A Domain Specific Design Tool for Spacecraft System Behavior

Sravanthi Venigalla, Brandon Eames
Electrical & Computer Engineering
Utah State University, USA
{sravanthi.venigalla@aggiemail|beames@engineering}.usu.edu

Allan McInnes
Electrical & Computer Engineering
University of Canterbury, New Zealand
allan.mcinnes@canterbury.ac.nz

Abstract

Specification of spacecraft subsystem interactions is typically carried out using informal diagrams and descriptions that can obscure subtle ambiguities and inconsistencies. As a result, problems in the way subsystems are designed to interact may remain undetected until the integration and test phase, when the cost of change is high. Our Behavioral Analysis of Spacecraft Systems (BASS) modeling tool provides a structured way to define spacecraft subsystem interfaces and interactions, and access to an underlying formal model of interaction that allows the specified interactions to be rigorously analyzed. The enforced consistency of the diagrams produced by our tool and the analytical power of the underlying formal model increases a developer's ability to discover and correct system design errors early in the development process.

Categories and Subject Descriptors D.2.2 [**Software Engineering**]: Design Tools and Techniques – computer aided software engineering; D.3.2 [**Programming Languages**]: Language Classifications – specialized application languages, D.3.1 [**Programming Languages**]: Formal Definitions and Theory – syntax, semantics.

General Terms Design, Languages, Verification.

Keywords Domain-Specific Language; Formal Verification; Behavior; Spacecraft System Design

1. Introduction

Spacecraft systems design is the domain of systems engineers, who are responsible for the high level design of not only computer software and hardware, but the physical structure, thermal properties, electrical wiring and harnessing, data communications, orbit management, and spacecraft control. A primary challenge in system-level spacecraft design is ensuring that the spacecraft subsystems interact correctly to allow the spacecraft to achieve its mission. For example, when a ground station commands the spacecraft to begin taking science data from an onboard instrument, that command may trigger a sequence of actions in which the Command & Data Handling (CDH) subsystem instructs the Attitude Determination & Control System (ADCS) to reorient the spacecraft to point at an object of scientific interest, commands the Power subsystem to supply power to the Payload, commands the Payload to begin data sampling, and then collects and stores the sampled science data for a later downlink.

Early in the design process, systems engineers typically develop a simple, high-level system block diagram that shows the partitioning of the system into subsystems, and the connectivity between those subsystems. Separate documentation defines the behavior of each spacecraft subsystem using simple state diagrams, tables, and textual descriptions of how the subsystems respond to internal and external events. Spacecraft subsystem design proceeds under the assumption that the combined behaviors specified for the subsystems will lead to the desired system behavior, and that the subsystems can each be designed in relative isolation as long as each subsystem adheres to the interface and behavior specified for it. However, there is a lack of tool support for capturing spacecraft system behavior specifications, and the mixture of informal notations currently used by systems engineers cannot be readily analyzed, making it difficult to check assumptions about the emergent system behavior until substantial resources have been expended on subsystem design.

In this paper, we introduce BASS (Behavior Analysis for Spacecraft Systems), a model based design tool that supports the modeling and verification of system-level spacecraft behavior, through the analysis of a composition of subsystem behavior models. We focus on the BASS domain-specific visual modeling language (Section 3), which is built upon the Generic Modeling Environment (GME) [1],[2]. This language provides spacecraft-specific modeling constructs that permit both system connectivity and subsystem behavior descriptions to be captured in a single hierarchical model, and provides a starting point from which to gather user feedback on domain-specific modeling needs. A key benefit of providing a language and tool support for describing spacecraft system behavior is the ability to automatically map visual models to an underlying mathematical semantics that permits rigorous analysis. Our semantic model is based on the concept of concurrency and process interaction, and is codified using the CSP (Communicating Sequential Processes) process algebra [3]. As part of the BASS project, we have developed a model interpreter tool, which is responsible for translating the visual models of spacecraft behavior into a CSP model that can be verified against a higher-level specification, or checked against user-specified assertions or constraints. In Section 4 we briefly describe the model interpreter, and the overall toolflow within which spacecraft system models can be constructed and verified.

2. Background

Spacecraft Systems

As with most complex systems, spacecraft designs are usually partitioned into functionally distinct subsystems. Although the exact names and functionality of the subsystems vary from organization to organization, unmanned spacecraft are typically divided into some variation on the following subsystems:

- **ADCS**-- Attitude Determination & Control, responsible for determining the direction the spacecraft is pointing, and for adjusting that direction as needed
- **CDH**-- Command & Data Handling, consisting of the main spacecraft computer system. CDH is responsible for manag-

ing spacecraft interactions with the ground station, as well as collecting, logging and transmitting data
- **Communications**—Transmission and reception of commands and data
- **Power**-- Consisting of the power generation (ex. solar panels), storage (batteries) and distribution (wiring) facilities
- **Payload**-- Offering a mission-dependent subsystem, typically involving some science based instrument or communication device
- **Propulsion**-- The facilities to physically alter the spacecraft velocity and/or position
- **Structures & Mechanisms**-- Physical support for the other subsystems, and deployment of booms, antennas, and solar arrays
- **Thermal Control**-- Regulation of the spacecraft thermal state

Both the behavior of individual subsystems and the interfaces between the subsystems are extremely mission-dependent. Some spacecraft omit subsystems that are unnecessary to their particular mission.

