software, Communications and Telemetry, Thermal Control, and Command and Data Handling [34]. The C&DH has a backplane into which electronic boards of other subsystems are plugged, providing data and serial buses for inter-board communication. The science operations are performed by a photometer which detects UV intensity. The spacecraft attitude must be actively maintained while performing the UV measurements. The appropriate spacecraft attitude for supporting the science mission is

nadir-pointing, indicating that the science instrument faces earth at all times. The Control and Management Software is defined as a subsystem in its own right, but executes on the C&DH hardware. Since there is little purpose to distinguish between hardware and software in a behavioral model, these two subsystems are combined and treated as a single C&DH subsystem in the BASS model. All the remaining TOROID subsystems other than the Configuration and Mechanical subsystem are also modeled in BASS.

The BASS model of TOROID's behavior primarily targets the mode-based behavior of the ADCS and Payload subsystems, a portion of the command handling aspects of C&DH, power related checks, and load-shedding behaviors. A comprehensive model is not considered because BASSMP currently does not support all aspects of spacecraft systems (e.g., we currently do not support the impact of Thermal or Structural considerations on the spacecraft behavior). Further, as with the construction of any model, the appropriate level of detail must be weighed against the goals of the modeling exercise. The focus of this model is the demonstration of BASS's modeling and verification capabilities; consequently, many details of the TOROID design are omitted. TOROID's design details were gathered from multiple sources, including Crace [34]. Some features from USUSAT presented by Srinivasan [35] are also modeled. System requirements are developed based on TOROID/USUSAT mission requirements. This case-study modeling TOROID in BASS progresses by covering the individual subsystems first, then composing a system-level model and finally composing and verifying the system requirements. The CSP generated by BASSI for the model presented here is included in the Appendix.

7.2 ADCS

7.2.1 Attitude

The attitude of interest for the Science mission, and correspondingly the ADCS subsystem is called *Nadir_Pointing*[1]. Maintenance of this attitude during science operations

[1] All objects referred to from the user-model are italicized.

ensures that the spacecraft science instrument faces earth, allowing the instrument to properly collect data. This and two more attitudes are defined within the SharedState Attitude as shown in fig. 7.1. *Uncontrolled* is used to represent the unknown attitude in which the spacecraft resides prior to powering the ADCS. *Achieving_Nadir* represents the attitude maintained during the transition to the *Nadir_Pointing* attitude.

7.2.2 ADCSModeSystem

TOROID's ADCS subsystem has eight modes of operation [34], as shown in fig. 7.2. *Tumbling* is the default state of the ADCS after it is switched on. *Detumble* is the state when the b-dot detumbling algorithm is applied to slow the spacecraft's rate of tumble. Once the tumbling has been slowed sufficiently, ADCS enters the *Detumbled* state. Once ADCS receives a valid TLE from the ground station and an attitude estimate is acquired, the *EstimateAcquired* state is entered. The *ControlAcquired* state implies that the ADCS has started using a magnetometer and torque coils to perform active attitude determination. Before any further deployment of the photometer can occur, the ground station makes sure that the spacecraft ADCS is in the *ControlAcquired* state. If the instrument has not been deployed, it is triggered in the *StationKeepingWithoutScience* state and only after the appropriate attitude related measurements are observed, the extension tube of the photometer can start spinning. The ADCS could be in this state for several orbits before it enters the *SpinUpPhotometer* state. Attitude estimates need to be made again due to the changes

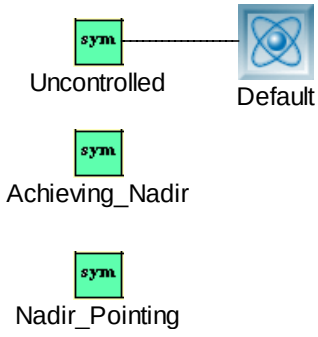

Fig. 7.1: AllowedValues within Attitude.

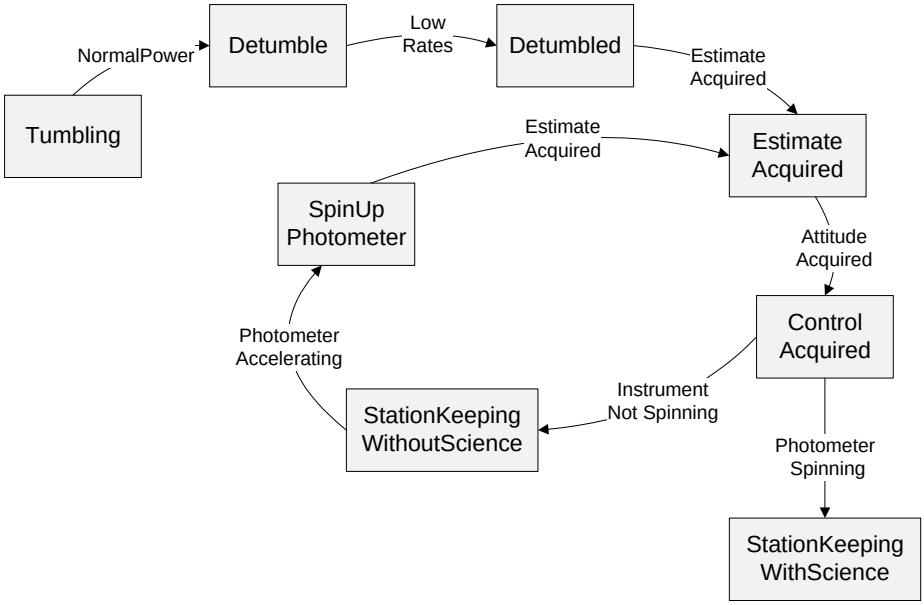

Fig. 7.2: FSM of TOROID's ADCS Manager.

caused by the acceleration of the photometer, before the final *StationKeepingWithScience* state can be entered.

When modeling the TOROID ADCS state machine as a ModeSystem in BASS, a distinction is made between different types of transitions in the ADCS state machine, based on whether those transitions are triggered due to the completion of some internal activity or by external events. Those triggered by internal activity are modeled as direct connections between modes. Those which are triggered by events external to the ADCS are modeled using ControlTransitions.

Figure 7.3 shows a portion of the ADCS ModeSystem in BASS, including the initial state through the *EstimateAcquired* state. A Mode *Unpowered* is the default state, and corresponds to the state of the ADCS state prior power-up. The ControlTransition *Switch_ADCS_On* is internally defined to trigger on the receipt of the Symbol defined in the ADCS SubSysPowerIf which indicates that power is turned on to the ADCS. This indicates that the transition will be taken only when power is supplied to the ADCS. The *Tumbling* and *Detumble* states are not controlled by any external event. Consequently,

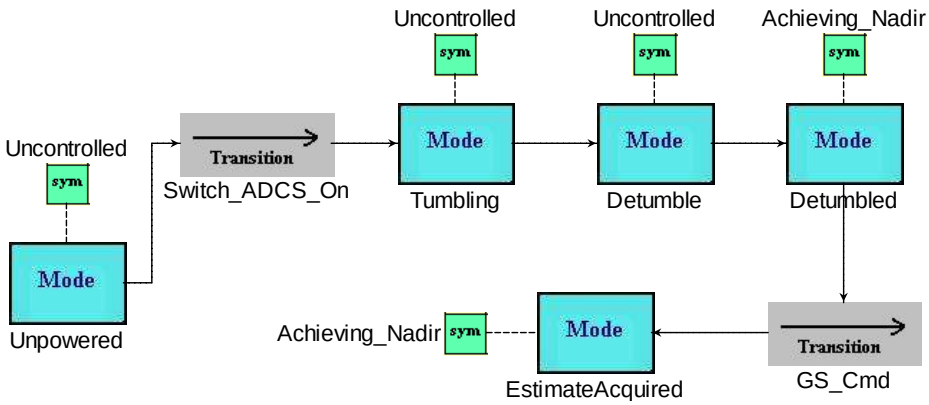

Fig. 7.3: First part of TOROID ADCS ModeSystem in BASS.

there are no ControlTransitions preceding these modes. When the ground station receives notification that the ADCS has entered the *Detumbled* state (handled by CDH as we see later), it transmits a TLE to the ADCS. This information is modeled as the parameter EstimateAcquired within the mode command triggered in response to a ground station command. Details about how a ground station command results in issue of the ADCS ModeCmd are covered as part of CDH in sec. 7.6.4. The internal definition of the *GS_Cmd* transition which contains the ADCS *ModeCmd* is shown in fig. 7.4.

Figure 7.5 shows the remaining portion of the ADCSModeSystem, including the *Estimate-Acquired* mode. The *EstimateAcquired* mode performs internal actions needed to acquire attitude control. Once those actions are carried out, the system immediately transitions to the *ControlAcquired* mode. In a real system, acquiring control of the attitude potentially requires multiple applications of complex control laws. In this model, we assume that

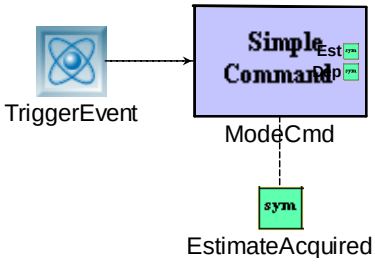

Fig. 7.4: Contents of ControlTransition GS_Cmd.

those control algorithms function correctly, and are successfully able to acquire control of the spacecraft. Alternative models could show the control acquisition process with more fidelity, but this was deemed unnecessary for the purpose of this model.

The mode following *ControlAcquired* is determined based on the state of the photometer, which can be either deployed or not deployed by the time control is acquired. ADCS triggers the photometer deploy and spin actions that are performed by the Payload subsystem, by raising events that are handled by the Payload subsystem. In addition to generating photometer related events, ADCS keeps track of the latest event raised by it so that it has knowledge of the next event to be raised. The Memory_Unit *Photometer_Events* is used for this purpose and can take the value *Photo_Deploy* or *Photo_Spin*, with *Photo_Not_Deployed* as the default state of the Memory_Unit. ADCS sets *Photometer_Events* to the appropriate values and the *trans* channel associated with it, used in Payload subsystem, models reception of the event by Payload. *Check_if_deployed* transition, whose contents are shown in fig. 7.6, checks if events triggering the deploy or spin actions have been raised earlier by including *Photo_Deploy* and *Photo_Spin* Symbols in the SuccessSet of the ControlDecision. Initially none of the events have been raised, therefore a query represented by the ControlTransition *Check_photo_deployed* results in a transition to the *WaitFor_GS_Cmd*. Alternatively, had the Photometer been deployed, a transition to the *Photometer_Deployed* mode would have occurred. In the *WaitFor_GS_Cmd* mode, the system waits for the arrival of a ground station command, instructing the spacecraft to initiate the deployment of the photometer. When such a command arrives, queried by the *GS_Cmd* transition, the system enters the *DeployPhotometer* mode. On entry to this mode, the Photometer should start spinning; consequently, the *Photometer_Events* is updated to the *Photo_Deploy* state inside this mode. The transition *Check_photo_spin* checks the photometer events triggered again, and if an event triggering Photometer spin has been raised, system transitions to the *Science_Active* mode, in which the Nadir_Pointing attitude is actively maintained. Otherwise, system enters the *Science_Idle* mode, and then transitions to the *SpinUpPhotometer* mode. In this mode, the photometer is commanded to spin up again, by updating the

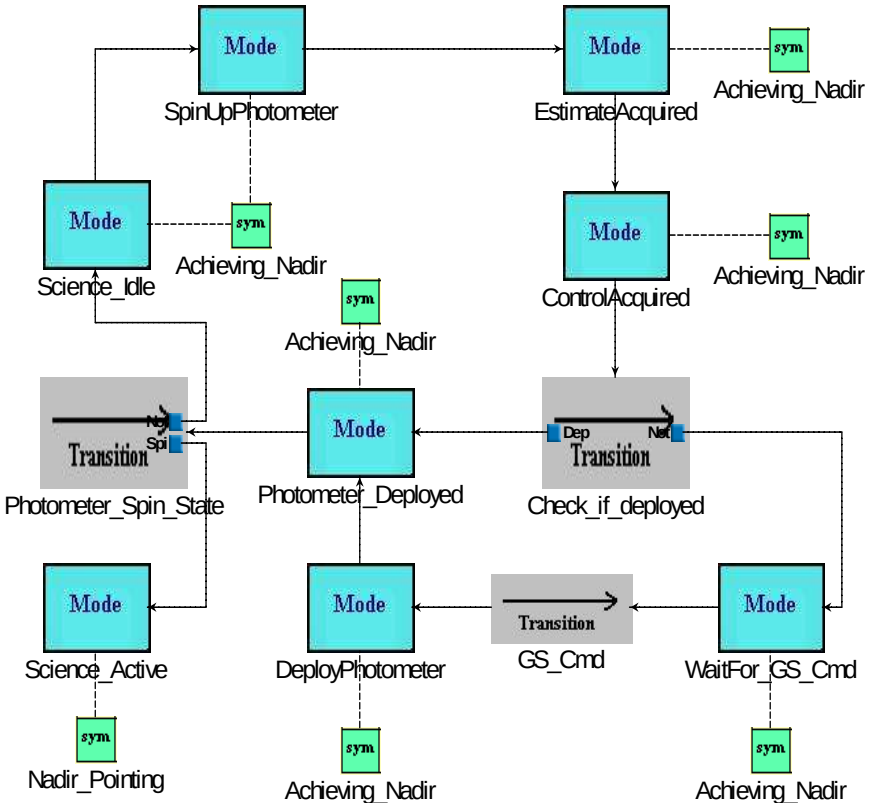

Fig. 7.5: Rest of the TOROID ADCS ModeSystem in BASS.

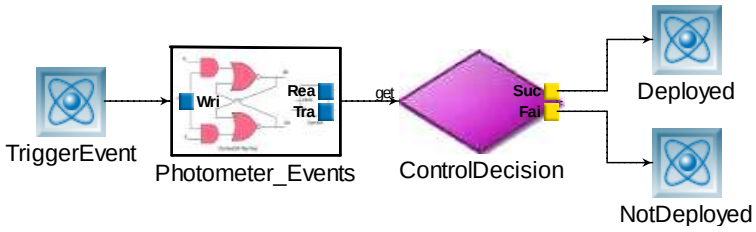

Fig. 7.6: Contents of check_if_deployed transition.

Photometer_Events to the value *Photo_Spin*.

TOROID uses a part of the Global Status Structure (GSS) to track the current ADCS mode and to make it available to other subsystems. In the BASS model, this variable is modeled using a Memory_Unit object named *ADCSModeVal*. Upon entry, each mode of the ADCSModeSystem updates the *ADCSModeVal* to its corresponding mode.

Each of the ADCS modes is mapped to one of the two qualitative PowerLevels LOW and HIGH as discussed in sec. 7.4.4. All the modes in ADCSModeSystem except *Science_Active* are mapped to LOW level of power.

7.2.3 ADCSModePower

The amount of power required for each mode of the ADCS is specified using this MapFunction, shown in fig. 7.7. Due to lack of available data, the current model associates arbitrary power values with each mode. However, the model could easily be modified to associate actual power values, once available.

7.2.4 ADCS Faults

In order to represent fault handling, the model includes a representation of a fault specified for ADCS hardware, and models the system behavior which reacts to the presence of that fault. The fault is flagged with the ADCS CONTROL HARDWARE NOT WORKING signal, defined in the USUSAT documentation. The documentation specifies the following reaction on receipt of the fault as shown in fig. 7.8.

Modeling a fault-handling design requires ADCS to trigger certain actions such as maintaining a standby attitude and communicating the fault to CDH. In addition, the CDH and Payload may need to perform some actions in response to the fault. The ADCS InternalFaults is added an *ADCS_HW_Fault* which generates *ADCS_Hardware_Exception*. Connections between these two objects are possible via ports on InternalFaults and Exceptions

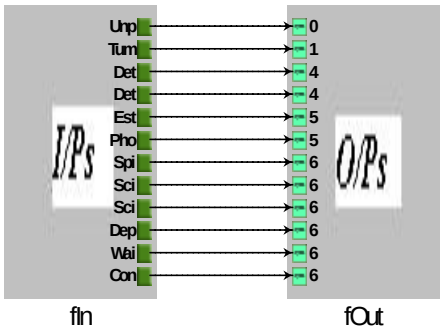

Fig. 7.7: SubSysModePower of ADCS.

"On receipt of the ADCS CONTROL HARDWARE NOT WORKING signal, the ADCS mode manager sends additional signals to turn off science data collection or high-rate data downlink, if they are on. This action is taken to compensate for the fact that the satellite attitude may no longer be correctly determined or maintained as a result of the signaled faults. On the reception of signals that indicate that the control hardware is working again, the mode manager does not automatically turn either science data collection or high-rate data downlink on. Instead, it waits for indication from the ground station to reactivate devices that have been shutdown."

Fig. 7.8: ADCS Hardware Fault Handling Design of USUSAT.

as shown later in sec. 7.2.5. The occurrence of an *ADCS_HW_Fault* embedded in the ControlTransition *Hard_fault* and its subsequent handling is shown in fig. 7.9. The occurrence of this fault when performing science operations causes ADCS to move into the *Wait_For_Spin* mode. ADCS waits in this mode for a command to restart the normal functioning which is modeled as the *start_spin_cmd* transition. From the *SpinUpPhotometer* mode, the *ADCSModeSystem* proceeds as in fig. 7.5 from the *EstimateAcquired* mode. In addition to the mode transitions, the *ADCS_HW_Fault* generates *ADCS_Hardware_Exception*, which is a PointToPointMsg sent to the CDH via the SystemBus.

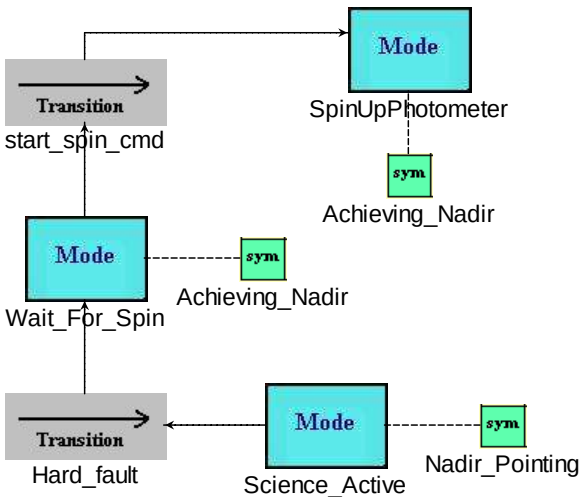

Fig. 7.9: Additional modes and transitions for ADCS_HW_Fault Handling.

