Reusable Software Architectures for Aerospace Systems*

Kathryn Anne Weiss, Elwin C. Ong, and Nancy G. Leveson
Massachusetts Institute of Technology

Abstract: Modern, complex control systems for specific application domains often display common system design architectures with similar subsystem functionality and interactions, making them suitable for representation by a reusable specification architecture. For example, every spacecraft requires attitude determination and control, power, thermal, communications, and propulsion subsystems. The similarities between these subsystems in most spacecraft can be exploited to create a model-driven system development environment in which generic reusable specifications and models can be tailored for the specific spacecraft design, executed and validated in a simulation environment, and then either manually or automatically transformed into software or hardware. Modifications to software and hardware during operations can be similarly made in the same controlled way, that is, starting from a model, validating the change, and finally implementing the change. The approach is illustrated using a spacecraft attitude determination and control subsystem, but applies equally to other types of aerospace systems.

1 Component-Based System Engineering

Reuse is clearly a partial solution to the long and costly development problems we are experiencing with complex control systems. The reuse of application software components, however, has had surprisingly limited success in many domains and has, at times, resulted in spectacular losses. In spacecraft, for example, NASA and the European Space Agency have lost billions of dollars and important scientific missions due to software reuse and poorly designed changes to operating software. The question is how to get the benefits of reuse without the drawbacks [15]. The answer may rest in the development level at which reuse is applied. The most problematic reuse has been attempted at the code level, but reuse may be more effective and safe by going back to an earlier development phase.

Component-based *system* engineering (as opposed to component-based *software* engineering) employs reuse at the requirements and specifications level, where required changes to the reused components are made and validated and the code is then regenerated either manually or automatically. Changes at that level can be made (or at least reviewed) by system engineers and domain experts who are more likely to understand the application's engineering requirements in depth and less likely to make changes that violate the basic engineering assumptions underlying the system. Domain knowledge can be captured and passed on to later projects. In this paper, we illustrate the

*The research in this paper was partially supported by NSF ITR Grant CCR-0085829, by a grant from the NASA Engineering for Complex Systems Program NAG2-1543, and by a grant from the NASA Intelligent Systems (Human-Centered Computing) Program NCC2-1223

approach with a domain-specific architecture for autonomous spacecraft. Although the development of such an architecture requires an investment, the payoff in terms of long-term development costs and time, risk reduction, and knowledge capture are potentially enormous.

The approach starts from a library of generic component specifications describing the component's hardware and software. The designer can then select appropriate components, assemble them into subsystems and, in turn, system specifications, and use simulation (the components are executable) and analysis tools to validate the design and do system testing early in the development process before any code is ever written. Human review can be enhanced by visualization tools and later testing of the implemented components by the automatic generation of test data from the component models. The executable specifications can act as an oracle during the code testing process.

A basic requirement for successful use of this approach is having specifications and models that include thoroughly documented design rationale and assumptions. The lack of such specifications has been cited in most of the well-known spacecraft accidents in the recent past [5], including several where reuse of software components was an important factor in the loss.

The accident report on the Ariane 501 explosion, for example, mentions poor specification practices in several places and notes that the structure of the documentation obscured the ability to review the critical design decisions and their underlying rationale [8]. Inadequate documentation of design rationale to allow effective review of design decisions is a very common problem in system and software specifications. It is especially critical when changes are made to the software or when software is reused. The Ariane accident report recommends that justification documents be given the same attention as code and that techniques for keeping code and its justifications consistent be improved.

The Mars Polar Lander report notes that the system-level requirements document did not specifically state the failure modes the requirement was protecting against (in this case possible transients) and speculates that the software designers or one of the reviewers might have discovered the missing requirement if they had been aware of the rationale underlying the requirements [3]. recommends that justification documents be given the same attention as code and that techniques for keeping code and its justifications consistent be improved [3].

The Mars Climate Orbiter (MCO) loss involved minor changes to software that was being reused from the Mars Global Surveyor (MGS) spacecraft [2]. According to the developers, the original software included a conversion from imperial to metric units, but that conversion was not documented and was inadvertently omitted when a new thruster equation had to be used because MCO had a different size Reaction Control System (RCS) thruster. "... the 4.5 conversion factor, although correctly included in the MGS equation by the previous development team, was not immediately identifiable by inspection (being buried in the equation) or commented in the code in an obvious way that the MCO team recognized it" [2] This is the type of problem that we believe can be avoided by the use of component-based development and reuse.