The Generic Modeling Environment

GME is a tool developed at Vanderbilt University for supporting the development and use of domain specific visual modeling languages. Each modeling language dictates a set of rules about the types of parts available, containment relationships and inter-object relations such as connectivity. These rules are codified in a configuration file called a *paradigm*. Once a particular paradigm has been loaded into GME, GME supports the editing of models according to that paradigm. GME supports the partitioning of a system into views called *aspects*, facilitating the separation of concerns.

GME is packaged with a modeling paradigm, called MetaGME, which supports the creation of *metamodels*, or models of modeling languages. With MetaGME, users can define a new language which conforms to a particular engineering domain. A translator tool produces a paradigm from a valid metamodel. The metamodeling language is an extension of UML class diagrams, and offers the flexibility to integrate concepts such as hierarchy, inter-object relationships, object attributes and referencing into a modeling language.

GME also offers multiple APIs or interfaces for creating translator tools called *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.

Communicating Sequential Processes

Communicating Sequential Processes (CSP) is a mathematical theory of concurrency and interaction, in which interacting processes are modeled as event-transition systems that synchronize on shared events. The fundamental objects from which CSP process models are built are *events*, which are abstract symbolic representations of interactions. For example, a model of a financial transaction might consist of events that represent placing an order, acknowledgement of the order, payment, providing change, and handing over the purchased goods. Simple processes are built by defining sequences of events, separated by the prefix operator →, e.g.

```
SellEspresso =
  espresso_order → order_cost!$3 →
  receive_payment?p → make_change!(p - $3)
  → give_espresso → SKIP
```

CSP also provides a variety of operators for defining behaviors such as alternative actions (`SellEspresso [] SellLatte`), nondeterministic outcomes (`(espresso_order → Transaction) |~| (out_of_coffee → CloseStore)`) sequences of processes (`LoneBarista = SellEspresso; SellEspresso;` …), parallel execution of processes (`TwoBaristas = LoneBarista ||| LoneBarista`), and interfaces between processes (`Customer [|OrderEvents|] TwoBaristas`).

CSP supports a rich theory of process equivalences and refinements. Industrial strength tools such as FDR2 [4] can be used to rapidly check process models for properties such as deadlock, livelock, or refinement of a more abstract specification process. FDR2 has been in use for over a decade, and has been applied to a variety of applications across several domains, from industrial applications [5] and defense applications [6], to hardware design verification [7].

3. Modeling Spacecraft Behavior

Spacecraft systems designers have traditionally examined behavior only informally. Often, diagrams are used, but only for documentation. Consequently, there is no widely adopted standard for graphically representing spacecraft behavior. The BASS modeling paradigm, presented here, represents a starting point for the development of a design tool. The design of the graphical syntax was influenced both by currently employed informal notations, as well as by constructs developed by McInnes [8] for modeling spacecraft behavior using CSP. We envision an iterative development model for BASS, using feedback from spacecraft systems designers to improve the language.

System-Level Modeling

The subsystems and their connections to one another are

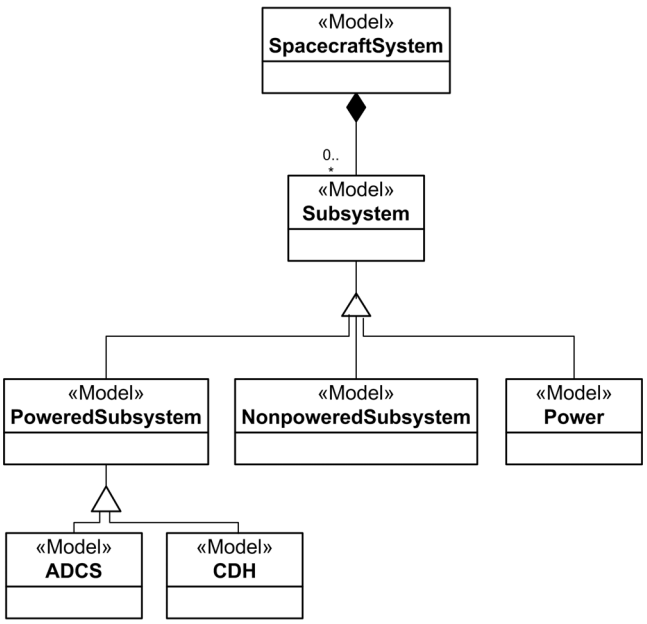

Figure 1. SpacecraftSystem and three types of Subsystems: Power, ADCS and CDH

captured in a top-level *SpacecraftSystem* model (Figure 1) that corresponds to a typical spacecraft system block diagram. We classify subsystems based on power consumption: some subsystems are powered, other subsystems are not powered (ex. structure). As the provider of power, the Power subsystem is in a category by itself.