7.2.5 Composite ADCS Model

The DataCommAspect model of the ADCS subsystem is presented in fig. 7.10. The ADCS employs two telemetry data streams. Information on the current spacecraft attitude is of interest to other subsystems, and is therefore relayed as the data stream *AttitudeDataStream*. The second data stream, *PowerLevelDataStream*, is used by the ADCS to determine whether the Power subsystem has sufficient power to permit a proposed ADCS mode transition. ACDS defines three PointToPoint messages to respond to commands received from CDH, *ADCSModeCmd_Accept*, *ADCSModeCmd_Reject*, and *ADCSPowerInsufficientMsg*. The *ADCSModeCmd_Accept* is sent when a command received from the CDH is successfully carried out. *ADCSModeCmdRej* (not visible in fig. 7.10, but as a port on the Exceptions construct) is sent to the CDH when the commanded mode transition is illegal according to the ADCSModeSystem. Since this message indicates an inconsistency in the CDH-ADCS intferface, it is specified as an exception by including it in the ADCS Exceptions construct. *ADCSPowerInsufficientMsg* is sent to the CDH when a mode change command is rejected due to insufficient available power. The TOROID design does not specify such message responses for each command, but does specify behavior in response to command rejection. The CommandSet has only two commands: the ADCS *ModeCmd* and the *start_spin_cmd*, both of which were described previously.

The PowerAspect for the ADCS (not shown) has only two constructs: the ADCSModePower object, mentioned above, the SubSysPowerIf object modeling the power interface for the ADCS. The ExceptionsAspect simply defines the mapping between the ADCSFaults and ADCSExceptions.

7.3 Payload

7.3.1 PayloadModeSystem

Similar to the ADCS subsystem, the TOROID design specifies a finite state machine model governing the behavior of the Payload subsystem, which is shown in fig. 7.11. On power-up, the Payload performs some internal initializations, and then waits for an event

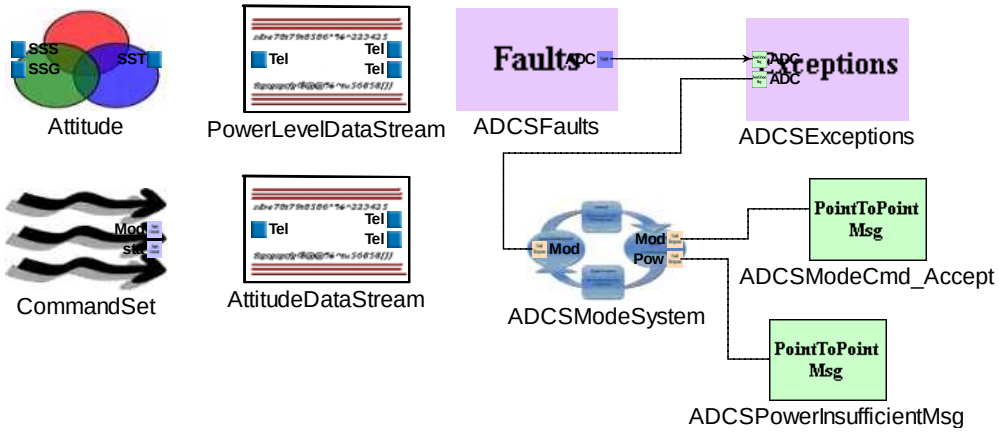

Fig. 7.10: DataCommAspect of the ADCS subsystem model.

prior to commencing the photometer deployment. The *DeployPhotometer* event is triggered by the ADCS when it receives a deploy command from the ground station. In *DeployPhotometer* state, the frangibolts holding the instrument are fired, releasing the photometer. The instrument starts accelerating to the desired rate after receiving the *SpinUpPhotometer* event. This event is triggered by ADCS when the instrument starts spinning as a result of the actions in *SpinUpPhotometer* state of fig. 7.3. Once the photometer has spun up, one of two operations are performed, based on a set of values queried from TOROID's GSS. First, a threshold voltage for the instrument can be set in the *SetThresholdVoltage* state. Once the voltage threshold is set, the system returns to the *CheckGSSFlags* state. Second, data can be sampled in the *CollectData* state. In the *CollectData* state, the photometer takes samples of UV emissions until the end of duration indicated by a timer event. Upon the timer event, the Payload Manager enters the *CheckGSSFlags* state again. The start time and the duration during which data needs to be collected is uplinked by the ground station and is updated by the VHF Manager (part of the CDH software).

An adaptation of this FSM into a ModeSystem in BASS is shown in fig. 7.12. Unpow is the default mode, representing the state of the payload prior to its receiving power from the Power subsystem. The *switch_on* ControlTransition is triggered by the receipt of the "on"

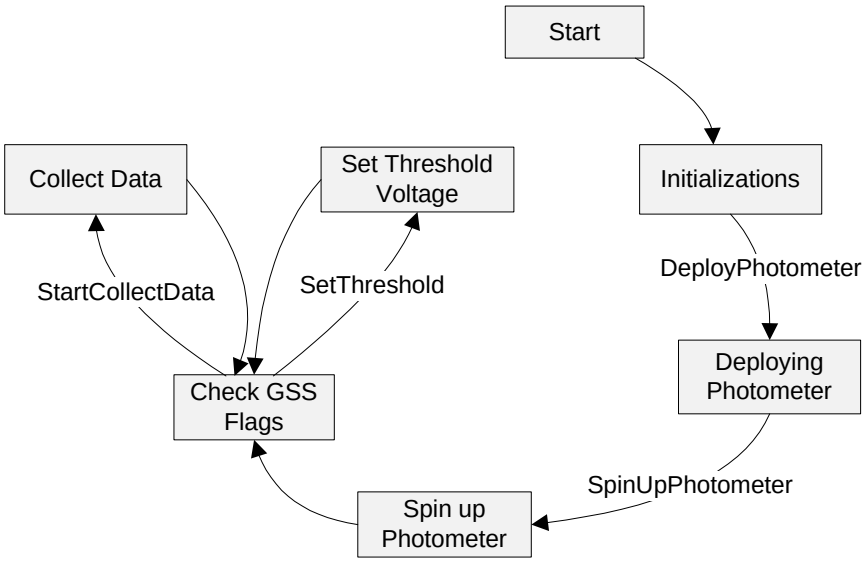

Fig. 7.11: FSM of TOROID's Payload Manager.

symbol defined in the Payload's SubSystemPowerIf, modeling the initial powering of the Payload subsystem. The system remains in the *Initializations* mode until ADCS changes the value stored in the *Photometer_Events* Memory_Unit. The *flag_deployed* ControlTransition is triggered when the *Photometer_Events* is set to *Photo_Deploy*. The decision logic to query the *Photometer_Events* object using a ControlDecision construct is shown in fig. 7.13.

Once deployed, the extension tube of the photometer starts spinning only after the *Photometer_Events* value is set to *Photo_Spin* by the ADCS. The transition *flag_spinning* polls the *Photometer_Events* similar to the *flag_deployed* transition. Transitions from *Wait_For_Cmd* are based on the values found in the GSS set by the VHF Manager. Since the BASS model does not explicitly represent the GSS, we model these values as commands because though communicated in a different way, they are controlled by CDH. The timeout generated by CDH which results in stopping the payload sampling is also modeled as a command. These three commands are defined as part of the command interface for the Payload subsystem, shown in fig. 7.14. The *SampleInstrumentCmd* is issued to start collecting samples from the Photometer. The *StopSamplingCmd* is issued to stop collecting samples. The *SetPhotometer_Threshold* command is issued to set the threshold value for the photometer board. These commands are used as triggers in the transitions associated

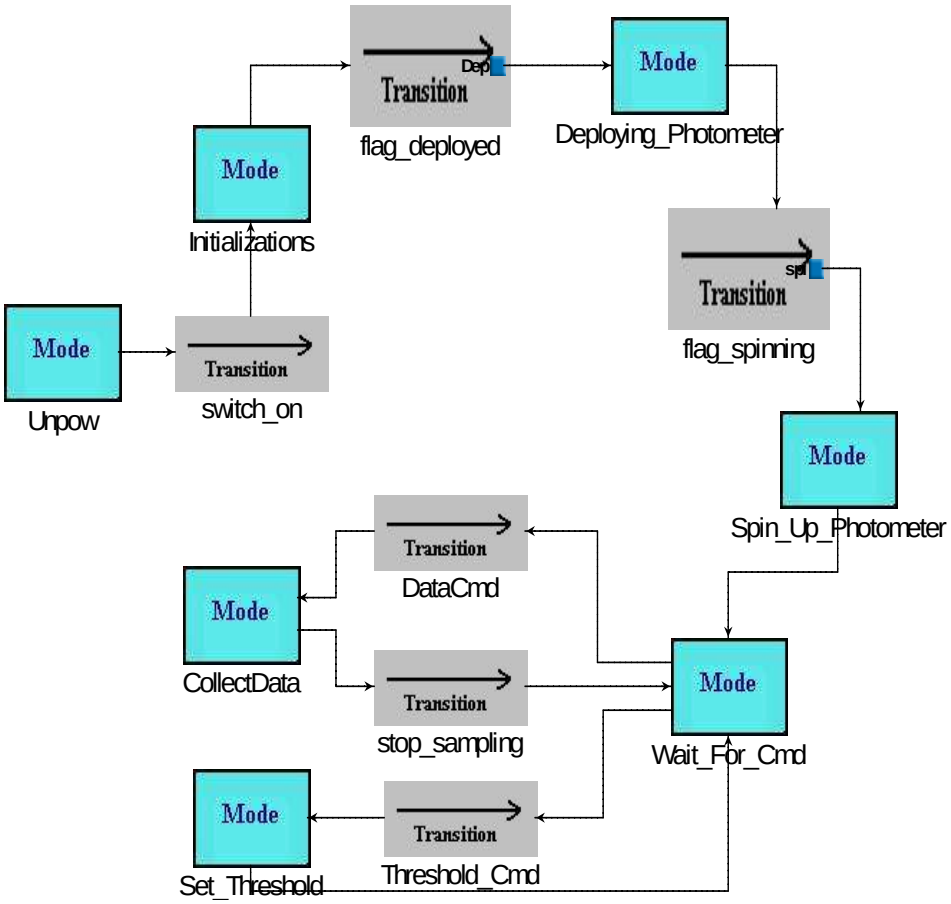

Fig. 7.12: TOROID Payload ModeSystem in BASS.

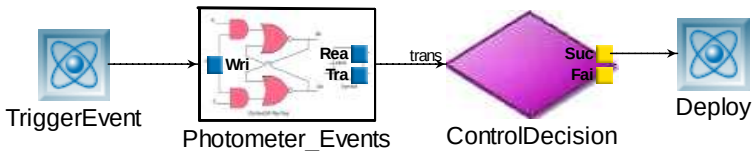

Fig. 7.13: Contents of ControlTransition flag_deployed.

with the *Wait_For_Cmd* mode.

7.3.2 PayloadModePower

The mode to power consumption mapping in Payload is modeled using its SubSys-ModePower construct. All the Payload modes other than the unpowered mode *Unpow* are arbitrarily set to consume one unit of power.

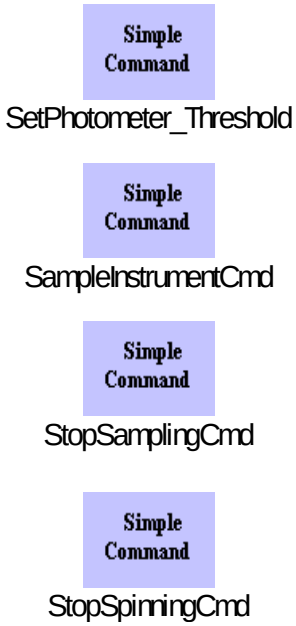

Fig. 7.14: Contents of the Payload subsystem CommandSet.

7.3.3 Fault Handling in Payload

According to the fault handling design discussed in sec. 7.2.4, data sampling operations need to be stopped when the ADCS detects a hardware fault. This requirement is embedded into the PayloadModeSystem by adding appropriate transitions between the modes. An additional command called *StopSpinningCmd* is added to the Payload CommandSet which is triggered by the CDH on receipt of *ADCS_Hardware_Exception*. This is included in the transition *stop_spinning_cmd* shown in fig. 7.15, causing the Payload to enter the *Wait_for_Deploy* mode again.

7.3.4 Composite Payload Model

The DataCommAspect of the Payload subsystem is shown in fig. 7.16. Since the *PowerLevelDataStream* must be queried for all the mode transitions within the PayloadModeSystem, it is made part of Payload subsystem. The CommandSet shown in fig. 7.14 has the commands mentioned earlier. The PowerAspect has only two constructs: PayloadModePower and a SubSysPowerIf object.

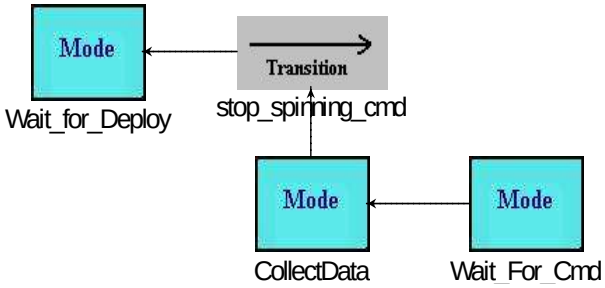

Fig. 7.15: StopSpinningCmd added as ControlTransition to PayloadModeSystem.

7.4 Power

TOROID's design includes power generation and load-shedding behaviors. However, since they are not well documented, the BASS model presented here adopts a possible design for the Power subsystem, including the specification of PowerInitSequence, PowerPorts, Commands, and AttitudeSpecificAvailablePower. To demonstrate how the load-shedding behavior can be modeled using BASS, a single battery threshold level is considered, thus creating two values for the PowerState: LOW and HIGH. The modeled behavior on observing a fall in power level is derived from USUSAT's load-shedding design.

7.4.1 CommandSet

The CommandSet of the Power subsystem consists of commands that toggle power on the PowerPorts. It consists of simple commands to switch the Uplink, Downlink, Payload,

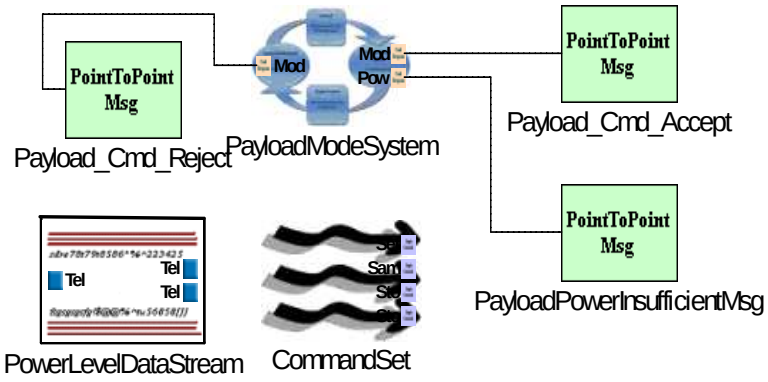

Fig. 7.16: DataCommAspect of the Payload subsystem model.

and CDH subsystems as shown in fig. 7.17.

7.4.2 PowerInitSequence

The order in which the subsystems are switched on is modeled by creating directed connections among the *ULSwitch*, *DLSwitch*, *CDHSwitch*, and *PayloadSwitch* commands. Since the ADCS is switched on by CDH, it is not included in this model.

7.4.3 AttitudeSpecificAvailablePower

The quantity of power generated in various attitudes (*Uncontrolled*, *Achieving_Nadir*, and *Nadir_Pointing*) is represented using the AttitudeSpecificAvailablePower MapFunction. In this case study, *Uncontrolled*, *Achieving_Nadir*, and *Nadir_Pointing* attitudes correspond to the generation of 15, 22, and 20 units of power, respectively.

7.4.4 PowerState

As mentioned earlier, this study only considers a single power threshold (and two states LOW and HIGH). Using the MapFunction PowerStateToNumMap, the qualitative states LOW and HIGH are assigned numerical values 1 and 2, respectively. These numbers are chosen randomly such that they are distinct as in a C/C++ enum datatype and are

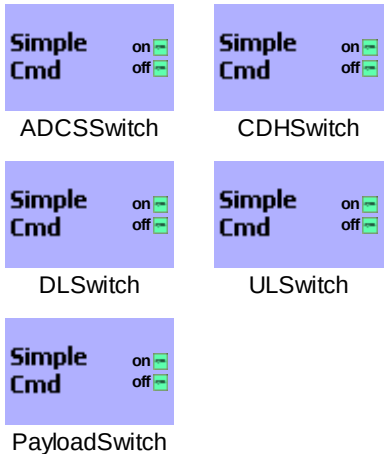

Fig. 7.17: CommandSet of Power subsystem.

relatively proportional to the power threshold values of interest. The PowerState value is streamed over the SystemBus via the *PowerLevelDataStream*.

7.4.5 PowerDropMsg

A *PowerDropMsg* is a special PointToPointMsg generated in response to a fall in the PowerState level, and must be handled by the CDH. It is similar in behavior to an exception. The load-shedding behavior modeled here was based on the the load-shedding specification of USUSAT, presented in fig. 7.18. This specification is adapted to TOROID by carrying out the following actions when a *PowerDropMsg* is received by the CDH, indicating that the power level has dropped to a level requiring load-shedding: the science operations of the Payload subsystem are shut down, and the attitude is allowed to drift from *Nadir_Pointing*. Consequently, CDH handles the *PowerDropMsg* and *ADCS_Hardware_Exception* messages in the same fashion. The science operations are restarted by CDH only after the receipt of an appropriate command from the ground station.