Complete and understandable specifications are not only necessary for development, but they are critical for operations and the handoff between developers, maintainers, and operators. The MCO operations staff had inadequate understanding of the automation and therefore were unable to monitor its operation effectively. Errors were introduced into the SOHO (Solar Heliospheric Observatory) ground software commands due to inadequate documentation and lack of system knowledge by those making the changes [9].

Good specifications that include requirements tracing and design rationale are critical for complex systems, particularly those that are software-intensive and those in which software components and design is reused. The specifications must be understandable and reviewable by domain experts and a wide range of engineering specialists, including operators. The information needed to reuse or

	Environment	Operator	System and components	V&V
Level 0	Project management plans, status information, safety plan, etc.			
Level 1 System Purpose	Assumptions Constraints	Responsibilities Requirements I/F requirements	System goals, high−level requirements, design constraints, limitations	Preliminary Hazard Analysis Reviews
Level 2 System Design Principles	External interfaces	Task analyses Task allocation Controls, displays	Logic principles, control laws, functional decomposition and allocation	Validation plan and results, System Hazard Analysis
Level 3 Blackbox Models	Environment models	Operator Task models HCI models	Blackbox functional models Interface specifications	Analysis plans and results, Subsystem Hazard Analysis
Level 4 Design Rep.		HCI design	Software and hardware design specs	Test plans and results
Level 5 Physical Rep.		GUI design, physical controls design	Software code, hardware assembly instructions	Test plans and results
Level 6 Operations	Audit procedures	Operator manuals Maintenance Training materials	Error reports, change requests, etc.	Performance monitoring and audits

Figure 1: A Sample Intent Specification

change the components as well as safely operate the spacecraft must be easily found when needed. Intent specifications were designed to satisfy these requirements [4].

2 Intent Specifications

Intent specifications are based on research on how experts solve problems and on basic principles of systems theory. An intent specification differs from a standard specification primarily in its structure, not its content: the specification is structured as a hierarchy of models designed to describe the system from different viewpoints, with complete traceability between the models. The structure is designed (1) to facilitate the tracing of system-level requirements and design constraints down into detailed design and implementation (and vice versa), (2) to assist in the assurance of various system properties (such as safety), and (3) to reduce the costs of implementing changes and of revalidating correctness and safety when the system is changed, as it inevitably will be.

There are seven levels in an intent specification, as shown in Figure 1. Levels do not represent refinement, as in other more common hierarchical structures, but instead each level of an intent specification represents a different model of the same system from a different perspective and supports a different type of reasoning about it. Refinement and decomposition occurs within each level of the specification, rather than between levels.

The top level (Level 0) provides a project management view and insight into the relationship between the plans and the project development status.

Level 1 is the customer view and assists system engineers and customers in agreeing on what should be built and whether that has been accomplished. Level 1 includes goals, high-level requirements, design constraints, hazards and preliminary hazard analyses, assumptions about the operating environment, and documentation of system limitations.

Level 2 is the system engineering view and allows engineers to reason about the system in terms of the physical principles and laws upon which the system design is based.

The blackbox model level (Level 3) includes executable models designed for specifying and reasoning about the logical design of the system as a whole and the interactions between its com-

ponents as well as its functional state without being distracted or overwhelmed by implementation issues. This level acts as an unambiguous interface between system engineering and subsystem engineering. This interface assists in communication and review of component blackbox behavioral requirements and in reasoning about the system or subsystem behavior using reviews, formal analysis, and simulation. The language used at this level, SpecTRM-RL, has a formal foundation so it can be executed and subjected to formal analysis while still being readable with minimal training and expertise in discrete math. It is at this level that we believe reuse is most effective and safest.

The next two levels provide the information necessary to reason about individual component design and implementation issues. Some parts of Level 4 may not be needed if physical code is generated automatically from the Level 3 blackbox software behavioral requirements models.

The final level, operations, provides a view of the operational system and is useful in mapping between the designed system and its underlying assumptions about the operating environment and real operating experience.

Each level also contains interface specifications and specification of human-automation interface design as well as a specification of the requirements for and results of verification and validation activities of the information at that specification level.