The *SpacecraftSystem* offers two aspects, separating the views of power connectivity from data connectivity in the system. The parts available in the *PowerAspect* view are shown in Figure 2. The Power subsystem may contain several *PowerPorts*. Each *PowerPort* is capable of delivering power to another subsystem. The topology of the *PowerPorts* also models the structural connectivity of the power distribution network (star topology, single power bus, multiple power busses, etc). The *PowerConnection* connects the *SubSysPowerIf* to the *PowerPort* of the Power subsystem, representing the connection of the subsystem to the power network.

Data communication between subsystems occurs in multiple ways, as depicted in Figure 3. The primary vehicle for data communication is a *SystemBus*. Spacecraft may have multiple, independent busses, redundant busses, or a single bus, depending on the mission and resource availability. The bus carries multiple types of information. First, commands can be issued by the CDH subsystem to other subsystems. The set of commands accepted by a subsystem is captured as a *CommandSet*. Commands issued by the CDH are carried by a *SystemBus*. Command transmission is associated with a particular bus instance via the *CommandInterface* connection. The means by which the user models how the CDH selects which commands to send will be shown below.

The second type of information carried by the bus is spacecraft state information, which typically includes data indicating the current health of the spacecraft (ex. current temperature, position data, power level, etc.). State information can be used by the CDH to make operational decisions, and is also often stored for later downlink to the ground station. State information may be sent over a *SystemBus* as discrete responses to individual requests, or may be transmitted as a stream of telemetry data. Streams of information are represented using the *TelDataStream* construct (not shown), to which the *TelDataStreamRef* refers. The

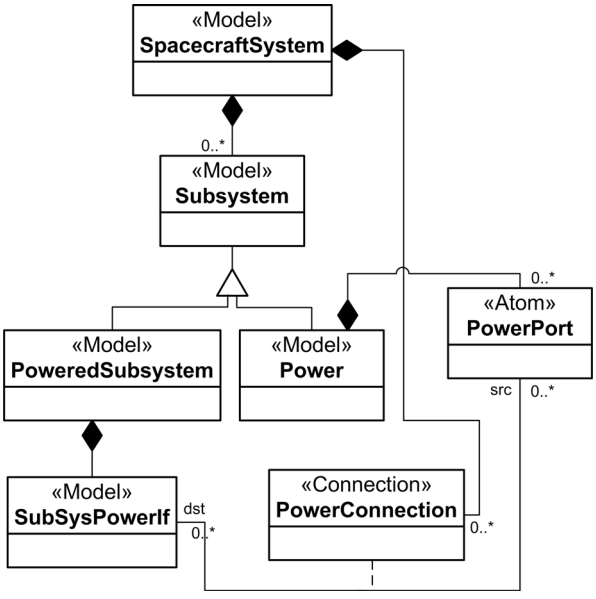

Figure 2. PowerAspect view of SpacecraftSystem

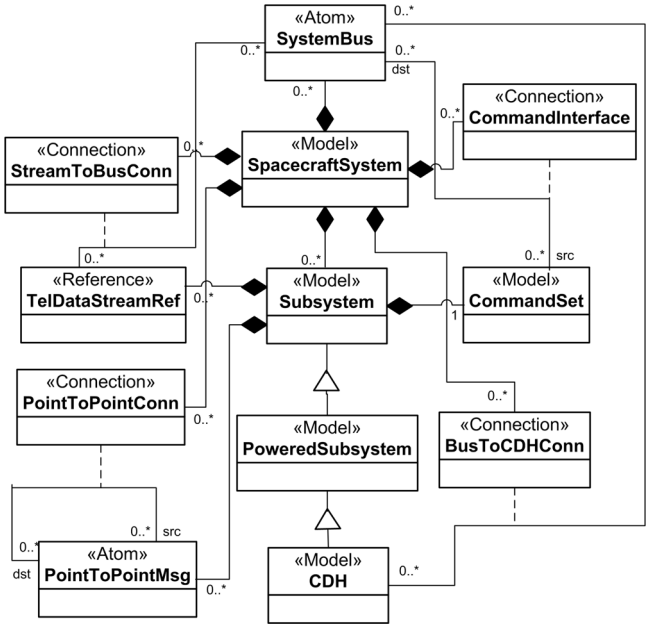

Figure 3. *SpacecraftSystem DataCommAspect*, showing data connectivity between the subsystems and CDH

StreamToBusConn allows the user to associate a stream with a particular *SystemBus*.

PointToPointMsgs are discrete messages sent from one subsystem to another. Physically, these messages are routed on dedicated wire connections between subsystems, modeled with the *PointToPointConn*. These messages are used to convey discrete packets of information which are not streamed, ex. an image captured by a science instrument to be recorded by CDH.