7.5 Uplink and Downlink

Uplink and Downlink subsystems behavior reflects the interactions between TOROID and the ground station. Crace [34] defined several sequence diagrams showing spacecraft system and subsystem interactions. In particular, fig. 7.19, derived from Crace's thesis [34], shows the interaction between TOROID and the ground station. The commands and mes-

"Whenever the power in the system decreases, signals are sent out to subsystems to shut down devices...Thus, whenever a PWR_STATUS_LOW signal is received as a result of transition from the PWR_STATUS_NORMAL to the PWR_STATUS_LOW state, the STAR_CAMERA_OFF signal is sent to the science subsystem, and the ADCS_POWER_OPTIMAL_ATTITUDE_ON signal is sent to the ADC subsystem...No device or subsystem is automatically turned on when the power state in the system increases from a lower power state to a higher power state. Consequently, no signals are sent to any of the other subsystems due to such a state transition in the power subsystem. However, the current power level is updated appropriately to ensure that subsystems can be turned on when commanded by the ground station."

Fig. 7.18: Load-shedding design of USUSAT.

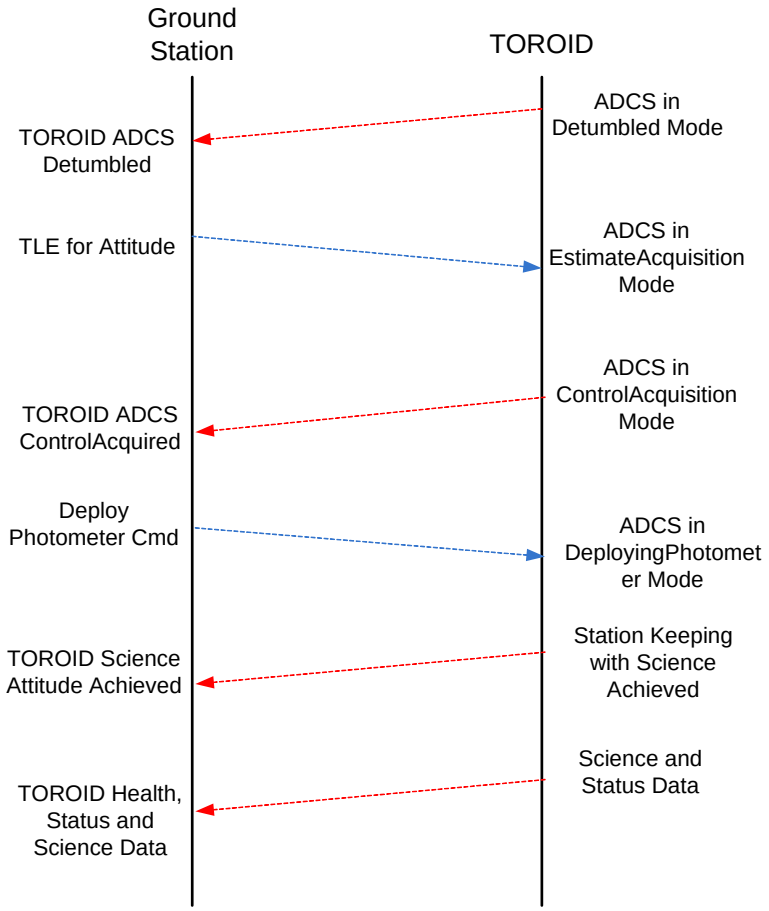

Fig. 7.19: Sequence diagram showing a few ground station-TOROID interactions.

sages shown in this figure are limited to the scope of the BASS model considered in this chapter. The messages downlinked by TOROID (shown on the right side) consist of ADCS mode status messages, science data, and status data for the spacecraft. The Downlink subsystem contains references to the DLMessages discussed in sec. 7.6.3. The TLE that allows ADCS to move to the EstimateAcquired mode and the Deploy Photometer command are uplinked by the ground station. These two commands are embedded as Symbols in the parameterized command SetADCSMode, which is contained in both the CDH and Uplink CommandSets.

7.6 CDH

The CDH design presented in this section discusses a possible design approach in order to achieve the desired system-level behavior of TOROID. If the developed design adheres to system requirements, it not only verifies the design but also provides important guidelines in implementing the CDH.

7.6.1 CommandSet

The CDH CommandSet has 10 commands that are equivalent to the subsystem commands of Power, ADCS, and Payload. Figure 7.20 shows the commands defined in the CDH CommandSet.

7.6.2 CDHCmdDispatch

Figure 7.21 shows, on the left side, a subset of the CDH commands defined in fig. 7.20, and the subsystem commands they trigger to the right. The remaining commands defined in the ADCS and Payload subsystems for controlling the photometer state and science data sampling are also mapped in a similar fashion. The issue of CDH commands is modeled using ControlFlow constructs, as discussed in sec. 7.6.4.

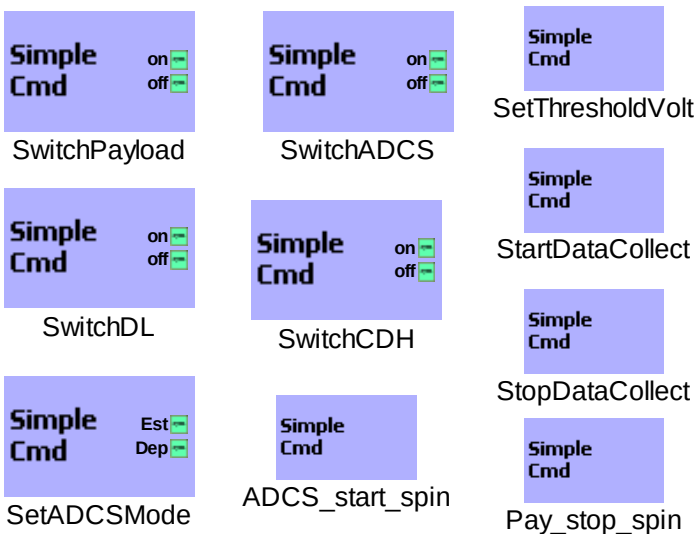

Fig. 7.20: CommandSet of the CDH subsystem.

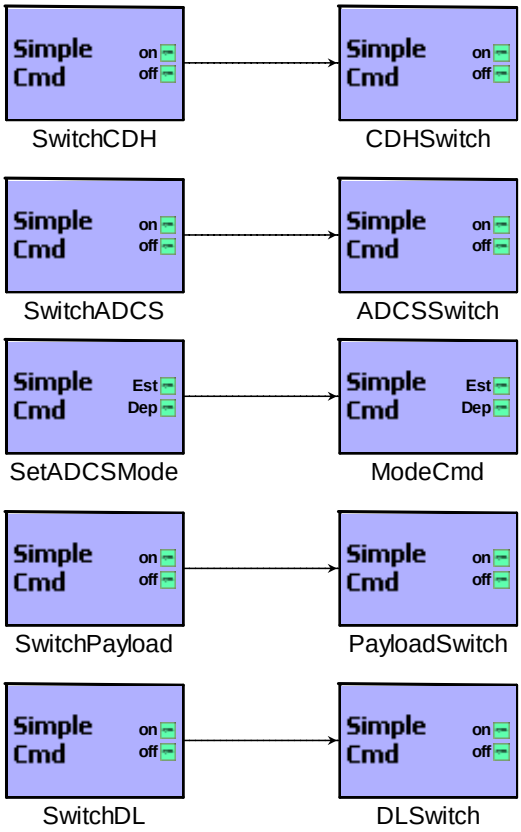

Fig. 7.21: CDHCmdDispatch containing power related commands.

7.6.3 DLData

As indicated by fig. 7.19, only a few ADCS modes are of interest to the ground station that controls TOROID. Consequently, ADCS mode information is to be downlinked only when a transition to the *Detumbled*, *ControlAcquired*, *Science_Active*, or *SpinUpPhotometer* mode occurs. The CDH ControlFlow transfers current ADCS mode information to Downlink via the *ADCSModeStatusMsg* message with the corresponding ADCS mode selected as a parameter.

7.6.4 ControlFlow

The ControlFlow presented here describes not only the normal sequencing of command and information, but also exception handling and load-shedding behaviors. Hence, the *PowerDropMsg* and *ADCS_HW_Exception* need to be handled as part of ControlFlow.

Before science operations can be started, ADCS must be switched on and the photometer must be spinning. CDH issues commands to the ADCS to guide it to the *ControlAcquired* mode in preparation for science operations. The states and transitions of the ControlFlow to this point are shown in figs. 7.22, 7.23, and 7.24. In fig. 7.22, the first ControlTransition *CDHPowerOn*, which uses the SubsysPowerIf of the CDH as its TriggerEvent, ensures that the CDH is powered on. The ADCS is switched on in the S*witchADCSOn* state, which causes the ADCS mode to transition from *Unpowered* to *Tumbling* and the Attitude from *Uncontrolled* to *Achieving_Nadir*. All transitions in Attitude are communicated to the CDH (as it has registered to receive transitions on the ADCS *Attitude_Data_Stream*), and hence a ControlTransition *Attitude_transition* is modeled, which expects a change in attitude. *Attitude_transition* has a trans connection from the *Attitude_Data_Stream* followed by a ControlDecision whose SuccessSet contains the *Achieving_Nadir* Symbol. The subsequent ControlFlow is embedded within the state *NextControlFlow*, shown in fig. 7.23.

CDH must track the ADCS modes of interest mentioned in sec. 7.6.3. This tracking is achieved by synchronizing on the transitions in *ADCSModeVal*. Information about a few other modes may also be needed by CDH in order to confirm entry into the expected ADCS mode. The ControlFlow in fig. 7.23 shows the *ADCS_Detumbled* mode, which waits on the *ADCSModeVal's* transition to *Mode_Detumbled*. This transition in mode is communicated to the ground station in the *DL_Detumbled* state using the *ADCSModeStatusMsg*. Following this, CDH waits for an *ADCSModeCmd* from the ground station that can move the ADCS into *EstimateAcquired* mode. This command is translated into the ADCS command *ModeCmd* as specified by the CDHCmdDispatch. Consequently, ADCS enters *EstimateAcquired* mode after generating an acknowledgment message. This message needs

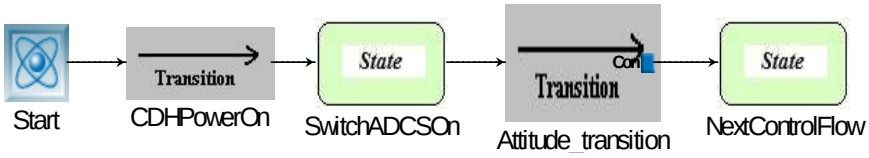

Fig. 7.22: Contents of CDH ControlFlow.

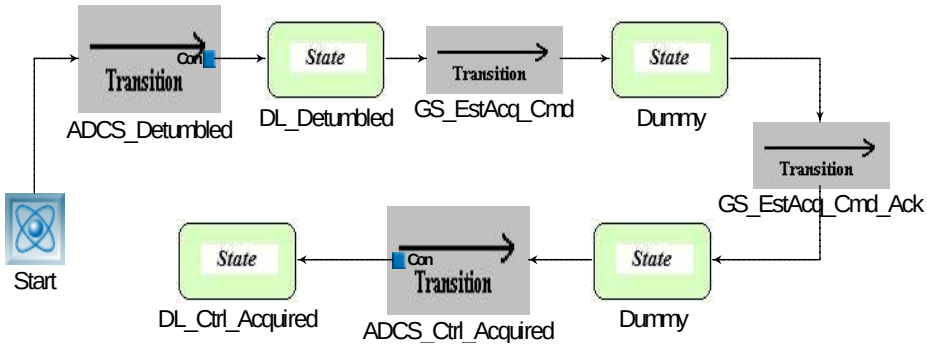

Fig. 7.23: Contents of NextControlFlow.

to be handled by the CDH and is modeled within the *GS_EstAcq_cmd_Ack* transition. States labeled *Dummy* do not contain any actions; they enable connections between two ControlTransitions.

The system next waits until the ADCS enters the *ControlAcquired* mode, as reflected by the value stored in the *ADCSModeVal*. Once this ADCS mode is achieved, the *ADCS_Ctrl_Acquired* transition gets triggered and this new mode is downlinked in the *DL_Ctrl_Acquired* state.

Figure 7.24 shows the continuation of the *NextControlFlow* system model. When the ground station receives notification that the ADCS has entered the *ControlAcquired* mode, it issues a command to deploy the photometer instrument. The *SetADCSMode* command is received by the CDH and dispatched to the ADCS, within the *GS_Deploy_Photo* transition. After the CDH receives an acknowledgment for this command from the ADCS, it synchronizes on the transitions in the *ADCSModeVal* to *Mode_Spin_Photo* and *Mode_Ctrl_Acq*. When the ADCS, following its internal transitions, enters the *ControlAcquired* mode with the photometer spinning, it is ready to enter the *Science_Active* mode without any further control by CDH. At this time, the Payload subsystem releases the photometer and sets *PayloadModeVal* to *Mode_Release_Photo*. The CDH reacts to this state change, with the behavior shown in fig. 7.25.

Transition to the Payload mode *Wait_For_Cmd*, is modeled as part of the *Payload_Wait_For_Cmd* transition. Following this transition, the CDH makes sure that attitude of

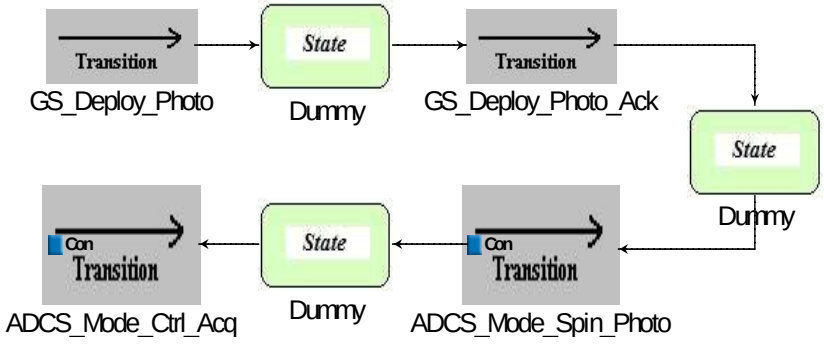

Fig. 7.24: Contents of NextControlFlow continued from the state DL_Ctrl_Acquired.

the spacecraft reaches the desired science attitude *Nadir_Pointing*. The *Att_Trans_Nadir* waits for a transition on the *AttitudeDataStream* corresponding to this attitude. When this attitude is achieved, the ground station is notified of the same in the *DL_SciActive* mode by sending the ADCSModeStatusMsg with the *Mode_Sci_Active* parameter selected. The rest of the control that follows once the desired attitude is achieved is modeled within the ControlFlow in the *AfterScience_Attitude*, which is shown in fig. 7.26.

When the science attitude is achieved and the Payload is in the *Wait_For_Cmd* mode, either the threshold voltage can be set or science data can be collected. In this design, the process of setting the threshold voltage, followed by the collection of samples from the ionosphere is repeated. Samples are taken only between the receipt of commands to start and stop data sampling. When in the science attitude, the power consumption is HIGH, leaving

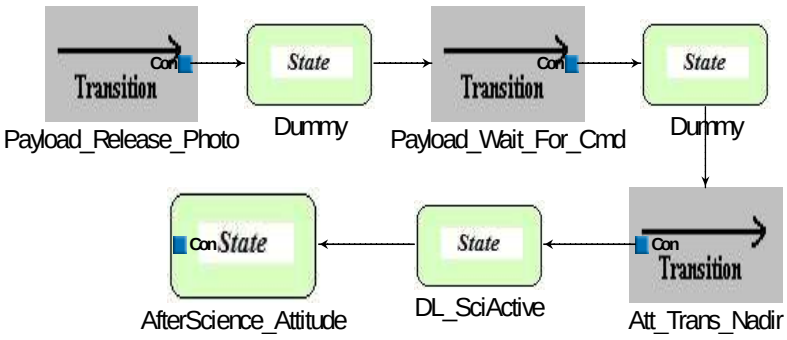

Fig. 7.25: Contents of NextControlFlow continued from the transition ADCS_Mode_Ctrl_-Acq.

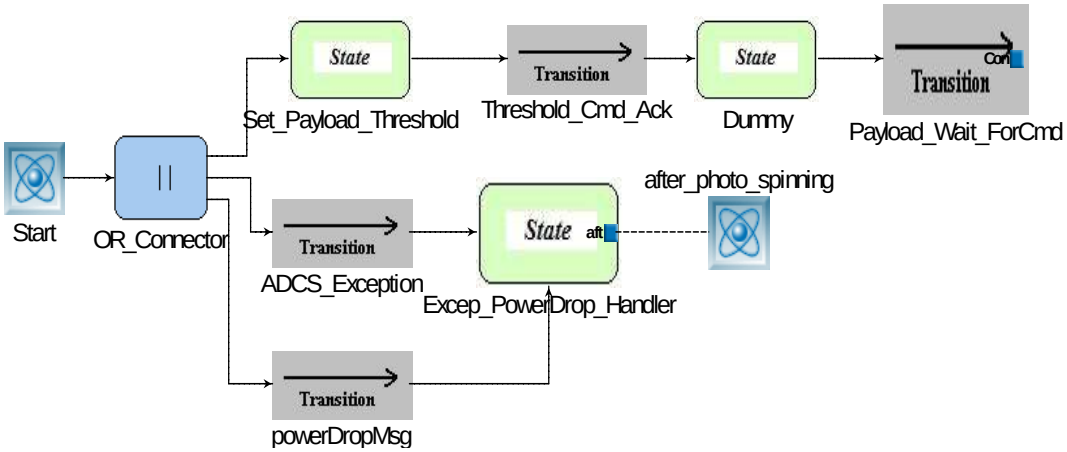

Fig. 7.26: Contents of the ControlFlow within the ControlState AfterScience_Attitude.

open the possibility of receiving a *PowerDropMsg* or the *ADCS_HW_Exception*. These two messages are handled in the same way, by suspending the sampling operations and moving the ADCS and Payload subsystems into their respective standby modes, *start_spin_cmd* and *Wait_For_Deploy*. An AND_Connector can be used between the normal command sequence and these two messages implying that they can occur at all times. However, such a model translated into CSP could not compile in the model-checking tool FDR due to the huge number of states which resulted. As an alternative, the OR_Connector construct was used to model three different paths of execution, any of which could be exercised, but not interleaved, at a given time. The first path models the normal sampling behavior of the payload. The second and third model the receipt and reaction to the two exception messages. Since the drop in power or the fault do not occur unless they are driven by a regulating process (in the form of a ModelCheck model), control always proceeds to the path modeling normal sampling behavior when the system is evaluated in isolation. However, when the system is evaluated in conjunction with a ModelCheck which introduces power drop faults, a must-testing approach is followed to verify the ModelCheck which ensures that the specified *PowerDropMsg*/fault is chosen at the OR_Connector. This approach adheres to the fault-tolerance approach discussed by Roscoe [22].