The information at each level is mapped to the levels above and below it (as illustrated later). These mappings provide the relational information that allows reasoning across the hierarchical levels and tracing from high-level requirements down to implementation and vice versa. Note that the structure of the specification does not imply that the development must proceed from the top levels down to the bottom levels in that order, only that at the end of the development process, all levels are complete. An environment that involves extensive reuse, for example, might follow a very different development process from one that involves a lot of first-time development.

Information about design rationale is critical to the success of model-driven development and component reuse. Intent information represents the design rationale upon which the specification is based. The required design rationale information necessary for successful reuse, including the underlying assumptions upon which the design and validation is based, is integrated directly into the intent specification and its structure, rather than relying on it somehow being captured in special documentation.

To avoid accidents and losses, reused components must be analyzed to determine whether they violate the design rationale and assumptions of the system within which they are to be used. This process is usually impractical, if not impossible, for reuse at the code level but not for reuse at the model or specification level (Level 3 or above).

During operations, if changes are made to any component or if conditions change such that the assumptions underlying the system design might be violated, new analyses should be triggered. Not only must the engineers know when assumptions change, but they must be able to figure out which parts of the design rely on those assumptions. Intent specifications are designed to make that process feasible.

We illustrate the use of a component-based specification architecture to support model-driven development and reuse with a domain-specific architecture for autonomous spacecraft constructed using a specification and modeling toolset called SpecTRM (Specification Tools and Requirements Methodology).

3 An Architecture for Model-Based Development of Autonomous Spacecraft

The first step in developing a reusable component-based specification architecture is the top-down decomposition of the system. For example, although spacecraft technology is continually advancing, most spacecraft require virtually the same generic functions. At the top-most level, a spacecraft control system consists of the Command and Data Handling Computer (CDHC). This software handles all of the resource allocation and subsystem commanding of the spacecraft as a whole. The CDHC can be decomposed into a set of subsystems: attitude determination and control, power, thermal, communications, guidance and navigation, propulsion, etc. [13].

The subsystems can then be further functionally decomposed into components. Most spacecraft attitude determination and control subsystems (ADCSs), for example, can be divided into a few types: inertial-based (gyros), celestial vector referenced (sun sensors, star trackers, earth sensors), magnetic-field referenced (magnetometers), or GPS-based. Attitude control devices are of two types: reaction control systems and reaction wheels. When designing a specific spacecraft, a set of attitude determination and control devices will be selected from these types.

In this paper we use the ADCS subsystem as an example, but a similar approach can be used for the power, thermal, and other spacecraft subsystems. Weiss has constructed a complete generic architecture for a NASA/MIT family of formation-flying nanosatellites [12]. This effort took one person approximately 40 hours to complete the specification and models of the generic nanosatellite architecture components.

After the functional decomposition, the next step in constructing a model-based, reusable spacecraft architecture is the construction of specifications for each of the generic components. For our spacecraft architecture, we used generic intent specifications we call SpecTRM-GCs (SpecTRM Generic Components). SpecTRM-GCs have four characteristics that are critical to the success of the architecture: each component is fully encapsulated, it has well-defined interfaces, it is generic, and it contains component-level fault protection.

Fully-Encapsulated Components. Each SpecTRM-GC component is fully encapsulated, meaning that all the functionality of that component is contained within it. In our example, it is important to note that traditionally many of the control functions for each ADCS component and much of the software to implement them is distributed among the CDHC, the ADCS, and the component itself. By fully encapsulating the operation of each device, the modularity of the design process and ease with which components are reused increases.

Well-Defined Interfaces. In order to create plug-and-play component models, a strict interface naming convention needs to be followed and an interface specification created. This requirement is no different than for any type of reusable component.

Generic Specifications and Models. To enhance reusability, specific information is either left out of the specification (but identified as required to be added when the generic specification is used for a particular spacecraft), or included but identified as potentially needing to be changed. The use of bold face and underlining highlights such information in SpecTRM-GC specifications. For example, one of our ADCS generic components is a digital sun sensor. When, say, an Adcole Digital Sun Sensor Model 18960 is being used, the SpecTRM-GCs require that specific information be added, such as the fact that the Adcole 18960 uses 15 bits of input from its sensor heads.

Any changes required in other parts of the specification and models implied by that decision are identified by links to other locations.