Modeling Subsystems

The system-level diagram specifies what subsystems are present in the spacecraft, and specifies paths for their interaction. The

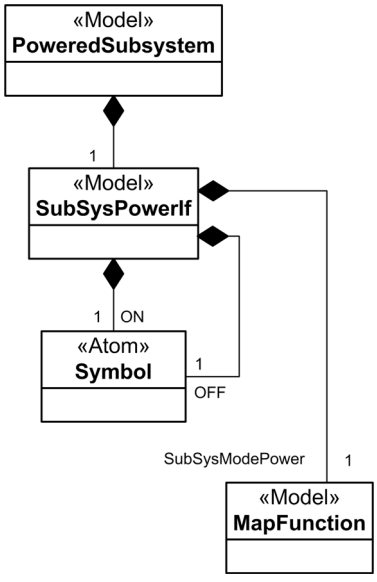

Figure 4. Power interface used by all powered subsystems

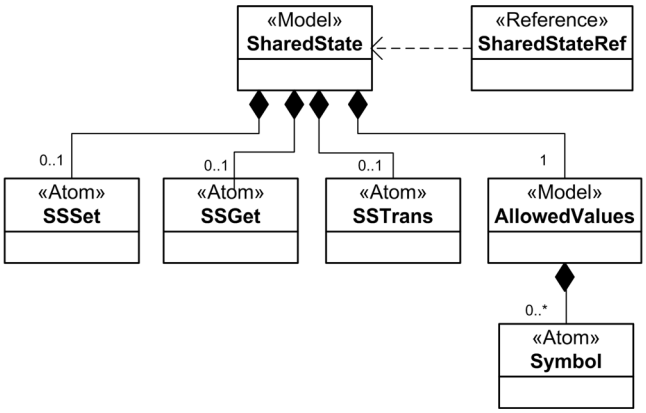

Figure 5. *SharedState* object, representing a shared variable

actual behavior of those subsystems is individually captured within each subsystem model.

Prior to discussing individual subsystems, we discuss some common constructs reused across multiple subsystems. We then discuss three types of subsystems: the Power, CDH and ADCS. Our discussion only summarizes the modeling facilities BASS offers to model subsystem behavior, with many low-level modeling details omitted.

Common Subsystem Constructs

Powered subsystems must interact with the Power subsystem. Each powered subsystem must specify a *SubSysPowerIf*, as shown in Figure 2. Figure 4 shows the internals of the *SubSysPowerIf*. The minimal power interface consists of two *Symbol* objects, one representing power to the subsystem being switched on, and the other representing power being switched off. These symbols are translated into CSP events by the model interpreter, and can be referred to in other parts of the system model. In addition to simple on and off states, some subsystems consume varying amounts of power depending on the mode they are in. The *MapFunction* allows the user to model this behavior, capturing a mapping between each mode of the subsystem and a corresponding change in power consumption.

Interactions between subsystems that are caused by dependencies on physical states are modeled in BASS using a *SharedState* object (Figure 5). *SharedState* objects have a well-defined type alphabet, as well as a well defined interface (*Set*, *Get* and *Trans* ports) for accessing the state.

As mentioned above, telemetry streams represent continuous flows of state information transmitted from one subsystem to another. During early phases of spacecraft design, specific values associated with streamed data are usually less important than the qualitative ranges of values that will trigger specific actions.

Furthermore, explicit enumeration of every possible value the streaming data can take on would inevitably produce a state explosion during model checking. Therefore, we restrict our stream model to qualitative transitions in the value of the state information the stream carries (see [8] for further details).

Power Subsystem

The Power subsystem is responsible for producing, storing and delivering power for the spacecraft. The most common kind of spacecraft uses solar arrays to generate power, and batteries to store power. Our current Power subsystem model focuses on solar-battery systems, and in particular attempts to address the fact that the amount of power that can sustainably be delivered by the Power subsystem can be a function of the attitude of the spacecraft (the attitude determines the angle at which on-board solar panels face the sun; angles approaching 90° result in higher power generation). The Power model (Figure 6) has two attributes, defining the minimum and maximum power generation capability of the spacecraft. The *PowerPorts* model the power interface to the outside world. The *MapFunction*, contained in the role of *AttitudeSpecificAvailablePower* is responsible for defining a mapping between spacecraft attitude and the power level available when the spacecraft is operating in that attitude. The definition of the *MapFunction* is omitted, but allows the user to associate a *Symbol* object, representing an attitude, with another *Symbol* object, representing a power level. Note that the Power subsystem also inherits containment of a *CommandSet* and *TelDataStreamRef* from the Subsystem class as shown in Figure 3. Hence the power subsystem can receive commands from CDH, and can stream health/status information back to CDH.