When the system receives either a *PowerDropMsg* or an *ADCS_HW_Exception*, CDH

enters the state *Excep_PowerDrop_Handler*. The *Threshold_Cmd_Ack* transition models the issue of an acknowledgement message on the receipt of the *SetThresdholdCmd* command by the Payload. CDH waits until the PayloadModeSystem returns to the *Wait_For_Cmd* mode before issuing any commands to start science data sampling. In fig. 7.27, the state *Payload_Sample* issues the *StartDataCollect* command to the Payload. Since it is possible for a *PowerDropMsg* or *ADCS_HW_Exception* to occur before the Payload can actually start sampling, multiple output connections are associated with *Payload_Sample*. This state internally has an OR Connector based ControlFlow identical to the one shown in fig. 7.26. The first connection from *Payload_Sample* state transitions to *Pay_Sample_Resp*, a state in which a response message sent by the Payload is received and processed by the CDH. When the Payload begins sampling, it sends a response message, and then proceeds with sampling. Sampling continues until the CDH issues the command to stop sampling operations. Once such a command is issued, control returns to the *Payload_Sample* state, after which the data collection process can begin again, as directed by the CDH. The second connection transitions to the *Excep_PowerDrop_Handler* state that handles the *PowerDropMsg* or *ADCS_HW_Exception*. The *Excep_PowerDrop_Handler* state models the behavioral responses to the receipt of either the *PowerDropMsg* or the *ADCS_HW_Exception*.

Excep_PowerDrop_Handler state is entered when either the *ADCS_HW_Exception* or the *PowerDropMsg* signals are received by the CDH and its contents are shown in fig. 7.28. An occurence of either of these two messages causes transition in the ADC-

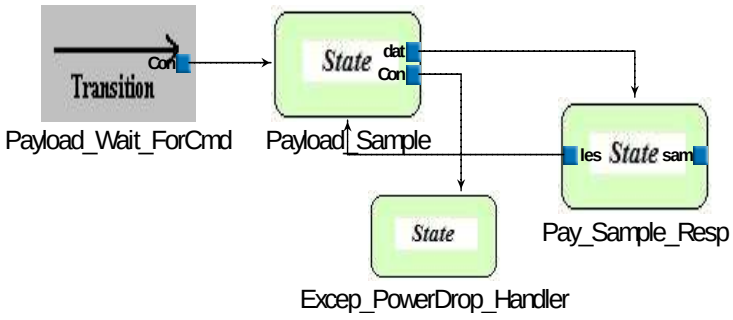

Fig. 7.27: The remaining ControlFlow following Payload_Wait_ForCmd.

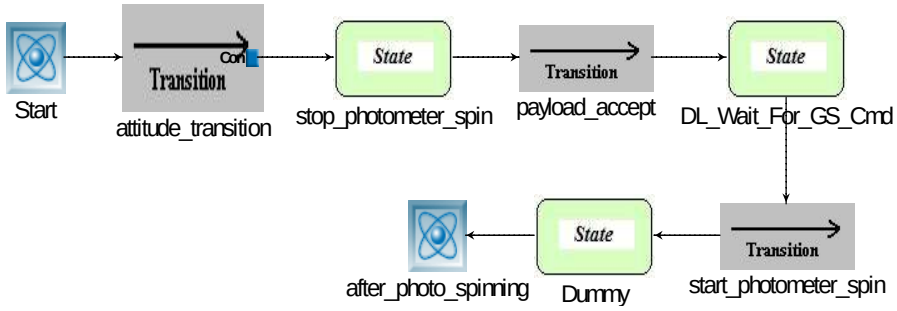

Fig. 7.28: ControlFlow within the Excep_PowerDrop_Handler state.

SModeSystem to the *Wait_For_Spin* mode and necessitates issue of a command to the Payload subsystem in order to stop photometer spin. Transition in the ADCSModeSystem to *Wait_For_Spin* mode causes the spacecraft attitude to change from *Nadir_Pointing* to *Achieving_Nadir*. Hence, *Excep_PowerDrop_Handler* state waits for the ADCS to change attitude to *Achieving_Nadir*, by querying the *AttitudeDataStream*. Once that transition occurs, CDH issues the *Pay_stop_spin* command to the Payload to stop the photometer from spinning. This results in the suspension of data sampling operations. Once the Payload acknowledges the receipt of this command, CDH generates the *ADCSModeStatusMsg*, parameterized with the *Mode_Wait_GS_startCmd* value, and issues it to Downlink. This message informs the ground station that the spacecraft is in a standby mode and operations cannot resume science operations until directed to do so by the ground station. The command to resume the operations is *ADCS_start_spin*, upon which the *start_photometer_spin* ControlTransition is triggered. As a result of the receipt of this command, control returns to the state in which CDH waits for *ADCSModeVal's* transition to *Mode_Spin_Photo* value. This is the state before the *ADCS_Mode_Spin_Photo* transition shown in fig. 7.24.

7.7 TOROID Design Verification

The TOROID-based design discussed till now covered the various subsystems and interactions between them. However, the structure and thermal subysstems have not been modeled. Even for the considered subsystems, only a subset of the operations carried out have been addressed. A comprehensive modeling of the TOROID command set, GSS, VHF

Manager, faults, and VHF Manager operations has not been done. However, the spacecraft model is non-trivial and covers the various power, fault-handling, and operations-based interdependencies. Each subsystem model has certain assumptions about the behavior of other subsystems, with the CDH model capturing intended behavior of the spacecraft as a whole. This design when verified using a model-checking tool, can expose any inconsistent interfaces or unhandled scenarios. The TOROID based design is interpreted into machine-readable CSP by the BASS Interpreter. The generated files are loaded into FDR to check for deadlock, livelock, and the system requirements composed as ModelChecks in BASS.

7.7.1 ADCS Deadlock

The system model was initially checked for deadlock freedom. This check failed in FDR, producing the error trace shown in fig. 7.29. The trace shows the sequence of events performed by CDH which lead to the deadlock. After receiving the message *SystemBus.PowerTlm.PowerDropMsg*, CDH expects a transition in the *AttitudeDataStream* to a non-science attitude. This is indicated by the presence of *SystemBus.ADCSTlm.Attitude_tlm_stream.trans.Achieving_Nadir* in the list of events CDH is ready to accept (shown in fig. 7.30). The trace of events for the ADCS subsystem indicates that the ADCS is not able to exit the *Science_Active* mode after a *PowerDropMsg* is generated. An analysis of the ADCSModeSystem reveals that attitude change due to a drop in power is not communicated to the ADCSModeSystem. Hence, the attitude remains unchanged even after the generation of the *PowerDropMsg*.

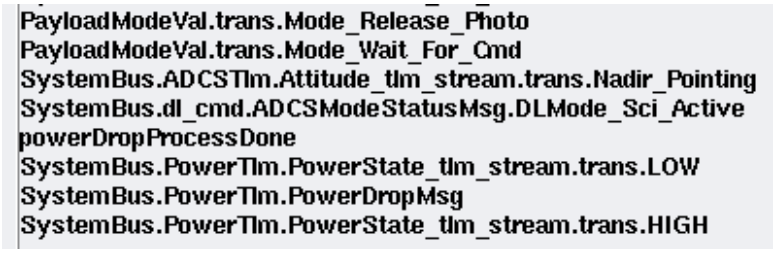

Fig. 7.29: Trace of events CDH performs before system deadlock.

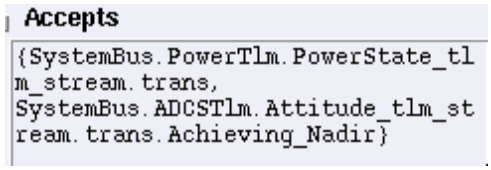

Fig. 7.30: List of events system is ready to accept during the deadlock.

In order to fix this problem with the load-shedding model, a new command *goto_standby* that signals generation of *PowerDropMsg* to the ADCS is added to the command interface of ADCS. An equivalent of this command is added to the CDH CommandSet and a mapping between the two is created in CDHCmdDispatch. The ControlTransition *goto_standby* shown in fig. 7.31 is added from the *Science_Active* mode that has the *goto_standby* command as the trigger event.

The addition of this command necessitates its trigger in the CDH ControlFlow when a *PowerDropMsg* is observed. Consequently, an additional state following the *PowerDropMsg* is inserted in order to issue the CDH equivalent of the *goto_standby* command as shown in fig. 7.32.

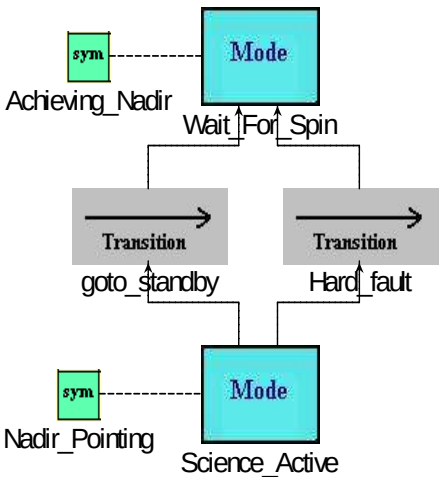

Fig. 7.31: Additional transition goto_standby added to ADCSModeSystem.

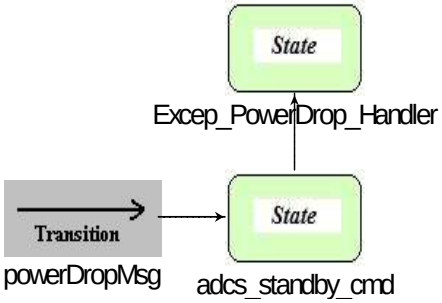

Fig. 7.32: Additional state adcs_standby_cmd added after the PowerDropMsg.

7.7.2 Payload Deadlock

The deadlock check fails even after the above modifications, revealing a problem in the PayloadModeSystem during exception handling. CDH issues a *stop_spin* command in order to suspend data collection. At this time, it is possible for the Payload to be in *Wait_For_Cmd* or *SetThreshold* states in addition to the *Collect_Data* mode (figs. 7.9 and 7.12). Since the *stop_spin* command is not handled in both these modes, its issue results in a deadlock. To fix this problem, transitions which are triggered by the *stop_spin* command are added as shown in fig. 7.33.

With the above modifications, the TOROID design model passes the deadlock-freedom tests. This check causes FDR to consider 254,817 states with 1,135,153 transitions in 2 seconds on a machine using an AMD Athlon X2 Dual Core Processor operating at 2.806Mhz with 2GB RAM. The design is livelock free as well, implying that the system does not perform the same internal event or sequence of internal events forever. The check to ensure power availability, modeled as a built-in check (sec. 6.3.1), succeeds as well after refine checking 254,817 states with 1,135,153 transitions in 1 second. In addition to deadlock and livelock freedom, and the power availability check, other more system-specific checks are derived from the system requirements, and are composed as part of the ModelChecksLib.

The first ModelCheck ensures that the spacecraft eventually reaches the *Sci_Active* mode. When in this mode, if the battery charge falls due to longer eclipse durations, the spacecraft needs to move into safe mode and wait for a ground station command before

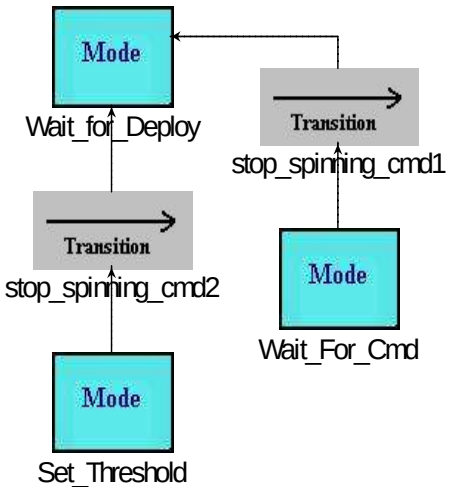

Fig. 7.33: Additional transitions stop_spinning_cmds added to the PayloadModeSystem.

resuming science operations. The fall in power is indicated by changing the SolarFlux value from HIGH to LOW. *ADCS_start_spin* is the command needed by CDH in order to restart its science operations and hence needs to be issued from the ground station. Subsequently, ADCS should reenter the *Science_Active* mode. The desired sequence of events for this scenario is shown in fig. 7.34. This model check is translated into a CSP refinement check over the Failures and Failures/Divergence model. It passes the Failures model successfully after refine checking 78,051 states with 281,538 transitions in 0 seconds, but fails over the Failures/Divergence model. The debug trace points out that the system has an ability to perform an infinite sequence of start and stop sampling operations in the Payload. As this scenario is intended, failure of this check does not necessitate any changes in spacecraft design.

The second ModelCheck is identical to the previous one, replacing the PowerDropMsg with *HW_Fault*. This verifies that the system stops all sampling and communicates this behavioral change to the ground station. Subsequent issue of the command *ADCS_start_spin* should be able to restore the ADCS mode. This is represented in the ModelCheck shown in figs. 7.35 and 7.36. Similar to the earlier check, this succeeds in the Failures model after refine checking 50,922 states with 175,635 transitions in 0 seconds but fails in the

Failures/Divergence model (again, for acceptable reasons).

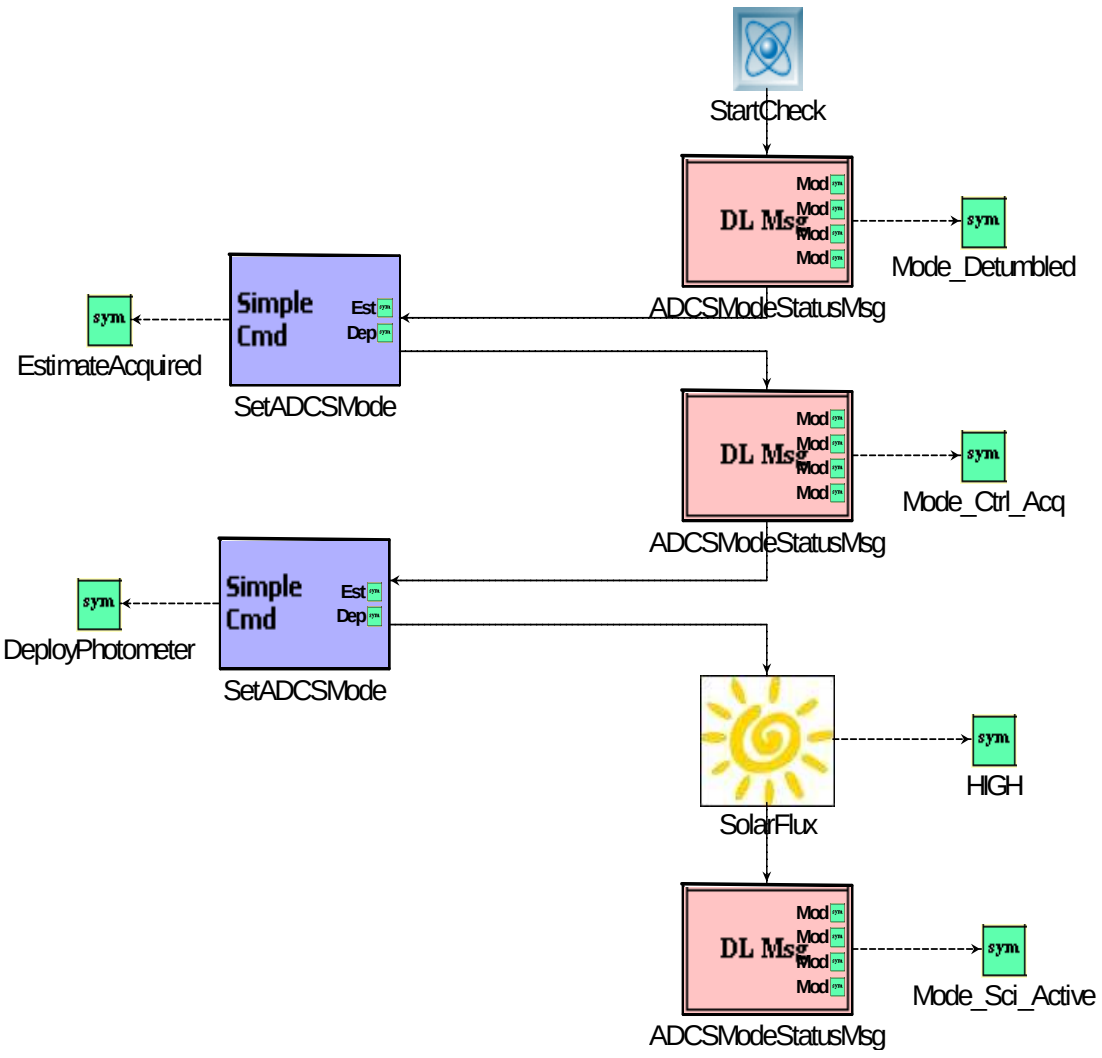

Fig. 7.34: ModelCheck that verifies the desired behavior in the event of drop in power.