Component-Level Fault Protection. Fault protection is particularly important in spacecraft architectural design. Because spacecraft engineers usually only get one chance to get it right, spacecraft design is traditionally very conservative. Control actions usually follow a programmed sequence in which specific activities are performed at fixed and predetermined times. These pre-programmed timelines assume all the participating components are operating nominally. If there is a faulty component, the control system immediately switches into a *safe mode* and radios back to earth for assistance. This approach has two important limitations. The first is the potential communication delay as the spacecraft gets farther from Earth. For nominal flight phases, the delay causes no problem. But if the fault occurs during a critical maneuver, for example, the inertial measuring unit fails during an insertion into a Martian orbit, the spacecraft could potentially be destroyed before ground control has a chance to intervene.

The second limitation is the large number of ground controllers currently required to support each phase of flight for a deep space mission. At any point during the Cassini mission, for example, there are 300 people manning ground control. If there are to be more and cheaper deep space missions, spacecraft will need to have more autonomy and be able to solve problems on their own.

Our example autonomous spacecraft architecture employs three levels of fault protection: intra-component fault protection, inter-component fault protection, and inter-subsystem fault protection. These three levels ensure that fault protection covers the entire system: not only must individual component failures be accounted for and handled, but also failures resulting from the interactions between components and subsystems. At the intra-component level, the fault protection logic assures that if the component is working in an off-nominal mode, it will alert its subsystem. Then, at the inter-component level, the subsystem determines how to handle that fault. The use of a model-driven development process means that fault protection can be designed and thoroughly validated before any code is generated [10].

4 An Example Model for a Generic Spacecraft Architecture

Consider the ADCS subsystem and one of its components, the Reaction Wheel Assembly (RWA). A RWA controls the spacecraft attitude using control moment gyros [14]. Each reaction wheel contains a motor connected to a mass, a motor speed sensor, and associated input and output electronics. In space, a rotating mass creates a torque, which is perpendicular to its spin axis. This torque is used to change the angular momentum of the spacecraft, thereby rotating the spacecraft in the opposite direction to the torque axis of the motor. To provide attitude control of all three spacecraft axes, at least three reaction wheels must be operational in any RWA, although to allow for failures, four reaction wheels are used in a typical spacecraft.

Our generic specifications include a complete intent specification: reuse of formal models alone will not provide the information necessary to create a safe and correct spacecraft implementation. We include here only a small and very incomplete part of the specification.

An example Level 1 requirement on the RWA is:

FR.1: The RWA shall receive commands from the ADCS **once per second** and transform them into torque commands on the individual reaction wheels. [*DP.1*]

Assumption: The ADCS commands will be in the form of torque values to be applied on each of the spacecraft's three axes.

Note the need to provide relevant operational environment assumptions that the RWA design makes about any potential spacecraft that uses this RWA specification. The phrase **once per second** is underlined and bold face to alert the system engineer that this part of the specification may need to be changed for a particular spacecraft ADCS design. The link at the end of the requirement ([*DP.1*]) points to the system design features at Level 2 created to implement the requirement.[1]

For the example requirement FR.1, Level 2 includes a definition of how the torque commands provided from the ADCS are transformed into torque commands to the individual wheels (much of the detail has been omitted from this example) :

DP.1: The speed commands for each individual reaction wheel are derived from the torque commands provided by the ADCS. [*FR.1*]

DP.1.1: Commands from the ADCS are in the form of three torque values, T_x, T_y and T_z. These commands are first transformed into the wheel reference frame and then converted into torque commands for each individual wheel using the following transformation [*Torque-Command*]:

$$\begin{bmatrix} \hat{T}_{cx} \\ \hat{T}_{cy} \\ \hat{T}_{cz} \end{bmatrix} = \begin{bmatrix} T_{cx}/\cos\beta \\ T_{cy}/\cos\beta \\ T_{cz}/\sin\beta \end{bmatrix} = \begin{bmatrix} 1 & 0 & -1 & 0 \\ 0 & 1 & 0 & -1 \\ 1 & 1 & 1 & 1 \end{bmatrix} \begin{bmatrix} T_1 \\ T_2 \\ T_3 \\ T_4 \end{bmatrix}$$

...

DP.1.2: The RWA sends each wheel a torque command **once per second**. This command is in the form of the torque value calculated from [*DP.2.1*]. The reaction wheel's electronics uses this value to compute the speed necessary to achieve the commanded torque and provides the motor with the current needed to reach the desired speed [*Torque-Command*].