CDH Subsystem

The Command and Data Handling subsystem is responsible for coordinating the various subsystems onboard, logging state information, and interacting with the ground station. We consider separately two portions of the CDH subsystem: command and control, and data handling. Command and control consists of receiving commands from a ground station and dispatching them appropriately. A command received from the ground station may involve sending a single command to one subsystem, but frequently involves issuing a sequence of commands, where one command must complete before the next is issued. Figure 7 illustrates how commands are modeled. A *SimpleCommand* may be parameterized with a set of Symbols. A *CommandSequence* consists of multiple *Commands*, whether they be *SimpleCommands* or other *CommandSequences*. The *CommandSequencing* connection imposes a linear order on the *Commands* contained in the command sequence. *CmdRef* is a Reference to another command, for example a command belonging to a different subsystem. *SymbolMappingConn* connections can be used to bind the parameters of one command to the parameters of the following command in a command sequence.

From a modeling perspective, the specification of how a command is handled when it is received by the CDH involves defining a mapping from a command in the CDH command set onto either a *CommandSequence* or a *SimpleCommand*. The target command or sequence may be drawn from either the CDH command set, or from the command set of a different subsystem. Figure 8 depicts how command dispatching is modeled in BASS. *CDHCmdDispatch* consists of sets of <*Trigger, Target*> pairs.

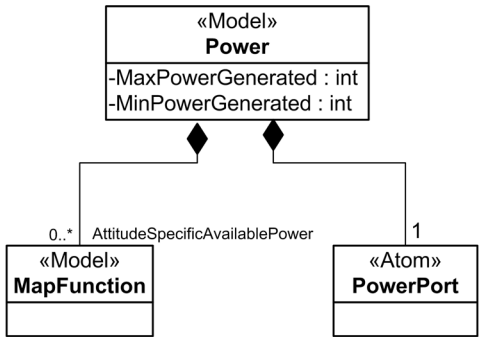

Figure 6. Power subsystem

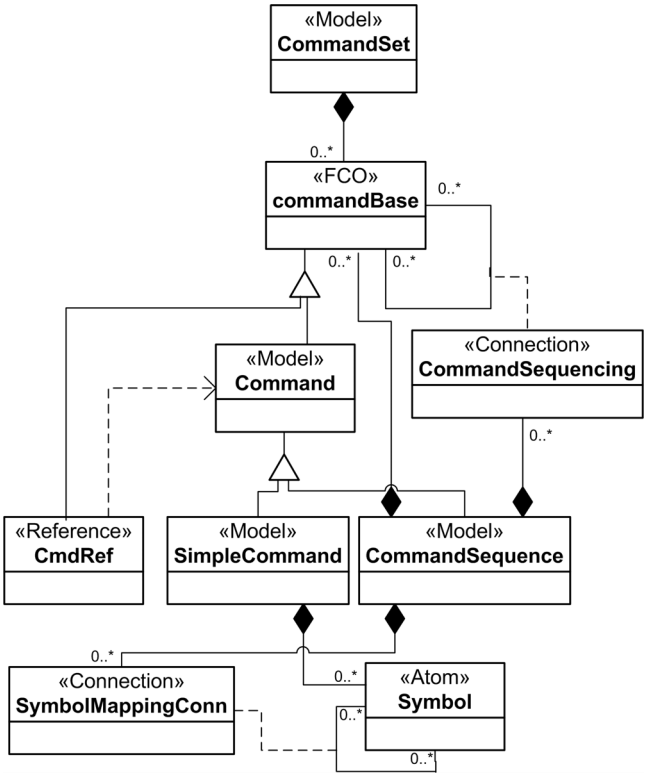

Figure 7. Spacecraft Commands

The *Trigger* is a reference to a command from the CDH command set which represents a command received from a ground station. The *Target* represents the result of the command dispatch, and can either be a reference to a command, or a *Symbol*. The *Symbol* is used to model the raising of an event, or the communication of a scalar flag to some subsystem. For example, in a command to the Power subsystem to turn on the power to ADCS the *Symbol* would be the *On Symbol* contained in the ADCS *SubSysPowerIf*. The *SymbolMappingConn* is used to indicate a mapping between the parameters of the *Trigger* to the parameters of the *Target*.

Attitude Determination and Control

ADCS is responsible for determining and maintaining spacecraft attitude, subject to commands issued by the CDH subsystem. Since we are concerned with system level behavior as a function of subsystem behavior, we abstract from the continuous dynamics control laws (which may be undefined during early design phases), and instead model the ADCS as a supervisory mode transition system. We assume that the ADCS includes one or more controllers that are capable of adjusting the spacecraft to attain the nominal attitude associated with a given ADCS mode when that mode is entered. Later design and analysis work by a control systems expert would be required to ensure that the ADCS does indeed meet this assumption. However, for the sake of high level behavioral analysis, the assumption allows us to determine whether the attitude changes resulting from a transition in ADCS mode cause, for example, undesirable changes in the available spacecraft power.