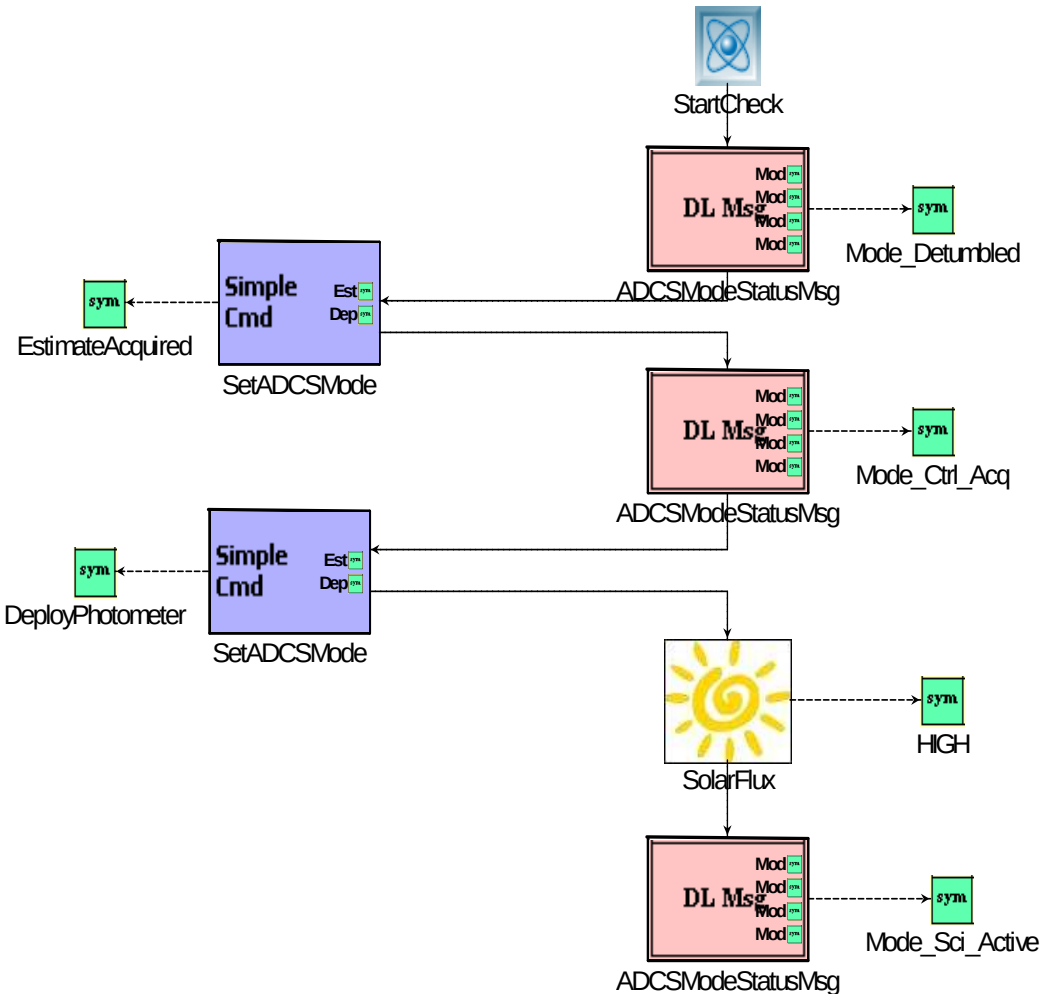

Fig. 7.35: First part of ModelCheck that indicates desired behavior in the event of ADCS hardware fault.

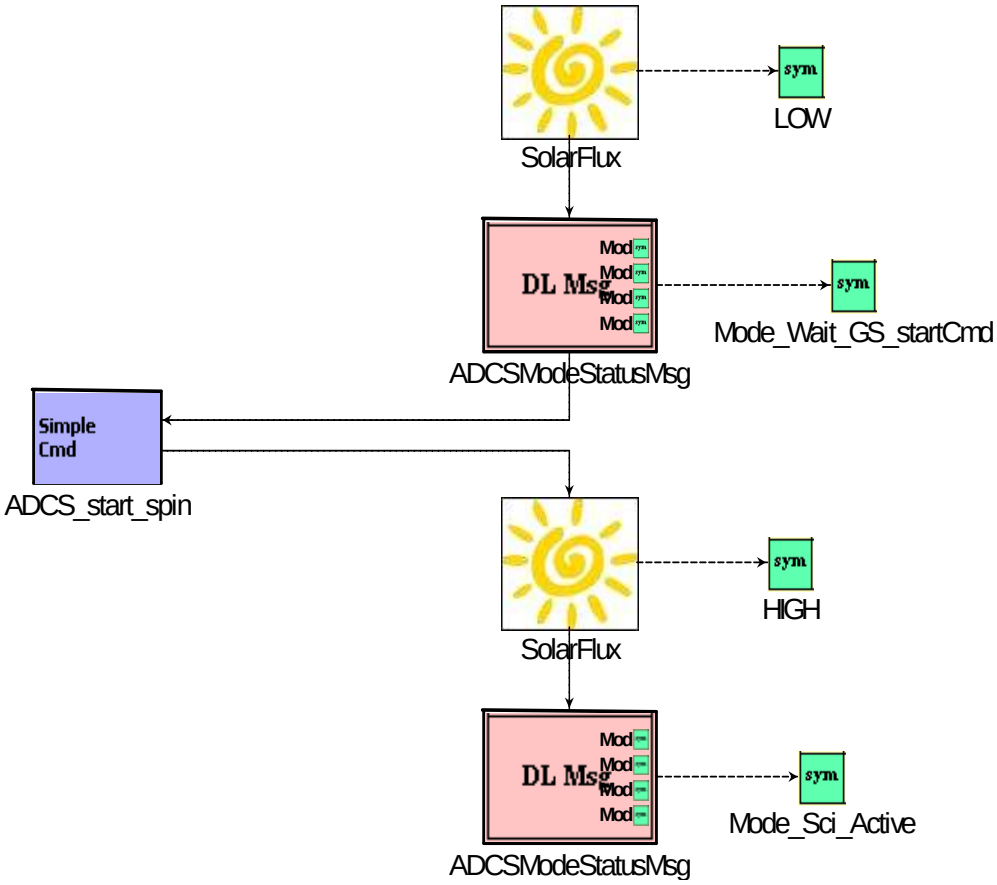

Fig. 7.36: Second part of ModelCheck that indicates desired behavior in the event of ADCS hardware fault.

Chapter 8

Conclusions

The tool developed in this thesis integrates the two distinct areas of domain-specific modeling and spacecraft behavior verification to provide a prototype model-based spacecraft behavior verification tool, with a good measure of success. The capabilities of BASS in creating a visual modeling environment for spacecraft designers and subsequent verification of the developed models for desired properties is amply demonstrated throughout this thesis. It proves that tools modeled on similar lines can be utilized in the initial design phases of small-satellites giving important feedback for already designed modules and guidelines for next implementation phases. By reducing the system-level inconsistencies and revealing unhandled scenarios early, they can prove indispensable to the spacecraft community. Significant tasks accomplished in the process of developing BASS, which advocate more research in this area, are summarized below:

- Weaving in the design methods and terminology used by the spacecraft community into the user-interface of a graphical modeling tool preventing in-depth knowledge of software engineering concepts on the part of spacecraft designers.

- Embedding some implicit structure and behavior of spacecraft at the meta-model level itself, thereby not requiring users to model details from scratch.

- Deriving the power of formal verification without any knowledge of formal semantics chosen for this purpose. Automatic generation of semantically equivalent models can save significant cost and time in the small-satellite design phase, compared to the traditional formal verification approach.

- Describing the black box specifications/system requirements with significant ease by re-using the constructs utilized in developing the spacecraft design.

- Identifying problems in the design if any, rectifying and verifying the changes iteratively till the requirements are satisfied, in very less time.

- Proving the practical usability of the tool by expressing parts of a real-world spacecraft design in BASS and successfully checking for a few system-level properties.

8.1 Areas for Improvement

BASS currently does not support behavioral modeling of a few subsystems like Structure and Thermal. Including them into BASSMP may not necessitate considerable effort since the already developed modeling constructs may suffice to define them. The presently supported formal semantics CSP may not be suitable for analysis when behaviors heavily dependent on timing need to be modeled and analysed for correctness. In addition, there are a few limitations imposed by the use of FDR in the present working BASS toolflow. Firstly, FDR runs on Linux, and therefore the CSP script files generated from GME and BASSI need to be transferred into another operating system. Secondly, the event trace obtained from FDR when the requirement checks fail in FDR is in terms CSP events and cannot be traced back to the user model. Hence, identifying and fixing any issues in the BASS model mandates an understanding of the correlation between the CSP events and the modeling objects used. This can be a difficult task and cannot be automated with current choice of semantics and modeling tool since the tool flow is spread over different operating systems. This shortcoming can be overcome only by using formal semantics other than CSP in the toolflow.

8.2 Future Work

The work demonstrated in this thesis opens up new avenues for further research in the areas of spacecraft behavior verification as mentioned below:

- Various other formal semantics can be coupled to the BASSMP to explore various aspects of spacecraft design. By modifying the BASSMP slightly, and creating appropriate interpreter, timed/resource-oriented semantic models of spacecraft can be

generated. PROMELA/SPIN [5], due to their direct support for modeling resources, is a suitable candidate for modeling resources.

- Requirements can be modeled using more sophisticated tools enabling the expression of more complicated requirements. Integration efforts with prototype requirements capture tools like SDW [9] which can subsequently be verified will be beneficial.

- Current work treats the structural/subsystem analyses and the behavioral analyses as separate. Interfacing behavioral analysis with existing tools that target subsystem analysis like MATLAB can make the spacecraft design analysis complete.

References

[1] C. D. Brown, *Elements of Spacecraft Design*. Reston, VA: American Institute of Aeronautics and Astronautics Education Series, 2002.

[2] V. L. Pisacane, *Fundamentals of Space Systems*. New York: Oxford University Press, 2005.

[3] NASA Langley formal methods site, [http://shemesh.larc.nasa.gov/fm/fm-what.html], 2008.

[4] E. C. Jeannette, E. M. Clarke, J. M. Wing, and E. Al, "Formal methods: State of the art and future directions," *Association for Computing Machinery Computing Surveys*, vol. 28, pp. 626–643, 1996.

[5] "Process or protocol meta language," [http://spinroot.com/spin/Man/Quick.html], 1997.

[6] "Petrinets," [http://www.petrinets.info/], 2009.

[7] E. Borger and R. F. Stark, *Abstract State Machines: A Method for High-Level System Design and Analysis*. Secaucus, NJ: Springer-Verlag, 2003.

[8] A. I. S. McInnes, *A Formal Approach To Specifying and Verifying Spacecraft Behavior*. Ph.D. dissertation, Utah State University, Logan, 2007.

[9] B. K. Eames, A. I. McInnes, J. E. Crace, and J. M. Graham, "A model-based design tool for systems-level spacecraft design," in *Proceedings of the 20th Annual American Institute of Aeronautics and Astronautics/USU Conference on Small Satellites*, 2006.

[10] P. Gluck and G. Holzmann, "Using spin model checking for flight software verification," in *Aerospace Conference Proceedings, IEEE*, vol. 1, pp. 1–105–1–113, 2002.

[11] C. Pecheur and R. G. Simmons, "From Livingstone to Symbolic Model Verifier (SMV) formal verification for autonomous spacecrafts," in *Formal Approaches to Agent-Based Systems*, pp. 103–113, 2000.

[12] S. Easterbrook, R. Lutz, R. Covington, J. Kelly, Y. Ampo, and D. Hamilton, "Experiences using lightweight formal methods for requirements modeling," *IEEE Transactions on Software Engineering*, vol. 24, no. 1, pp. 4–14, 1998.

[13] M. H. Smith, G. C. Cucullu, G. J. Holzmann, and B. D. Smith, "Using Spin model checking for flight software verification," in *Aerospace Conference Proceedings, IEEE*, Mar. 2005.

[14] G. Hilderink, "Graphical modelling language for specifying concurrency based on Communicating Sequential Processes," *IEEE Software Proceedings*, vol. 150, no. 2, pp. 108–120, Apr. 2003.

[15] M. Y. Ng and M. Butler, "Towards formalizing Unified Modeling Language (UML) state diagrams in Communicating Sequential Processes (CSP)," in *Software Engineering and Formal Methods, International Conference*, p. 138, 2003.

[16] L. K. Dillon, G. Kutty, L. E. Moser, P. M. Melliar-Smith, and Y. S. Ramakrishna, "A graphical interval logic for specifying concurrent systems," *Association for Computing Machinery Transactions on Software Engineering and Methodology*, vol. 3, pp. 131–165, 1994.

[17] A. Rockstrom and R. Saracco, "Specification and Description Language (SDL)–International Telegraph and Telephone Consultative Committee," *IEEE Transactions on Communications*, vol. 30, no. 6, pp. 1310–1318, June 1982.

[18] S. Ayache, E. Conquet, P. Humbert, C. Rodriguez, J. Sifakis, and R. Gerlich, "Formal methods for the validation of fault tolerance in autonomous spacecraft," in *Proceedings of the Twenty-Sixth Annual International Symposium on Fault-Tolerant Computing (FTCS '96)*, p. 353, 1996.

[19] Vanderbilt University, "GME User's Manual," 2004.

[20] C. A. R. Hoare, "Communicating sequential processes," *Communications of the Association for Computing Machinery*, vol. 21, no. 8, pp. 666–677, 1978.

[21] S. Schneider, *Concurrent and Real Time Systems: The Communicating Sequential Processes Approach.* New York: John Wiley & Sons, 1999.

[22] A. W. Roscoe, *The theory and practice of concurrency.* Upper Saddle River, NJ: Prentice Hall, 1998 [Online]. Available: `[http://web.comlab.ox.ac.uk/oucl/work/bill.roscoe/publications/68b.pdf]`.

[23] D. W. Oliver, T. P. Kelliher, and J. G. Keegan, *Engineering Complex Systems with Models and Objects.* Columbus, OH: McGraw-Hill, 1997.

[24] W. J. Larson and J. R. Wertz, Eds., *Space Mission Analysis and Design.* Bloomington, IN: Microcosm Press, 1992.

[25] P. Fortescue, J. Stark, and G. Swinerd, Eds., *Modern Formal Methods and Applications.* Chichester, West Sussex: John Wiley & Sons, 2003.

[26] D. Harel and A. Naamad, "The statemate semantics of statecharts," *Association for Computing Machinery Transactions on Software Engineering and Methodology*, vol. 5, no. 4, pp. 293–333, 1996.

[27] A. W. Roscoe and Z. Wu, "Verifying statemate statecharts using Communicating Sequential Processes (CSP) and Failures Divergences Refinement (FDR)," in *Proceedings of International Conference on Formal Engineering Methods*, 2006 [Online]. Available: `[http://web.comlab.ox.ac.uk/oucl/work/bill.roscoe/publications/115.pdf]`.

[28] A. Roscoe, "Modelling and verifying key-exchange protocols using Communicating Sequential Processes (CSP) and Failures Divergences Refinement (FDR)," *Computer Security Foundations Workshop, IEEE*, p. 98, 1995.

[29] G. Lowe and B. Roscoe, "Using Communicating Sequential Processes to detect errors in the TMN protocol," *IEEE Transactions on Software Engineering*, vol. 23, no. 10, pp. 659–669, 1997.

[30] M. B. Josephs, "Gate-level modelling and verification of asynchronous circuits using CSPM and Failures Divergences and Refinement (FDR)," in *Proceedings of the 13th IEEE International Symposium on Asynchronous Circuits and Systems*, pp. 83–94, 2007.

[31] M. Goldsmith and I. Zakiuddin, "Critical systems validation and verification with csp and fdr," in *FM-Trends 98: Proceedings of the International Workshop on Current Trends in Applied Formal Method*, pp. 243–250, 1999.

[32] S. Forbes and G. Hunyadi, "The university nanosat program from concept to flight: A dual student program perspective on what works and what does not," in *Proceedings of the 20th Annual AIAA/USU Conference on Small Satellites*, 2006.

[33] "Utah state university small satellite program," [http://ususat.usu.edu/currentprojects.htm], 2005.

[34] J. E. Crace, "Toroid software subsystem," Master's thesis, Utah State University, Logan, 2007.

[35] B. Srinivisan, "Power-aware mode management software for a nanosatellite," Master's thesis, Utah State University, Logan, 2004.

Appendix

Appendix

CSP Generated by BASSI for TOROID Model of Chapter 7

This appendix contains the CSP files generated by the BASSI for each subsystem of the TOROID model discussed in Chapter 7. Each of the subsystem models is interpreted by BASSI into a separate CSP file, with the system-level CSP in the System.csp file. The CSP for the ModelChecks and the refinement checks is also included in the same file.