Level 2 includes links to the corresponding functional requirement(s) in Level 1, links to other related information within Level 2 (such as the corresponding validation requirements and results), and links to the formal models at Level 3 containing the detailed logic needed to implement the system design. In the example, [*Torque-Command*] is a link to the detailed logic (the equivalent of a software requirements specification in SpecTRM) required for issuing a torque command to a reaction wheel. We model this detailed logic at Level 3 of the intent specification using a blackbox modeling language called SpecTRM-RL. SpecTRM-RL has an underlying state machine model [7], making the models executable and formally analyzable.

[1] In the SpecTRM tool, these links are implemented as hyperlinks and can be easily created and changed.

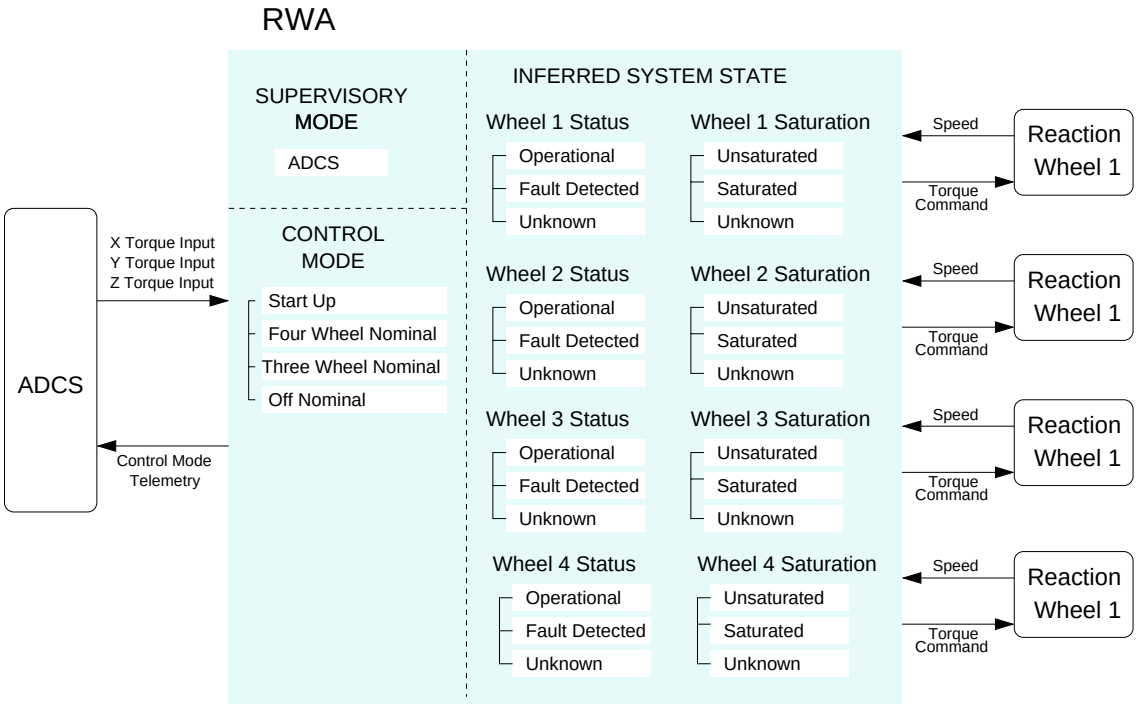

Figure 2: The Graphical Blackbox Model of the Reaction Wheel Controller in SpecTRM-RL

For successful model-driven development, domain experts must be able to review and ideally to create the models. The translation from system requirements and design to component requirements is critical—many recent software-related aerospace system losses, such as Mars Polar Lander, have stemmed from problems arising in the translation from system requirements to software requirements or in changes to existing software made by software engineers who did not understand all the basic engineering assumptions underlying the functionality provided (Mars Climate Orbiter) [5].

It is unreasonable to expect engineers and other domain experts (for example, ground controllers and astronauts) to read and find errors in code or even in most of the widely used modeling and specification languages such as UML. SpecTRM-RL was designed with the primary goals of readability and ease of learning and to be a requirements specification and modeling language that is usable by everyone involved in the creation of a spacecraft and its components.