The *ADCSModeSystem* (Figure 9) is composed of *ADCSMode* objects, which model the ADCS modes, and *Symbol* objects, which model the rules for transitions between modes. The mode

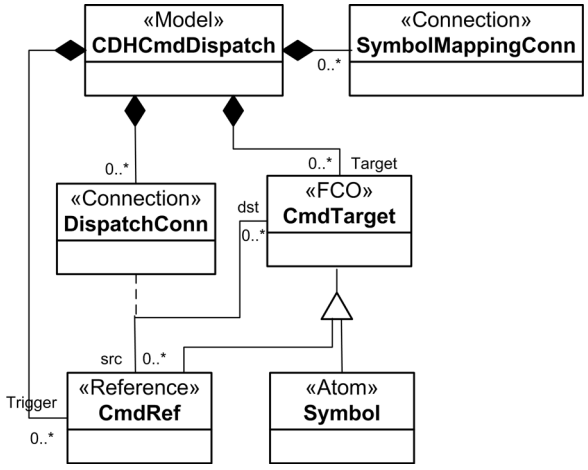

Figure 8. CDH Command Dispatch

transition *Symbols* may include *Symbols* present in the ADCS *CommandSet*, allowing receipt of a *SimpleCommand* to trigger an *ADCSModeSystem* transition. The *ADCSModeSystem* also contains an *AttitudeSet* object, which is a set of *Symbols* representing the nominal attitudes attainable by the spacecraft. These attitudes are associated with modes through the *AttToModeMap* connection. Each *Mode* must be associated with an attitude, but *Modes* may share attitudes. Also associated with each mode is a *ModeSpecificFn*, which represents the actions to be taken while in a particular mode. Such actions could include interacting with *SharedStates*, sending signals, or modifying telemetry streams.

Figure 10 depicts an example *ADCSModeSystem* containing three modes, *Safehold*, *Sci_Active* and *Sci_Standby*. Solid lines connecting modes to symbols ("sym" objects) model transitions. The symbols involved in transitions correspond to commands

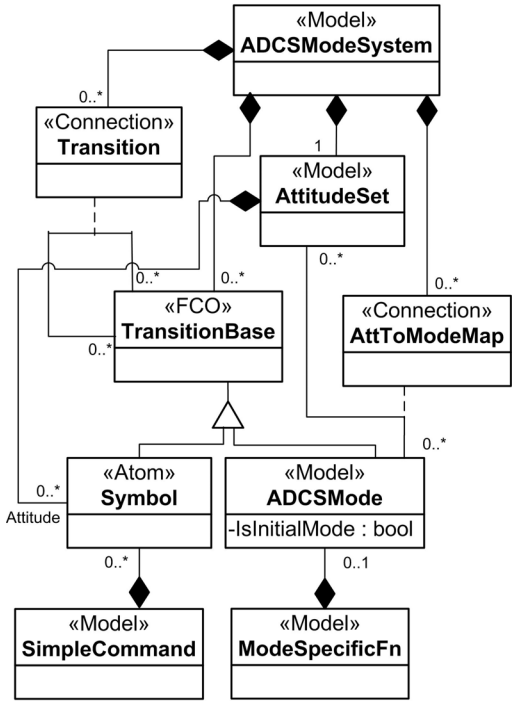

Figure 9. ADCS Mode System

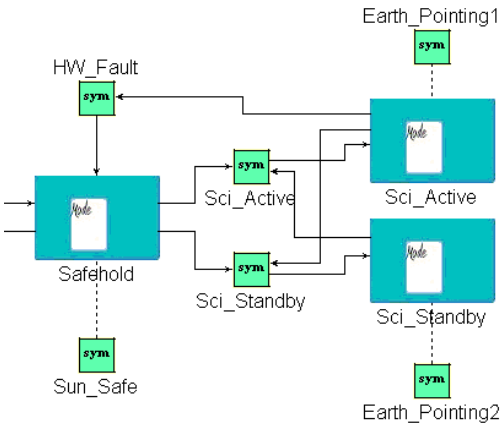

Figure 10. Portion of an ADCSModeSystem

received from the CDH, or to some other event (ex. *HW_Fault*) that can cause mode transitions. A dashed line connecting a symbol to a mode shows the mapping between attitude and mode.

4. BASS Toolflow

BASS offers the ability to model a spacecraft at the system level using the modeling language described in detail above. The modeling language is only one part of the BASS tool, as depicted in Figure 11. Development begins with the capture of system behavior models using GME and BASSML. An example system level diagram is shown in Figure 12. This example has only three subsystems – Power, ADCS, and CDH, communicating over a single system bus. The CDH port within the CDH model is actually a *CDHCmdDispatch* model, and contains the rules for how commands received from the ground station are dispatched to other subsystems via the System Bus. The Com ports of Power and ADCS are of type *CommandSet*, and contain command definitions for their respective subsystems. The connections between ports named Att and the *SystemBus* model the communication of the current attitude via a telemetry data stream from the ADCS to CDH. The models shown in Figure 12 are further refined into other diagrams which are omitted for brevity.