ADCS.csp

```
datatype AttitudeValues = Uncontrolled | Nadir_Pointing | Achieving_Nadir
channel Attitude: StateIF.AttitudeValues
channel ADCSModeSystemChannel: ModeTransDelimiter.ADCSModeCmdType

datatype ADCSModeCmdType = Unpowered | Tumbling | Detumble | Detumbled |
                           EstimateAcquired | Science_Active | Science_Idle |
                           SpinUpPhotometer | Photometer_Deployed |
                           DeployPhotometer | WaitFor_GS_Cmd | ControlAcquired
                           | Wait_For_Spin

nametype CommandableADCSModeCmd =
diff(ADCSModeCmdType,{Unpowered,Tumbling,Detumble,Detumbled,Science_Active,
Science_Idle,SpinUpPhotometer,Photometer_Deployed,WaitFor_GS_Cmd,
ControlAcquired,Wait_For_Spin})

datatype ADCSCmd = ADCSModeCmd.CommandableADCSModeCmd | ADCSstartSpinCmd |
                   ADCSgoto_standby
datatype ADCSInternalFaults = ADCS_HW_Fault
```

```
datatype ADCS_Tlm = ADCSModeCmd_Accept.ADCSModeCmdType | ADCSPowerInsufficientMsg.
                    ADCSModeCmdType | ADCSModeCmdRej.ADCSModeCmdType |
                    ADCS_Hardware_Exception | Attitude_tlm_stream.StateIF.AttitudeTlm

channel CDHAttitudeTlmGet:AttitudeTlm

AllowedADCSModeSystem(Detumbled) = {EstimateAcquired}
AllowedADCSModeSystem(WaitFor_GS_Cmd) = {DeployPhotometer}
DisAllowedADCSModeSystem(x) = diff(CommandableADCSModeCmd, AllowedADCSModeSystem(x))

ADCSModePower = SubsysModePower(Unpowered, ADCSModeSystemChannel,
                power.load_delta.ADCS, ADCSModePowerFn)

ADCSModePowerFn(c) =
let
f = {(Detumble, 4),
   (Detumbled, 4),
   (SpinUpPhotometer, 6),
   (ControlAcquired, 6),
   (WaitFor_GS_Cmd, 6),
   (Unpowered, 0),
   (Science_Idle, 6),
   (Photometer_Deployed, 5),
   (Tumbling, 1),
   (Science_Active, 6),
   (DeployPhotometer, 6),
   (EstimateAcquired, 5),
   (Wait_For_Spin, 3)}
within apply(f,c)

ADCSModePowerLevelFn(c) =
```

```
let
f = {(Unpowered, LOW),
   (Tumbling, LOW),
   (Detumble, LOW),
   (Detumbled, LOW),
   (EstimateAcquired, LOW),
   (Science_Active, HIGH),
   (Science_Idle, LOW),
   (SpinUpPhotometer, LOW),
   (Photometer_Deployed, LOW),
   (DeployPhotometer, LOW),
   (WaitFor_GS_Cmd, LOW),
   (ControlAcquired, LOW),
   (Wait_For_Spin, LOW)}
within apply(f,c)

ADCSModeSystemMapFn(c) =
let
f = {(Unpowered, Uncontrolled),
   (Science_Active, Nadir_Pointing),
   (SpinUpPhotometer, Achieving_Nadir),
   (EstimateAcquired, Achieving_Nadir),
   (Tumbling, Uncontrolled),
   (Detumble, Uncontrolled),
   (Detumbled, Achieving_Nadir),
   (DeployPhotometer, Achieving_Nadir),
   (WaitFor_GS_Cmd, Achieving_Nadir),
   (ControlAcquired, Achieving_Nadir),
   (Photometer_Deployed, Achieving_Nadir),
   (Science_Idle, Achieving_Nadir),
   (Wait_For_Spin, Achieving_Nadir)}
within apply(f,c)
```

```
AttitudeState(init) = AssignableState(Attitude.setval, Attitude.getval,
                      Attitude.trans, init)

PreADCSModeSystemState(next, curr, SendRsp) =
ADCSPowerStateTlmGet?x:{d|(d,r) <- PowerStateEnum} -> if fPowerStateEnumFn(x) >=
fPowerStateEnumFn(ADCSModePowerLevelFn(next)) then powerChange.ADCS ->
ADCSAcceptProcess(next,curr,SendRsp) else powerNoChange.ADCS ->
ADCSRejectProcess(next,curr,SendRsp)

ADCSAcceptProcess(next,curr,SendRsp) =
if (SendRsp == true) then SystemBus.ADCSTlm.ADCSModeCmd_Accept!next ->
ADCSModeSystemChannel.begin!next -> Attitude.setval!ADCSModeSystemMapFn(next)
-> ADCSModeSystemChannel.end!next -> ADCSModeSpecificSetVars(next);
ADCSModeSystem(next) else ADCSModeSystemChannel.begin!next -> Attitude.setval!
ADCSModeSystemMapFn(next) -> ADCSModeSystemChannel.end!next ->
ADCSModeSpecificSetVars(next);ADCSModeSystem(next)

ADCSRejectProcess(next,curr,SendRsp) =
if (SendRsp == true) then SystemBus.ADCSTlm.ADCSPowerInsufficientMsg!next ->
ADCSModeSystem(curr) else ADCSModeSystem(curr)

ADCSModeSystem(Unpowered) =
(power.load_switch.ADCS.on -> PreADCSModeSystemState(Tumbling,Unpowered,false))
[]
(power.load_switch.ADCS.off -> powerChange.ADCS -> ADCSModeSystemChannel.
begin!Unpowered -> Attitude.setval!ADCSModeSystemMapFn(Unpowered) ->
ADCSModeSystemChannel.end!Unpowered -> ADCSModeSystem(Unpowered))

ADCSModeSystem(Tumbling) =
(internalEvent.ADCS ->PreADCSModeSystemState(Detumble,Tumbling,false))
[]
```

```
(power.load_switch.ADCS.off -> powerChange.ADCS -> ADCSModeSystemChannel.
begin!Unpowered -> Attitude.setval!ADCSModeSystemMapFn(Unpowered) ->
ADCSModeSystemChannel.end!Unpowered -> ADCSModeSystem(Unpowered))

ADCSModeSystem(Detumble) =
(internalEvent.ADCS ->PreADCSModeSystemState(Detumbled,Detumble,false))
[]
(power.load_switch.ADCS.off -> powerChange.ADCS -> ADCSModeSystemChannel.
begin!Unpowered -> Attitude.setval!ADCSModeSystemMapFn(Unpowered) ->
ADCSModeSystemChannel.end!Unpowered -> ADCSModeSystem(Unpowered))

ADCSModeSystem(Detumbled) =
(SystemBus.allADCSCmds.ADCSModeCmd?m1: AllowedADCSModeSystem(Detumbled) ->
PreADCSModeSystemState(m1,Detumbled,true))
[]
(SystemBus.allADCSCmds.ADCSModeCmd?m1: DisAllowedADCSModeSystem(Detumbled)
-> SystemBus.ADCSTlm.ADCSModeCmdRej!m1 -> powerNoChange.ADCS ->
ADCSModeSystem(Detumbled))
[]
(power.load_switch.ADCS.off -> powerChange.ADCS -> ADCSModeSystemChannel.begin
!Unpowered -> Attitude.setval!ADCSModeSystemMapFn(Unpowered) ->
ADCSModeSystemChannel.end!Unpowered -> ADCSModeSystem(Unpowered))

ADCSModeSystem(EstimateAcquired) =
(internalEvent.ADCS -> PreADCSModeSystemState(ControlAcquired,EstimateAcquired,false))
[]
(power.load_switch.ADCS.off -> powerChange.ADCS -> ADCSModeSystemChannel.begin!
Unpowered -> Attitude.setval!ADCSModeSystemMapFn(Unpowered) -> ADCSModeSystemChannel.
end!Unpowered -> ADCSModeSystem(Unpowered))

ADCSModeSystem(Science_Active) =
(SystemBus.ADCSTlm.ADCS_Hardware_Exception -> PreADCSModeSystemState
```

```
(Wait_For_Spin,Science_Active,false))
[]
(power.load_switch.ADCS.off -> powerChange.ADCS -> ADCSModeSystemChannel.begin!
Unpowered -> Attitude.setval!ADCSModeSystemMapFn(Unpowered) -> ADCSModeSystemChannel
.end!Unpowered -> ADCSModeSystem(Unpowered))
[]
(SystemBus.allADCSCmds.ADCSgoto_standby -> PreADCSModeSystemState(Wait_For_Spin,
Science_Active,false))

ADCSModeSystem(Science_Idle) =
(internalEvent.ADCS ->PreADCSModeSystemState(SpinUpPhotometer,Science_Idle,false))
[]
(power.load_switch.ADCS.off -> powerChange.ADCS -> ADCSModeSystemChannel.begin!
Unpowered -> Attitude.setval!ADCSModeSystemMapFn(Unpowered) ->
ADCSModeSystemChannel.end!Unpowered -> ADCSModeSystem(Unpowered))

ADCSModeSystem(SpinUpPhotometer)
= (internalEvent.ADCS ->PreADCSModeSystemState(EstimateAcquired,SpinUpPhotometer,false))
[]
(power.load_switch.ADCS.off -> powerChange.ADCS -> ADCSModeSystemChannel.
begin!Unpowered -> Attitude.setval!ADCSModeSystemMapFn(Unpowered) ->
ADCSModeSystemChannel.end!Unpowered -> ADCSModeSystem(Unpowered))

ADCSModeSystem(Photometer_Deployed) =
(ADCSPhotometer_EventsMemoryGet?x:{Photo_Spin} ->PreADCSModeSystemState
(Science_Active,Photometer_Deployed,false))
[]
(ADCSPhotometer_EventsMemoryGet?x:diff(Photometer_EventsDType, {Photo_Spin}) ->
PreADCSModeSystemState(Science_Idle,Photometer_Deployed,false))
[]
(power.load_switch.ADCS.off -> powerChange.ADCS -> ADCSModeSystemChannel.
begin!Unpowered -> Attitude.setval!ADCSModeSystemMapFn(Unpowered) ->
```

```
ADCSModeSystemChannel.end!Unpowered -> ADCSModeSystem(Unpowered))

ADCSModeSystem(DeployPhotometer) =
(internalEvent.ADCS ->PreADCSModeSystemState(Photometer_Deployed,DeployPhotometer,false))
[]
(power.load_switch.ADCS.off -> powerChange.ADCS -> ADCSModeSystemChannel.
begin!Unpowered -> Attitude.setval!ADCSModeSystemMapFn(Unpowered) ->
ADCSModeSystemChannel.end!Unpowered -> ADCSModeSystem(Unpowered))

ADCSModeSystem(WaitFor_GS_Cmd) =
(SystemBus.allADCSCmds.ADCSModeCmd?m1: AllowedADCSModeSystem(WaitFor_GS_Cmd) ->
PreADCSModeSystemState(m1,WaitFor_GS_Cmd,true))
[]
(SystemBus.allADCSCmds.ADCSModeCmd?m1: DisAllowedADCSModeSystem(WaitFor_GS_Cmd)
-> SystemBus.ADCSTlm.ADCSModeCmdRej!m1 -> powerNoChange.ADCS ->
ADCSModeSystem(WaitFor_GS_Cmd))
[]
(power.load_switch.ADCS.off -> powerChange.ADCS -> ADCSModeSystemChannel.begin!Unpowered
-> Attitude.setval!ADCSModeSystemMapFn(Unpowered) -> ADCSModeSystemChannel.end!Unpowered
-> ADCSModeSystem(Unpowered))

ADCSModeSystem(ControlAcquired) =
(ADCSPhotometer_EventsMemoryGet?x:{Photo_Deploy,Photo_Spin} ->
PreADCSModeSystemState(Photometer_Deployed,ControlAcquired,false))
[]
(ADCSPhotometer_EventsMemoryGet?x:diff(Photometer_EventsDType,{Photo_Deploy,Photo_Spin})
-> PreADCSModeSystemState(WaitFor_GS_Cmd,ControlAcquired,false))
[]
(power.load_switch.ADCS.off -> powerChange.ADCS -> ADCSModeSystemChannel.begin!Unpowered
-> Attitude.setval!ADCSModeSystemMapFn(Unpowered) -> ADCSModeSystemChannel.end!Unpowered
-> ADCSModeSystem(Unpowered))
```

```
ADCSModeSystem(Wait_For_Spin) =
(SystemBus.allADCSCmds.ADCSstartSpinCmd -> PreADCSModeSystemState
(SpinUpPhotometer,Wait_For_Spin,false))
[]
(power.load_switch.ADCS.off -> powerChange.ADCS -> ADCSModeSystemChannel.begin!
Unpowered -> Attitude.setval!ADCSModeSystemMapFn(Unpowered) ->
ADCSModeSystemChannel.end!Unpowered -> ADCSModeSystem(Unpowered))

ADCSModeSpecificSetVars(Unpowered) = SKIP
ADCSModeSpecificSetVars(Tumbling) = ADCSModeVal.setval!Mode_tumbling -> SKIP
ADCSModeSpecificSetVars(Detumble) = ADCSModeVal.setval!Mode_Detumble -> SKIP
ADCSModeSpecificSetVars(Detumbled) = ADCSModeVal.setval!Mode_Detumbled -> SKIP
ADCSModeSpecificSetVars(EstimateAcquired) = ADCSModeVal.setval!Mode_Estimate_Acq -> SKIP
ADCSModeSpecificSetVars(Science_Active) = ADCSModeVal.setval!Mode_Sci_Active -> SKIP
ADCSModeSpecificSetVars(Science_Idle) = ADCSModeVal.setval!Mode_Sci_Idle -> SKIP
ADCSModeSpecificSetVars(SpinUpPhotometer) = ADCSModeVal.setval!Mode_Spin_Photo ->
                                            Photometer_Events.setval!Photo_Spin -> SKIP
ADCSModeSpecificSetVars(Photometer_Deployed) = ADCSModeVal.setval!Mode_Deployed_Photo -> SKIP
ADCSModeSpecificSetVars(DeployPhotometer) = ADCSModeVal.setval!Mode_Deploy_Photo ->
                                            Photometer_Events.setval!Photo_Deploy -> SKIP
ADCSModeSpecificSetVars(WaitFor_GS_Cmd) = SKIP
ADCSModeSpecificSetVars(ControlAcquired) = ADCSModeVal.setval!Mode_Ctrl_Acq -> SKIP
ADCSModeSpecificSetVars(Wait_For_Spin) = Photometer_Events.setval!Photo_Deploy
-> ADCSModeVal.setval!Mode_Wait_GS_startCmd -> SKIP

ADCSProcess = ((((((ADCSModeSystem(Unpowered)
[|{|Attitude.setval|}|] AttitudeState(Uncontrolled)) \ {|Attitude.setval|})
|[{|power.load_switch.ADCS,Attitude.getval,Attitude.trans|}]|
StateTelemetryStream(power.load_switch.ADCS,Attitude.getval,Attitude.trans,
SystemBus.ADCSTlm.Attitude_tlm_stream.setval,
SystemBus.ADCSTlm.Attitude_tlm_stream.getval,
SystemBus.ADCSTlm.Attitude_tlm_stream.trans,fID))
```

```
\{|SystemBus.ADCSTlm.Attitude_tlm_stream.setval|})
[|{|ADCSModeSystemChannel.end|}|]ADCSModePower)\{|ADCSModeSystemChannel|})
[[SystemBus.ADCSTlm.Attitude_tlm_stream.getval <- SystemBus.ADCSTlm.
Attitude_tlm_stream.getval,
SystemBus.ADCSTlm.Attitude_tlm_stream.getval <- CDHAttitudeTlmGet]]

ADCS_IF = {|SystemBus.allADCSCmds, power.load_switch.ADCS ,Attitude.trans,
CDHAttitudeTlmGet, ADCSPowerStateTlmGet, SystemBus.ADCSTlm, power.load_delta.ADCS,
powerChange.ADCS, powerNoChange.ADCS, internalEvent.ADCS,
SystemBus.ADCSTlm.ADCS_Hardware_Exception, ADCSPhotometer_EventsMemoryGet,
ADCSModeVal.setval, Photometer_Events.setval|}
```

Payload.csp

```
datatype PayloadType = Unpow | Wait_for_Deploy | Release_Photometer | Wait_For_Cmd
                      | CollectData|Set_Threshold|Spin_Up_Photometer
channel PayloadModeSystemChannel: ModeTransDelimiter.PayloadType

datatype PayloadCmd = PayloadSampleInstrumentCmd | PayloadSetPhotometer_Threshold
                    | PayloadStopSamplingCmd | PayloadStopSpinningCmd

PayloadModePower = SubsysModePower(Unpow,PayloadModeSystemChannel,
                  power.load_delta.Payload,PayloadModePowerFn)

datatype Payload_Tlm = Payload_Cmd_Accept.PayloadType | Payload_Cmd_Reject.
                      PayloadType |PayloadPowerInsufficientMsg.PayloadType

PayloadModePowerFn(c) =
let
f = {(Unpow, 0),
   (Wait_for_Deploy, 1),
   (Wait_For_Cmd, 1),
```

```
   (CollectData, 1),
   (Set_Threshold, 1),
   (Release_Photometer, 1),
   (Spin_Up_Photometer, 1)}
within apply(f,c)

PayloadModePowerLevelFn(c) =
let
f = {(Unpow, LOW),
   (Wait_for_Deploy, LOW),
   (Release_Photometer, LOW),
   (Wait_For_Cmd, LOW),
   (CollectData, HIGH),
   (Set_Threshold, LOW),
   (Spin_Up_Photometer, LOW)}
within apply(f,c)

PrePayloadModeSystemState(next, curr,SendRsp) =
PayloadPowerStateTlmGet?x:{d|(d,r) <- PowerStateEnum} -> if fPowerStateEnumFn(x) >=
fPowerStateEnumFn(PayloadModePowerLevelFn(next)) then powerChange.Payload ->
PayloadAcceptProcess(next,curr,SendRsp) else powerNoChange.Payload ->
PayloadRejectProcess(next,curr,SendRsp)

PayloadAcceptProcess(next,curr,SendRsp) =
if SendRsp == true then SystemBus.PayloadTlm.Payload_Cmd_Accept!next ->
PayloadModeSystemChannel.begin!next -> PayloadModeSystemChannel.end!next ->
PayloadModeSpecificSetVars(next);PayloadModeSystem(next) else
PayloadModeSystemChannel.begin!next -> PayloadModeSystemChannel.end!next ->
PayloadModeSpecificSetVars(next);PayloadModeSystem(next)

PayloadRejectProcess(next,curr,SendRsp) =
if SendRsp == true then SystemBus.PayloadTlm.PayloadPowerInsufficientMsg!next
```

```
-> PayloadModeSystem(curr) else PayloadModeSystem(curr)

PayloadModeSystem(Unpow) =
(power.load_switch.Payload.on -> PrePayloadModeSystemState(Wait_for_Deploy,Unpow,false))
[]
(power.load_switch.Payload.off -> powerChange.Payload -> PayloadModeSystemChannel.
begin!Unpow -> PayloadModeSystemChannel.end!Unpow -> PayloadModeSystem(Unpow))

PayloadModeSystem(Wait_for_Deploy) =
(Photometer_Events.trans?x:{Photo_Deploy} ->PrePayloadModeSystemState
(Release_Photometer,Wait_for_Deploy,false))
[]
(power.load_switch.Payload.off -> powerChange.Payload -> PayloadModeSystemChannel
.begin!Unpow -> PayloadModeSystemChannel.end!Unpow -> PayloadModeSystem(Unpow))

PayloadModeSystem(Release_Photometer) =
(Photometer_Events.trans?x:{Photo_Spin} ->PrePayloadModeSystemState(Spin_Up_Photometer,
Release_Photometer,false))
[]
(power.load_switch.Payload.off -> powerChange.Payload -> PayloadModeSystemChannel.
begin!Unpow -> PayloadModeSystemChannel.end!Unpow -> PayloadModeSystem(Unpow))

PayloadModeSystem(Wait_For_Cmd) =
(SystemBus.allPayloadCmds.PayloadSampleInstrumentCmd -> PrePayloadModeSystemState
(CollectData,Wait_For_Cmd,true))
[]
(SystemBus.allPayloadCmds.PayloadSetPhotometer_Threshold -> PrePayloadModeSystemState
(Set_Threshold,Wait_For_Cmd,true))
[]
(SystemBus.allPayloadCmds.PayloadStopSpinningCmd -> PrePayloadModeSystemState
(Wait_for_Deploy,Wait_For_Cmd,true))
[]
```

```
(power.load_switch.Payload.off -> powerChange.Payload -> PayloadModeSystemChannel.
begin!Unpow -> PayloadModeSystemChannel.end!Unpow -> PayloadModeSystem(Unpow))

PayloadModeSystem(CollectData) = (SystemBus.allPayloadCmds.PayloadStopSamplingCmd ->
PrePayloadModeSystemState(Wait_For_Cmd,CollectData,true))
[]
(SystemBus.allPayloadCmds.PayloadStopSpinningCmd -> PrePayloadModeSystemState
(Wait_for_Deploy,CollectData,true))
[]
(power.load_switch.Payload.off -> powerChange.Payload -> PayloadModeSystemChannel.
begin!Unpow -> PayloadModeSystemChannel.end!Unpow -> PayloadModeSystem(Unpow))

PayloadModeSystem(Set_Threshold) =
(internalEvent.Payload ->PrePayloadModeSystemState(Wait_For_Cmd,Set_Threshold,false))
[]
(SystemBus.allPayloadCmds.PayloadStopSpinningCmd -> PrePayloadModeSystemState
(Wait_for_Deploy,Set_Threshold,true))
[]
(power.load_switch.Payload.off -> powerChange.Payload -> PayloadModeSystemChannel.
begin!Unpow -> PayloadModeSystemChannel.end!Unpow -> PayloadModeSystem(Unpow))

PayloadModeSystem(Spin_Up_Photometer) =
(internalEvent.Payload ->PrePayloadModeSystemState(Wait_For_Cmd,Spin_Up_Photometer,false))
[]
(power.load_switch.Payload.off -> powerChange.Payload -> PayloadModeSystemChannel.
begin!Unpow -> PayloadModeSystemChannel.end!Unpow -> PayloadModeSystem(Unpow))

PayloadModeSpecificSetVars(Unpow) = SKIP
PayloadModeSpecificSetVars(Wait_for_Deploy) = PayloadModeVal.setval!
                                              Mode_Wait_For_Deploy -> SKIP
PayloadModeSpecificSetVars(Release_Photometer) = PayloadModeVal.setval!
                                                 Mode_Release_Photo -> SKIP
```

```
PayloadModeSpecificSetVars(Wait_For_Cmd) = PayloadModeVal.setval!Mode_Wait_For_Cmd -> SKIP
PayloadModeSpecificSetVars(CollectData) = PayloadModeVal.setval!Mode_Collect_Data -> SKIP
PayloadModeSpecificSetVars(Set_Threshold) = PayloadModeVal.setval!Mode_Set_Threshold ->
                                            Photometer_voltage.setval!five -> SKIP
PayloadModeSpecificSetVars(Spin_Up_Photometer) = SKIP

PayloadProcess = ((PayloadModeSystem(Unpow))
                 [|{|PayloadModeSystemChannel.end|}|]
                 PayloadModePower)\{|PayloadModeSystemChannel|}

Payload_IF = {|SystemBus.allPayloadCmds, power.load_switch.Payload,
power.load_delta.Payload,PayloadPowerStateTlmGet, SystemBus.PayloadTlm,
powerChange.Payload, powerNoChange.Payload, Photometer_Events.trans.Photo_Deploy,
Photometer_Events.trans.Photo_Spin, internalEvent.Payload, PayloadModeVal.setval,
Photometer_voltage.setval|}
```