The SpecTRM-RL language design has evolved over the past ten years through what we have learned from using it on real projects as well as careful laboratory experimentation [16] about the modeling language design features that enhance readability, reduce error-proneness [7], and encourage complete specification of important information that is often omitted [6]. Most engineers can learn to read SpecTRM-RL models (and they can find errors in the modeled behavior) with about ten to fifteen minutes training.

Figure 2 shows the graphical part of the SpecTRM-RL model of the RWA and the immediate control loops within which it is embedded in our spacecraft architecture. The RWA is controlled by the ADCS and in turn controls the reaction wheels. On the left is its controller (the ADCS). The right side shows the hardware the RWA is in turn controlling and from which the RWA receives feedback (the individual reaction wheels). The shaded part of the model describes the required blackbox behavior of the reaction wheel controller—internal design information is not included. The ADCS would have an equivalent model as well as one for each of the generic ADCS components.

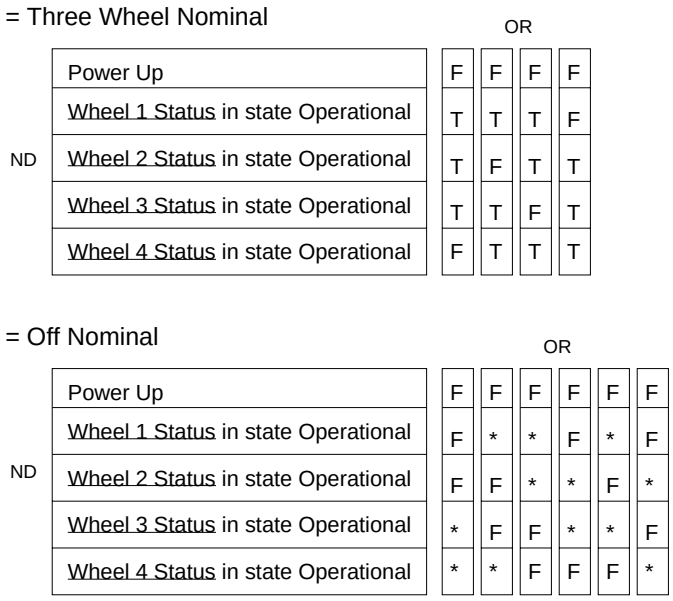

Figure 3: AND/OR Tables Showing Control Mode Transition Logic

When designing a specific spacecraft, the designer would select from a library the SpecTRM-GC models for the components being used in that spacecraft.

There are three parts to the blackbox behavior description. Using the SpecTRM-RL RWA model in Figure 2 as an example, the supervisory mode (in the upper left-hand corner) shows the components that can provide control commands to the RWA controller. In this case, there is only one (the ADCS). In general, any controller may in turn be controlled by multiple supervisory controllers and it may be necessary to limit the behaviors allowed under different supervisory modes.

The possible subsystem or component control modes are shown below the supervisory mode. For the RWA controller these are Startup, Four Wheel Nominal, Three Wheel Nominal, and Off Nominal.

The component's model of relevant parts of the current system state is shown to the right of the dashed line. At any time, a controller has only a model, inferred from inputs, previous outputs, and other information about the real state of the controlled components (the *plant* in control theory terminology). The required behavior of the component is defined with respect to the current state of this inferred system state model. In the RWA controller, the inferred state model has eight state variables, representing information about the current state of the four reaction wheels that make up the assembly. The graphical notation also shows the possible values for these state variables; for example, the Wheel 1 Status state variable can have the values operational, fault detected, or unknown. When executing SpecTRM-RL models, the current values of the inputs, outputs, control modes, and state variables light up or are otherwise depicted on the screen. Outputs can also be directed to other animation and analysis tools to assist in validating the models and system design.

The behavior of the RWA controller, i.e., the logic for sending commands to the wheels and updating the inferred system state, is specified using a tabular notation called AND/OR tables although we are working on additional ways to visualize this information [1]. The rows of the tables represent AND relationships while the columns represent OR. For example, Figure 3 shows the conditions under which the RWA control mode becomes *Three Wheel Nominal* and *Off Nominal*. The transition to the particular control mode is taken if any of the columns evaluate to TRUE. A column evaluates to TRUE if all of the rows have the value specified for that row in the column. An

asterisk indicates that the value for the row is irrelevant. For the example, Three Wheel Nominal control mode is entered if one and only one of the reaction wheels has entered a Fault Detected State (does not have the value *operational*) while Off Nominal mode results if a fault is detected in two or more of the wheels. Code can easily be generated automatically from these tables and other parts of SpecTRM-RL as well as unit test data to verify the implementation of the logic in the table. The SpecTRM-RL models can act as the test oracle.