Once the system is captured in the BASS Modeling Language (BASSML), the Interpreter is applied to translate the model into

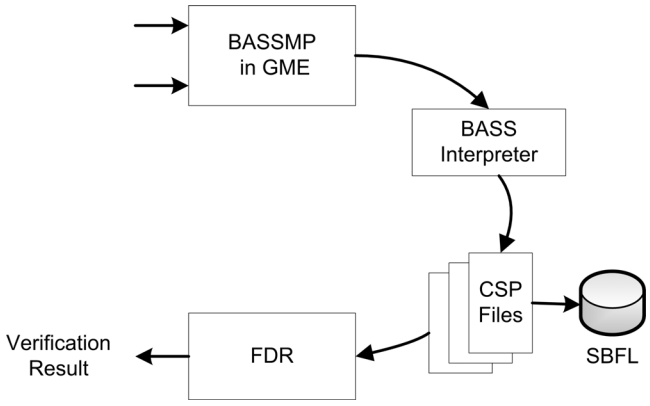

Figure 11. BASS Tool-flow

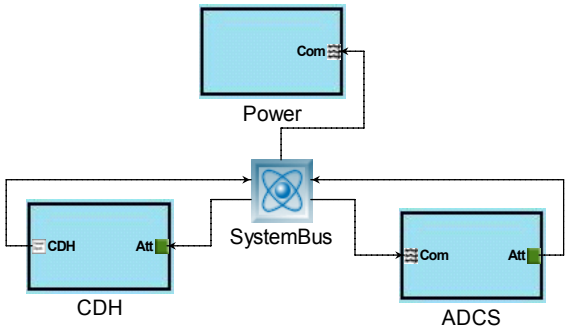

Figure 12. CDH Aspect of an example spacecraft model

machine-readable CSP. For each data communication path in the model, the interpreter produces a channel. Each *Symbol* defined in the model corresponds to an event which can be sent over a channel. Subsystem behavior is encoded as a set of processes, which interact using the generated channels. Our underlying process-based semantic model for BASS is described in detail in [8], which also describes a library of CSP processes for modeling spacecraft behavior (the "Spacecraft Behavior Framework Library", or SBFL) that is heavily used by BASS. The CSP model generated by the interpreter can be sent to the FDR2 model-checker for verification of specific behavioral properties (e.g. the spacecraft never reaches a deadlock state), as well as confirmation that nominal mission scenarios are feasible or that the system design implements a higher-level specification of system behavior (e.g. a functional flow block diagram). Analyses with FDR2 can be used to detect unanticipated interactions between subsystems that lead to errors such as activating a payload while the spacecraft is in an attitude that could damage the instrument, or improperly transitioning into a mode that require more power than is currently available. Such errors are often subtle, and can easily be overlooked during a cursory visual review of a model.

BASS can be used by spacecraft systems engineers throughout the entire system lifecycle, but is primarily intended to support specification and analysis in the preliminary design phase - what NASA calls "Phase B" [9]. Engineers at the Space Dynamics Laboratory and Air Force Research Laboratory have both expressed some interest in using a tool like BASS. However as yet, BASS has not been used outside of the laboratory, and we intend to refine the tool further via experiments with specification and analysis of student satellites such as USUsat1, USUsat2, and Toroid before releasing it to a wider audience.

Although BASS itself has not yet seen extensive use, our initial experiments with analyses of example specifications developed using the underlying CSP semantic model have shown that these analyses can be useful for uncovering several different kinds of errors, including

- **Interaction design errors:** for example, a mission-ending power-up sequencing error that escaped manual review (and indeed that the design had been specifically created to avoid);
- **System specification errors:** for example, an incompatibility between the subsystem interaction model and a higher-level system behavior specification (an FFBD) which exposed omissions in the higher-level specification;
- **Operations planning errors:** for example, a faulty commanding scenario that failed to place the spacecraft into the correct attitude for data gathering.

5. Related Work

Applying formal methods to spacecraft analysis is certainly not a new topic, although the focus of previous efforts has largely been on individual elements of software or hardware rather than on system-level interactions, and none have involved development of domain-specific languages.

NASA has carried out several experiments with formal methods. An analysis of flight software based on model extraction directly from source code into the SPIN model checker has been examined [10], and exposed design-level problems in the legacy software of the Deep Space One mission. Easterbrook *et al*. [11] successfully applied the PVS theorem-prover to check software requirements for consistency, and for safety and liveness properties. CSP has been evaluated and proposed for use as a specification language for use in the NASA ANTS mission architecture [12], and in the Formal Approaches to Swarm Technology project for specifying and verifying SWARM based missions [13].

Some tools offer a graphical interface to support formal verification. Hilderink developed a graphical modeling tool that has constructs for representing system behavior, and generates machine-readable CSP [14]. The generated CSP can be model-checked in FDR. However, the language constructs are generic and CSP-specific, rather than being designed for an application domain such as spacecraft design.