Power.csp

```
nametype CommandablePowerADCSSwitch = OnOff
nametype CommandablePowerCDHSwitch = OnOff
nametype CommandablePowerDLSwitch = OnOff
nametype CommandablePowerULSwitch = OnOff
nametype CommandablePowerPayloadSwitch = OnOff

datatype PowerCmd = PowerADCSSwitch.CommandablePowerADCSSwitch | PowerCDHSwitch.
                    CommandablePowerCDHSwitch | PowerDLSwitch.CommandablePowerDLSwitch |
                    PowerULSwitch.CommandablePowerULSwitch |
                    PowerPayloadSwitch.CommandablePowerPayloadSwitch
nametype PowerRange = { -22..22}

datatype PowerType = load_switch.{Payload,Uplink,Downlink,CDH,ADCS}.OnOff |
                     load_delta.{ADCS,CDH,Uplink,Downlink,Payload}.PowerRange
```

```
datatype PowerStateValues = LOW | HIGH
datatype Power_Tlm = PowerDropMsg | PowerState_tlm_stream.StateIF.PowerStateTlm

channel power : PowerType
channel power_delta : PowerRange
channel powerDropProcessDone
channel power_avail, power_alloc : StateIF.PowerRange
channel eps_exception:ResourceException
channel PowerState: StateIF.PowerStateValues
channel changePower:StateIF.PowerStateValues
channel ADCSPowerStateTlmGet:PowerStateTlm
channel PayloadPowerStateTlmGet:PowerStateTlm

PowerStateEnum = {(LOW, 1), (HIGH, 2)}
fPowerStateEnumFn(c) = apply(PowerStateEnum, c)

available(Uncontrolled) = 15-3
available(Achieving_Nadir) = 22-3
available(Nadir_Pointing) = 20-3

Check(pA, pL) = if(pA < pL) then eps_exception.resource_overflow -> STOP
                else DynamicCapacityCheck

DynamicCapacityCheck = (power_avail.trans?pA ->
                            (power_alloc.getval?pL -> Check(pA,pL)
                            []
                            power_alloc.trans?pL ->Check(pA,pL)))
                       []
                       (power_alloc.trans?pL ->
                            (power_avail.getval?pA -> Check(pA,pL)
                            []
                            power_avail.trans?pA -> Check(pA,pL)))
```

```
AvailablePower(a) =
(Attitude.trans?a' -> power_avail.trans!available(a') -> AvailablePower(a'))
[]
power_avail.getval!available(a) -> AvailablePower(a)

AllocatedPower = QuantResource(power_delta, power_alloc.getval, power_alloc.trans,
                               0,19,0)

PowerSwitchCommandMap =
SystemBus.allPowerCmds.PowerCDHSwitch?x -> power.load_switch.CDH!x ->
PowerSwitchCommandMap
[]
SystemBus.allPowerCmds.PowerADCSSwitch?x -> power.load_switch.ADCS!x ->
PowerSwitchCommandMap
[]
SystemBus.allPowerCmds.PowerULSwitch?x -> power.load_switch.Uplink!x ->
PowerSwitchCommandMap
[]
SystemBus.allPowerCmds.PowerDLSwitch?x -> power.load_switch.Downlink!x ->
PowerSwitchCommandMap
[]
SystemBus.allPowerCmds.PowerPayloadSwitch?x -> power.load_switch.Payload!x ->
PowerSwitchCommandMap

PowerStateState(init) = AssignableState(PowerState.setval, PowerState.getval,
                        PowerState.trans, init)

PowerExceptionCheck =
(((AllocatedPower [|{|power_alloc|}|] DynamicCapacityCheck)
[|{|power_avail|}|] AvailablePower(Uncontrolled))\{|power_avail, power_alloc|})
[[power_delta <- power.load_delta.s | s <- {ADCS, CDH, Uplink, Downlink, Payload}]]
```

```
PowerStateTransitions = PowerState.getval?x ->SystemBus.PowerTlm.PowerState_tlm_stream.
                        trans!x -> PowerStateProcess(LOW)
PowerStateProcess(prev) =
changePower.setval?new:{d|(d,r) <- PowerStateEnum, r == fPowerStateEnumFn(prev)+1 or
r == fPowerStateEnumFn(prev)-1} -> if(fPowerStateEnumFn(prev) > fPowerStateEnumFn(new))
then powerDropProcessDone -> PowerState.setval!new -> SystemBus.PowerTlm.
PowerState_tlm_stream.trans!new -> SystemBus.PowerTlm.PowerDropMsg ->
PowerStateProcess(new) else PowerState.setval!new -> SystemBus.PowerTlm.
PowerState_tlm_stream.trans!new -> PowerStateProcess(new)

PowerStateModel = (PowerStateTransitions [|{|PowerState.setval, PowerState.getval|}|]
                   PowerStateState(LOW))\{|PowerState.setval|}
PowerSubSystem =
let
Processes = {PowerExceptionCheck, PowerSwitchCommandMap, PowerStateModel}
within (|||x:Processes @ x)

PowerInitProc = power.load_switch.Uplink.on -> power.load_switch.Downlink.on ->
                power.load_switch.CDH.on -> power.load_switch.Payload.on -> SKIP

PowerComposite = (PowerInitProc ||| PowerSubSystem)\{|power_delta|}

PowerProcess =
((PowerComposite|[{|power.load_switch.CDH, PowerState.getval, PowerState.trans,
SystemBus.PowerTlm.PowerState_tlm_stream.trans|}]| StateTelemetryStream
(power.load_switch.CDH, PowerState.getval, PowerState.trans,
SystemBus.PowerTlm.PowerState_tlm_stream.setval,
SystemBus.PowerTlm.PowerState_tlm_stream.getval,
SystemBus.PowerTlm.PowerState_tlm_stream.trans,fID))
\{|SystemBus.PowerTlm.PowerState_tlm_stream.setval, PowerState.trans,
PowerState.getval|})
```

```
[[SystemBus.PowerTlm.PowerState_tlm_stream.getval <- SystemBus.PowerTlm.
  PowerState_tlm_stream.getval,
  SystemBus.PowerTlm.PowerState_tlm_stream.getval <- ADCSPowerStateTlmGet,
  SystemBus.PowerTlm.PowerState_tlm_stream.getval <- PayloadPowerStateTlmGet]]

Power_IF =
{|SystemBus.allPowerCmds, power, qr_exception, eps_exception, ADCSPowerStateTlmGet,
PayloadPowerStateTlmGet, SystemBus.PowerTlm, Attitude.trans, changePower,
SystemBus.PowerTlm, powerDropProcessDone|}
```

Uplink.csp

```
datatype UplinkCmd = UplinkSetADCSMode.CommandableUplinkSetADCSMode |
                     UplinkADCS_start_spin
nametype CommandableUplinkSetADCSMode = CommandableADCSModeCmd
channel uplink,cmdin: UplinkCmd

UplinkCmdProcess =
uplink?_ -> UplinkCmdProcess
[]
power.load_switch.Uplink.off -> power.load_delta.Uplink!-2 -> UplinkProcess

UplinkProcess = power.load_switch.Uplink.on -> power.load_delta.Uplink!2 ->
                UplinkCmdProcess

Uplink_IF = {|uplink, power.load_switch.Uplink, power.load_delta.Uplink|}
```

Downlink.csp

```
datatype ADCSModeStatusMsgType = DLMode_Detumbled | DLMode_Ctrl_Acq |
```

```
                                DLMode_Sci_Active | DLMode_Wait_GS_startCmd
datatype DLCmd = ADCSModeStatusMsg.ADCSModeStatusMsgType
channel downlink : DLCmd

DownlinkProcess =
let
DLProcess(DLstate) =
(DLstate == off) & SystemBus.dl_cmd?x -> DLProcess(DLstate)
[]
(DLstate == on) & SystemBus.dl_cmd?x -> downlink!x -> DLProcess(DLstate)
[]
power.load_switch.Downlink.on -> power.load_delta.Downlink!2-> DLProcess(on)
[]
power.load_switch.Downlink.off -> power.load_delta.Downlink!-2-> DLProcess(off)
within
DLProcess(off)

Downlink_IF = {|SystemBus.dl_cmd, downlink, power.load_switch.Downlink,
               power.load_delta.Downlink|}
```

CDH.csp

```
nametype CommandableCDHSwitchADCS = OnOff
nametype CommandableCDHSwitchCDH = OnOff
nametype CommandableCDHSwitchPayload = OnOff
nametype CommandableCDHSwitchDL = OnOff

datatype CDHCmd = CDHSwitchADCS.CommandableCDHSwitchADCS |
                  CDHSwitchCDH.CommandableCDHSwitchCDH | CDHSetThresholdVolt |
                  CDHStopDataCollect | CDHSwitchPayload.CommandableCDHSwitchPayload |
                  CDHSwitchDL.CommandableCDHSwitchDL | CDHPay_stop_spin |
```

CDHADCS_Goto_Standby | CDHStartDataCollect

```
Loaded = {}
ExecutionControl =
let
EC(Loaded) =
SystemBus.allCDHCmds.CDHCommandSeqCmd.load_seq?s -> EC(union(Loaded,{s}))
[]
SystemBus.allCDHCmds.CDHCommandSeqCmd.run_seq?s :Loaded -> seq_exec.s.run -> EC(Loaded)
[]
SystemBus.allCDHCmds.CDHCommandSeqCmd.run_seq?s :diff(allCmdSeqs, Loaded) -> EC(Loaded)
[]
SystemBus.allCDHCmds.CDHCommandSeqCmd.unload_seq?s :Loaded -> seq_exec.s.terminate
-> EC(diff(Loaded, {s}))
[]
SystemBus.allCDHCmds.CDHCommandSeqCmd.unload_seq?s :diff(allCmdSeqs, Loaded) ->
EC(Loaded)
[]
SystemBus.allCDHCmds.CDHCommandSeqCmd.stop_seq?s -> seq_exec.s.terminate -> EC(Loaded)
within
EC(Loaded)

CDHCmdDispatch =
cmdin.UplinkSetADCSMode?x -> SystemBus.allADCSCmds.ADCSModeCmd!x -> CDHCmdDispatch
[]
SystemBus.allCDHCmds.CDHSetThresholdVolt -> SystemBus.allPayloadCmds.
PayloadSetPhotometer_Threshold -> CDHCmdDispatch
[]
SystemBus.allCDHCmds.CDHStopDataCollect -> SystemBus.allPayloadCmds.
PayloadStopSamplingCmd -> CDHCmdDispatch
[]
SystemBus.allCDHCmds.CDHSwitchCDH?x -> SystemBus.allPowerCmds.PowerCDHSwitch!x
```

```
-> CDHCmdDispatch
[]
SystemBus.allCDHCmds.CDHSwitchADCS?x -> SystemBus.allPowerCmds.PowerADCSSwitch!x
-> CDHCmdDispatch
[]
SystemBus.allCDHCmds.CDHSwitchPayload?x -> SystemBus.allPowerCmds.
PowerPayloadSwitch!x -> CDHCmdDispatch
[]
SystemBus.allCDHCmds.CDHSwitchDL?x -> SystemBus.allPowerCmds.PowerDLSwitch!x ->
CDHCmdDispatch
[]
cmdin.UplinkADCS_start_spin -> SystemBus.allADCSCmds.ADCSstartSpinCmd ->
CDHCmdDispatch
[]
SystemBus.allCDHCmds.CDHPay_stop_spin -> SystemBus.allPayloadCmds.
PayloadStopSpinningCmd -> CDHCmdDispatch
[]
SystemBus.allCDHCmds.CDHADCS_Goto_Standby -> SystemBus.allADCSCmds.ADCSgoto_standby
-> CDHCmdDispatch
[]
SystemBus.allCDHCmds.CDHStartDataCollect -> SystemBus.allPayloadCmds.
PayloadSampleInstrumentCmd -> CDHCmdDispatch

--controlFlow of CDH
CDHControlFlowProcess =
power.load_switch.CDH.on -> power.load_delta.CDH!3 ->SystemBus.allCDHCmds.
CDHSwitchADCS!on -> SystemBus.ADCSTlm.Attitude_tlm_stream.trans?x :
{Achieving_Nadir} -> NextControlFlowProcess

NextControlFlowProcess = ControlFlow1Process

ControlFlow1Process = ADCSModeVal.trans?x:{Mode_Detumbled} -> DL_DetumbledProcess
```

```
DL_DetumbledProcess = SystemBus.dl_cmd!ADCSModeStatusMsg.DLMode_Detumbled ->
cmdin.UplinkSetADCSMode!EstimateAcquired -> SystemBus.ADCSTlm.ADCSModeCmd_Accept.
EstimateAcquired -> ADCSModeVal.trans?x:{Mode_Ctrl_Acq} -> DL_Ctrl_AcquiredProcess

DL_Ctrl_AcquiredProcess = SystemBus.dl_cmd!ADCSModeStatusMsg.DLMode_Ctrl_Acq ->
cmdin.UplinkSetADCSMode!DeployPhotometer -> SystemBus.ADCSTlm.ADCSModeCmd_Accept.
DeployPhotometer -> ADCS_Mode_Spin_PhotoProcess

ADCS_Mode_Spin_PhotoProcess = ADCSModeVal.trans?x:{Mode_Spin_Photo} ->
                              ADCS_Mode_Ctrl_AcqProcess
ADCS_Mode_Ctrl_AcqProcess = ADCSModeVal.trans?x:{Mode_Ctrl_Acq} ->
                            Payload_Release_PhotoProcess
Payload_Release_PhotoProcess = PayloadModeVal.trans?x:{Mode_Release_Photo} ->
                               Payload_Wait_For_CmdProcess
Payload_Wait_For_CmdProcess = PayloadModeVal.trans?x:{Mode_Wait_For_Cmd} ->
                              Att_Trans_NadirProcess
Att_Trans_NadirProcess = SystemBus.ADCSTlm.Attitude_tlm_stream.trans?x :
                         {Nadir_Pointing} -> DL_SciActiveProcess
DL_SciActiveProcess = SystemBus.dl_cmd!ADCSModeStatusMsg.DLMode_Sci_Active ->
                      After_Science_AttitudeProcess

After_Science_AttitudeProcess =
SystemBus.allCDHCmds.CDHSetThresholdVolt -> SystemBus.PayloadTlm.Payload_Cmd_Accept.
Set_Threshold -> PayloadModeVal.trans?x:{Mode_Wait_For_Cmd} -> Payload_SampleProcess
[]
SystemBus.ADCSTlm.ADCS_Hardware_Exception -> Excep_PowerDrop_HandlerProcess
[]
powerDropProcessDone -> SystemBus.PowerTlm.PowerDropMsg -> SystemBus.allCDHCmds.
CDHADCS_Goto_Standby -> Excep_PowerDrop_HandlerProcess

Excep_PowerDrop_HandlerProcess = ADCS_Exception_HandlerProcess
```

```
ADCS_Exception_HandlerProcess = SystemBus.ADCSTlm.Attitude_tlm_stream.trans?x :
{Achieving_Nadir} -> stop_photometer_spinProcess

Payload_SampleProcess = collect_data_ctrl_flowProcess
collect_data_ctrl_flowProcess =
SystemBus.allCDHCmds.CDHStartDataCollect -> Pay_Sample_RespProcess
[]
SystemBus.ADCSTlm.ADCS_Hardware_Exception -> Excep_PowerDrop_HandlerProcess
[]
powerDropProcessDone -> SystemBus.PowerTlm.PowerDropMsg -> SystemBus.allCDHCmds.
CDHADCS_Goto_Standby -> Excep_PowerDrop_HandlerProcess

stop_photometer_spinProcess =
SystemBus.allCDHCmds.CDHPay_stop_spin -> SystemBus.PayloadTlm.Payload_Cmd_Accept.
Wait_for_Deploy -> SystemBus.dl_cmd!ADCSModeStatusMsg.DLMode_Wait_GS_startCmd ->
cmdin.UplinkADCS_start_spin -> ADCS_Mode_Spin_PhotoProcess

Pay_Sample_RespProcess = Response_Sample_CmdProcess

Response_Sample_CmdProcess = SystemBus.PayloadTlm.PayloadPowerInsufficientMsg.
CollectData -> Payload_SampleProcess
[]
SystemBus.PayloadTlm.Payload_Cmd_Accept.CollectData -> stop_cmdProcess

stop_cmdProcess = stop_cmd_ctrl_flowProcess

stop_cmd_ctrl_flowProcess =
SystemBus.ADCSTlm.ADCS_Hardware_Exception -> Excep_PowerDrop_HandlerProcess
[]
powerDropProcessDone -> SystemBus.PowerTlm.PowerDropMsg -> SystemBus.allCDHCmds.
CDHADCS_Goto_Standby -> Excep_PowerDrop_HandlerProcess
[]
```

```
SystemBus.allCDHCmds.CDHStopDataCollect -> stop_cmd_ackProcess

stop_cmd_ackProcess = SystemBus.PayloadTlm.Payload_Cmd_Accept.Wait_For_Cmd ->
                      PayloadModeVal.trans?x:{Mode_Wait_For_Cmd} -> Payload_SampleProcess

--End of CDH ControlFlow

PowerStateTransObserve(prev) = (SystemBus.PowerTlm.PowerState_tlm_stream.trans?x
                                -> PowerStateTransObserve(x))
CDHProcess1 = let
Processes = {CDHCmdDispatch, PowerStateTransObserve(LOW)}
within (|||x:Processes @ x)

--Control process
CDHControl = CDHControlFlowProcess

CDHProcess = (CDHProcess1 [|{|SystemBus.allCDHCmds, cmdin|}|] CDHControl)
             [[cmdin <- uplink]]

CDH_IF =
{|power.load_switch.CDH.on, power.load_delta.CDH, SystemBus, uplink, ADCSModeVal.trans.
Mode_Detumbled, ADCSModeVal.trans.Mode_Ctrl_Acq, ADCSModeVal.trans.Mode_Spin_Photo,
PayloadModeVal.trans.Mode_Release_Photo, PayloadModeVal.trans.Mode_Wait_For_Cmd,
powerDropProcessDone, CDHAttitudeTlmGet|}
```