5 Using the Generic Models in a Model-Driven Development Environment

Once a reusable component-based architecture for a particular application domain has been created, system designers can select components, tailor them to the particular design, and put them together in a plug and play environment. If the models are executable, then they can be executed individually, interacting with each other, or within a more extensive simulation environment to validate the spacecraft design and to evaluate alternative design options. If the models are also formal, various types of analysis are possible. For example, SpecTRM has automated tools to check for non-determinism and robustness of SpecTRM-RL models, thus assisting the engineer in eliminating logical inconsistencies and important types of incompleteness.

Expert review is a critical part of any development process, and any models must be easily reviewed and understood by domain experts. Visualization tools can assist in this process, but to be acceptable in an industrial environment, the basic modeling language must be carefully designed to avoid ambiguous or complex semantics and obscure notations.

Once the design is validated, the implementation of the components can be generated either automatically or manually from the component models. If the models are formal, then test data can be generated from the model to ensure various types of coverage and correctness of the implementation.

The same process can be used if a new design is not being created, but a change is being made to an existing spacecraft design or implementation. Instead of making the change directly to the code, the change would be made to the model, validated, and then new code generated. In fact, we know this process can work—our model of TCAS II (an airborne collision avoidance system) that we created for the FAA a decade ago using a predecessor of SpecTRM-RL is still being used today to evaluate potential changes and fixes to the system before they are implemented.

References

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

[2] Edward A. Euler, Steven D. Jolly, and H.H. Curtis. (2001). The Failures of the Mars Climate Orbiter and Mars Polar Lander: A Perspective from the People Involved. *Guidance and Control Conference*, Paper AAS 01-074, American Astronautical Society.

[3] JPL Special Review Board. (2000). Report on the Loss of the Mars Polar Lander and Deep Space 2 Missions. Nasa Jet Propulsion Laboratory, 22 March.

[4] Nancy G. Leveson. (2000). Intent Specifications: An Approach to Building Human-Centered Specifications. *IEEE Transactions on Software Engineering*, SE-26(1), January.

[5] Nancy G. Leveson. (2003) The Role of Software in Aerospace Accidents. *AIAA Journal of Spacecraft and Rockets*, to appear.

[6] Nancy G. Leveson (2000). Completeness in Formal Specification Language Design for Process-Control Systems. *ACM Conference on Formal Methods in Software Practice*, Portland, August.

[7] Nancy G. Leveson, Mats M.P.E. Heimdahl, and Jon D. Reese. (1999). Designing Specification Languages for Process Control Systems: Lessons Learned and Steps to the Future. *ACM/Sigsoft Foundations of Software Engineering/European Software Engineering Conference*, Toulouse, September.

[8] Lions, J.L. (1996). Ariane 501 Failure: Report by the Inquiry Board. European Space Agency, 19 July.

[9] NASA/ESA Investigation Board. (1998). SOHO Mission Interruption. NASA, 31 August.

[10] Elwin Ong. (2003). *Specifying Spacecraft Fault Protection using Generic, Domain-Specific Intent Specifications.* Master's Thesis, Aeronautics and Astronautics Dept., MIT.

[11] Marcel J. Sidi. (1997). *Spacecraft Dynamics and Control: A Practical Engineering Approach.* Cambridge, U.K.: Cambridge University Press.

[12] Kathryn A. Weiss. (2003). *Building a Reusable Spacecraft Architecture using Component-Based System Engineering.* Master's Thesis, Aeronautics and Astronautics Dept., MIT.

[13] James R. Wertz and Wiley J. Larsen (Editors). (1999). *Space Mission Analysis and Design*, Third Edition. Dordrecht, The Netherlands: Kluwer Academic Publishers.

[14] James R. Wertz. (1978). *Spacecraft Attitude Determination and Control.* Dordrecht, The Netherlands: Kluwer Academic Publishers.

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

[16] Mark Zimmerman, Nancy Leveson, and Kristina Lundqvist. (2002). Investigating the Readability of State-Based Specification Languages. *International Conference on Software Engineering*, Orlando Florida, July.