Specification Description Language (SDL) is another graphical specification language which uses formal methods [15]. 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[16].

6. Conclusions and Future Work

Spacecraft system design is difficult, and can lead to expensive, even catastrophic consequences when subtle design flaws are not caught early in the design process. In this paper, we present BASS, a prototype modeling tool for spacecraft systems. BASS utilizes a domain specific language targeting spacecraft designers. BASS integrates a model interpreter, capable of translating the captured spacecraft design models into machine readable CSP, which can be formally verified using the FDR2 model checker.

As part of our efforts to further refine BASS, we intend to examine and incorporate lessons learned from similar initiatives in other domains, such as the AUTOSAR-based modeling in the automotive domain [17] and MIMAD in the avionics domain [18] We will also explore closer integration of BASS with tools for spacecraft requirements capture. A prototype behavioral requirements capture tool called SDW [19], which we developed previously, is a particularly good candidate for integration efforts, since like BASS it relies on CSP for its semantic model.

References

[1] A. Ledeczi, A. Bakay, M. Maroti, P. Volgyesi, G. Nordstrom, J. Sprinkle, and G. Karsai, "Composing domain-specific design environments," Computer, vol. 34(11), pp. 44-+, NOV, 2001.

[2] "GME User's Manual," Vanderbilt University March 2004

[3] A. W. Roscoe, The Theory and Practice of Concurrency. Englewood Cliffs, New Jersey: Prentice Hall, 1998.

[4] P. Gardiner, M. Goldsmith, J. Hulance, D. Jackson, B. Roscoe, B. Scattergoood, and P. Armstrong, "Failures-Divergence Refinement: FDR2 User Manual," Formal Systems (Europe) Ltd. May 2003

[5] B. Buth, J. Peleska, and H. Shi, "Combining methods for livelock analysis of a fault-tolerant system," Proc. 7th International Conference on Algebraic Methodology and Software Technology (AMAST'98), Jan., 1999, pp 124-139.

[6] M. Goldsmith and I. Zakiuddin, "Critical systems validation and verification with CSP and FDR," Proc. International Workshop on Current Trends in Applied Formal Methods (FM-Trends 98), Boppard, Germany, Oct 7-9, 1998, pp 243-250.

[7] G. Barrett, "Model checking in practice: The T9000 Virtual Channel Processor," IEEE Transactions on Software Engineering, vol. 21(2), pp. 69-78, 1995.

[8] A. I. McInnes, "A Formal Approach to Specifying and Verifying Spacecraft Behavior," Ph.D. Dissertation, Utah State University, 2007

[9] R. Shishko and R. G. Chamberain, NASA Systems Engineering Handbook: NASA SP-6105, 1995.

[10] P. R. Gluck and G. J. Holzmann, "Using SPIN model checking for flight software verification," Proc. 2002 IEEE Aerospace Conference, 2002.

[11] 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(1), pp. 4-14, 1998.

[12] C. Rouff, A. Vanderbilt, W. Truskowski, J. Rash, and M. Hinchey, "Properties of a Formal Method for Prediction of Emergent Behaviors in Swarm-based Systems," Proc. Second International Conference on Software Engineering and Formal Methods (SEFM'04), 2004.

[13] M. Hinchey, J. Rash, and C. Rouff, "Some Verification Issues at NASA Goddard Space Flight Center," Proc. Verified Software: Theories, Tools, Experiments, 2005.

[14] G. H. Hilderink, "Graphical modelling language for specifying concurrency based on CSP," IEE Proceedings: Software, vol. 150(2), pp. 108-120, Apr., 2003.

[15] A. RockStrom and R. Saracco, "SDL-CCITT Specification and Description Language," IEEE Transactions on Communications, vol. COM-30(6), June, 1982.

[16] S. Ayache, E. Conquet, P. Humbert, C. Rodriguez, J. Sifakis, and R. Gerlich, "Formal Methods for the Validiation of Fault Tolerance in Autonomous Spacecraft," Proc. The Twenty-Sixth Annual International Symposium on Fault-Tolerant Computing (FTCS '96), 1996.

[17] "Specification of Interaction with Behavioral Models," Doc. No. 205, V1.0.5, Feb 14 2008. http://www.autosar.org/download/AUTOSAR_InteractionBehavioralModels.pdf

[18] A. Gamati, C. Brunette, R. Delamare, T. Gautier, and J.-P. Talpin, "A modeling paradigm for integrated modular avionics design," Proc. 32nd Euromicro Conference on Software Engineering and Advanced Applications (SEAA06), Cavtat/Dubrovnik, Croatia, Aug, 2006.

[19] B. Eames, A. I. McInnes, J. E. Crace, and J. M. Graham, "A Model-Based Design Tool for Systems-level Spacecraft Design," Proc. 20th Annual AIAA/USU Conference on Small Satellites, Logan, UT, August, 2006.