System.csp

```
include "lib.csp"
--lib.csp is the SBFL developed by McInnes
include "ADCS.csp"
include "CDH.csp"
include "Uplink.csp"
```

```
include "Downlink.csp"
include "Payload.csp"
include "Power.csp"

datatype SubSysCmd = allADCSCmds.ADCSCmd | allPayloadCmds.PayloadCmd |
                     allPowerCmds.PowerCmd | dl_cmd.DLCmd| allCDHCmds.CDHCmd
datatype SubSysTlm = ADCSTlm.ADCS_Tlm | PowerTlm.Power_Tlm | PayloadTlm.Payload_Tlm
datatype Subsystem = ADCS | CDH | Uplink | Downlink | Payload | Power

channel SystemBus : union(SubSysCmd, SubSysTlm)
channel internalEvent: {ADCS,Payload}
fID(x) = x
channel success

channel fault:{ADCS}.ADCSInternalFaults

--Nametypes for all the Data Streams
nametype AttitudeTlm = union(AttitudeValues, GenericStreamState)
nametype PowerStateTlm = union(PowerStateValues, GenericStreamState)

--Processes for MemoryUnits
datatype Photometer_voltageDType = five|seven|ten
channel Photometer_voltage: StateIF.Photometer_voltageDType

Photometer_voltageState(init) = AssignableState(Photometer_voltage.setval,
                                Photometer_voltage.getval, Photometer_voltage.trans, init)

datatype ADCSModeValDType =
Mode_Unpow | Mode_tumbling | Mode_Detumble | Mode_Estimate_Acq | Mode_Ctrl_Acq |
Mode_Deploy_Photo | Mode_Deployed_Photo | Mode_Sci_Idle | Mode_Sci_Active |
Mode_Spin_Photo | Mode_Detumbled | Mode_Wait_GS_startCmd
channel ADCSModeVal: StateIF.ADCSModeValDType
```

```
ADCSModeValState(init) = AssignableState(ADCSModeVal.setval, ADCSModeVal.getval,
                         ADCSModeVal.trans, init)

datatype PayloadModeValDType =
Mode_Unpowered | Mode_Wait_For_Deploy | Mode_Release_Photo | Mode_Wait_For_Cmd |
Mode_Collect_Data | Mode_Set_Threshold
channel PayloadModeVal: StateIF.PayloadModeValDType

PayloadModeValState(init) = AssignableState(PayloadModeVal.setval, PayloadModeVal.getval,
                            PayloadModeVal.trans, init)

datatype Photometer_EventsDType = Photo_Spin|Photo_Deploy|Photo_Not_Deployed
channel Photometer_Events: StateIF.Photometer_EventsDType

Photometer_EventsState(init) = AssignableState(Photometer_Events.setval,
                               Photometer_Events.getval, Photometer_Events.trans, init)

channel PayloadPhotometer_voltageMemoryGet : Photometer_voltageDType
channel CDHADCSModeValMemoryGet : ADCSModeValDType
channel CDHPayloadModeValMemoryGet : PayloadModeValDType
channel PayloadPhotometer_EventsMemoryGet : Photometer_EventsDType
channel ADCSPhotometer_EventsMemoryGet : Photometer_EventsDType

HideTransSet = diff({|Photometer_voltage.trans, ADCSModeVal.trans, PayloadModeVal.
trans, Photometer_Events.trans|},{|ADCSModeVal.trans.Mode_Detumbled, ADCSModeVal.
trans.Mode_Ctrl_Acq, ADCSModeVal.trans.Mode_Spin_Photo, ADCSModeVal.trans.
Mode_Ctrl_Acq, PayloadModeVal.trans.Mode_Release_Photo, PayloadModeVal.trans.
Mode_Wait_For_Cmd, PayloadModeVal.trans.Mode_Wait_For_Cmd, PayloadModeVal.trans.
Mode_Wait_For_Cmd, Photometer_Events.trans.Photo_Deploy, Photometer_Events.trans.
Photo_Spin|})
```

```
MemoryUnits1 =
let
MemoryUnitProcesses = {(Photometer_EventsState(Photo_Not_Deployed)),
                       (PayloadModeValState(Mode_Unpowered)),
                       (ADCSModeValState(Mode_Unpow)),
                       (Photometer_voltageState(five))}
within (||| x:MemoryUnitProcesses @ x)

MemoryUnits = (MemoryUnits1\HideTransSet)
[[Photometer_voltage.getval <- Photometer_voltage.getval,
  Photometer_voltage.getval <- PayloadPhotometer_voltageMemoryGet,
  ADCSModeVal.getval <- ADCSModeVal.getval,
  ADCSModeVal.getval <- CDHADCSModeValMemoryGet,
  PayloadModeVal.getval <- PayloadModeVal.getval,
  PayloadModeVal.getval <- CDHPayloadModeValMemoryGet,
  Photometer_Events.getval <- Photometer_Events.getval,
  Photometer_Events.getval <- PayloadPhotometer_EventsMemoryGet,
  Photometer_Events.getval <- ADCSPhotometer_EventsMemoryGet]]

MemoryUnits_IF = union({|ADCSModeVal.trans.Mode_Detumbled, ADCSModeVal.trans.
Mode_Ctrl_Acq, ADCSModeVal.trans.Mode_Spin_Photo, PayloadModeVal.trans.
Mode_Release_Photo, PayloadModeVal.trans.Mode_Wait_For_Cmd, Photometer_Events.
trans.Photo_Deploy, Photometer_Events.trans.Photo_Spin|},{|ADCSPhotometer_EventsMemoryGet,
Photometer_voltage.setval, ADCSModeVal.setval, PayloadModeVal.setval,
Photometer_Events.setval|})

--Processes for PowerDeltaSequencer
nametype powerChangeType = diff(Subsystem, {CDH,Uplink,Downlink,Power})
channel powerChange:powerChangeType
channel powerNoChange:powerChangeType

PowerResponse(m) =  powerChange.m -> power.load_delta.m?y -> PowerDeltaSequencer
```

```
                    []
                    powerNoChange.m -> PowerDeltaSequencer
PowerDeltaSequencer =
power.load_switch.CDH?x -> power.load_delta.CDH?y ->
   PowerDeltaSequencer
[] power.load_switch.Uplink?x -> power.load_delta.Uplink?y -> PowerDeltaSequencer
[] power.load_switch.Downlink?x -> power.load_delta.Downlink?y -> PowerDeltaSequencer
[] power.load_switch.ADCS?x -> powerChange.ADCS -> power.load_delta.ADCS?y ->
   PowerDeltaSequencer
[] SystemBus.allADCSCmds.ADCSModeCmd?x -> PowerResponse(ADCS)
[] SystemBus.ADCSTlm.ADCS_Hardware_Exception -> PowerResponse(ADCS)
[] SystemBus.allADCSCmds.ADCSgoto_standby-> PowerResponse(ADCS)
[] ADCSPhotometer_EventsMemoryGet?x -> PowerResponse(ADCS)
[] SystemBus.allADCSCmds.ADCSstartSpinCmd -> PowerResponse(ADCS)
[] power.load_switch.Payload?x -> powerChange.Payload -> power.load_delta.Payload?y ->
   PowerDeltaSequencer
[] Photometer_Events.trans?x:{Photo_Deploy} -> PowerResponse(Payload)
[] Photometer_Events.trans?x:{Photo_Spin} -> PowerResponse(Payload)
[] SystemBus.allPayloadCmds.PayloadSampleInstrumentCmd -> PowerResponse(Payload)
[] SystemBus.allPayloadCmds.PayloadSetPhotometer_Threshold -> PowerResponse(Payload)
[] SystemBus.allPayloadCmds.PayloadStopSpinningCmd -> PowerResponse(Payload)
[] SystemBus.allPayloadCmds.PayloadStopSamplingCmd -> PowerResponse(Payload)
[] internalEvent?m1 -> PowerResponse(m1)

PowerDeltaSequencer_IF = {|power, powerChange, powerNoChange, SystemBus.allADCSCmds.
ADCSModeCmd, SystemBus.ADCSTlm.ADCS_Hardware_Exception, SystemBus.
allADCSCmds.ADCSgoto_standby, ADCSPhotometer_EventsMemoryGet, SystemBus.allADCSCmds.
ADCSstartSpinCmd, Photometer_Events.trans.Photo_Deploy, Photometer_Events.
trans.Photo_Spin, SystemBus.allPayloadCmds.PayloadSampleInstrumentCmd, SystemBus.
allPayloadCmds.PayloadSetPhotometer_Threshold, SystemBus.allPayloadCmds.
PayloadStopSpinningCmd,SystemBus.allPayloadCmds.PayloadStopSamplingCmd, internalEvent|}
```

```
--Processes representing ModelChecksLib
check2 = SystemBus.PowerTlm.PowerState_tlm_stream.trans.LOW -> Check2Proc1
Check2Proc0 = changePower.setval.HIGH-> SystemBus.PowerTlm.PowerState_tlm_stream.
              trans.HIGH -> Check2Proc8
Check2ORProc0 = SKIP; Check2Proc8

Check2Proc1 = Check2Proc4
Check2ORProc1 = SKIP; Check2Proc4

Check2Proc2 = changePower.setval.LOW-> SystemBus.PowerTlm.PowerState_tlm_stream.
trans.LOW -> Check2Proc9
Check2ORProc2 = SKIP; Check2Proc9

Check2Proc3 = changePower.setval.HIGH-> SystemBus.PowerTlm.PowerState_tlm_stream.
trans.HIGH -> Check2Proc11
Check2ORProc3 = SKIP; Check2Proc11

Check2Proc4 = downlink.ADCSModeStatusMsg.DLMode_Detumbled-> Check2Proc6
Check2ORProc4 = SKIP; Check2Proc6

Check2Proc5 = downlink.ADCSModeStatusMsg.DLMode_Ctrl_Acq-> Check2Proc7
Check2ORProc5 = SKIP; Check2Proc7

Check2Proc6 = uplink.UplinkSetADCSMode.EstimateAcquired-> Check2Proc5
Check2ORProc6 = SKIP; Check2Proc5

Check2Proc7 = uplink.UplinkSetADCSMode.DeployPhotometer-> Check2Proc0
Check2ORProc7 = SKIP; Check2Proc0

Check2Proc8 = downlink.ADCSModeStatusMsg.DLMode_Sci_Active-> Check2Proc2
Check2ORProc8 = SKIP; Check2Proc2
```

```
Check2Proc9 = downlink.ADCSModeStatusMsg.DLMode_Wait_GS_startCmd->
              Check2Proc10
Check2ORProc9 = SKIP; Check2Proc10

Check2Proc10 = uplink.UplinkADCS_start_spin-> Check2Proc3
Check2ORProc10 = SKIP; Check2Proc3

Check2Proc11 = downlink.ADCSModeStatusMsg.DLMode_Sci_Active-> SKIP
Check2ORProc11 = SKIP

check2IF = {|uplink, downlink, changePower.setval,SystemBus.PowerTlm.
PowerState_tlm_stream.trans, SystemBus.ADCSTlm.ADCS_Hardware_Exception|}

check3 = SystemBus.PowerTlm.PowerState_tlm_stream.trans.LOW -> Check3Proc0
Check3Proc0 = Check3Proc10
Check3ORProc0 = SKIP; Check3Proc10

Check3Proc1 = changePower.setval.HIGH-> SystemBus.PowerTlm.
              PowerState_tlm_stream.trans.HIGH -> Check3Proc6
Check3ORProc1 = SKIP; Check3Proc6

Check3Proc2 = SystemBus.ADCSTlm.ADCS_Hardware_Exception-> Check3Proc5
Check3ORProc2 = SKIP; Check3Proc5

Check3Proc3 = downlink.ADCSModeStatusMsg.DLMode_Sci_Active-> SKIP
Check3ORProc3 = SKIP

Check3Proc4 = uplink.UplinkADCS_start_spin-> Check3Proc3
Check3ORProc4 = SKIP; Check3Proc3

Check3Proc5 = downlink.ADCSModeStatusMsg.DLMode_Wait_GS_startCmd-> Check3Proc4
Check3ORProc5 = SKIP; Check3Proc4
```

```
Check3Proc6 = downlink.ADCSModeStatusMsg.DLMode_Sci_Active-> Check3Proc2
Check3ORProc6 = SKIP; Check3Proc2

Check3Proc7 = uplink.UplinkSetADCSMode.DeployPhotometer-> Check3Proc1
Check3ORProc7 = SKIP; Check3Proc1

Check3Proc8 = uplink.UplinkSetADCSMode.EstimateAcquired-> Check3Proc9
Check3ORProc8 = SKIP; Check3Proc9

Check3Proc9 = downlink.ADCSModeStatusMsg.DLMode_Ctrl_Acq-> Check3Proc7
Check3ORProc9 = SKIP; Check3Proc7

Check3Proc10 = downlink.ADCSModeStatusMsg.DLMode_Detumbled-> Check3Proc8
Check3ORProc10 = SKIP; Check3Proc8

check3IF = {|uplink, downlink, changePower.setval,SystemBus.PowerTlm.
PowerState_tlm_stream.trans, SystemBus.ADCSTlm.ADCS_Hardware_Exception|}

check1 = STOP
assert check1 [T= SpacecraftSystem \diff(Events,{|eps_exception,qr_exception|})

--Composite Spacecraft System Definition
SpacecraftSystem =
let
Subsystems = {(ADCS_IF, ADCSProcess),
              (CDH_IF, CDHProcess),
              (Uplink_IF, UplinkProcess),
              (Downlink_IF, DownlinkProcess),
              (Payload_IF, PayloadProcess),
              (Power_IF, PowerProcess),
              (MemoryUnits_IF, MemoryUnits),
```

```
            (PowerDeltaSequencer_IF,PowerDeltaSequencer)}
within (|| (IF, Subsys):Subsystems @ [IF] Subsys)

assert (success -> STOP) [F= ((check2;success -> STOP)
                              [|check2IF|]
                              SpacecraftSystem)\diff(Events,{success})
assert (success -> STOP) [FD= ((check2;success -> STOP)
                              [|check2IF|]
                              SpacecraftSystem)\diff(Events,{success})
assert (success -> STOP) [F= ((check3;success -> STOP)
                              [|check3IF|]
                              SpacecraftSystem)\diff(Events,{success})
assert (success -> STOP) [FD= ((check3;success -> STOP)
                              [|check3IF|]
                              SpacecraftSystem)\diff(Events,{success})
```