Making Embedded Software Reuse Practical and Safe

Nancy G. Leveson, Kathryn Anne Weiss
Aeronautics and Astronautics; Engineering Systems
Massachusetts Institute of Technology

ABSTRACT

Reuse of application software has been limited and sometimes has led to accidents. This paper suggests some requirements for successful and safe application software reuse and demonstrates them using a case study on a real spacecraft.

Categories and Subject Descriptors

C.3 [**Computer Systems Organization**]: Real-Time and Embedded Systems; D.2.1 [**Software Engineering**]: Requirements/Specifications; D.2.2 [**Software Engineering**]: Design Tools and Techniques—*Software Libraries*; D.2.13 [**Software Engineering**]: Reusable Software

General Terms

Documentation, Design

Keywords

Software reuse, Real-time and embedded software

1. UNDERSTANDING THE PROBLEM

Reuse is clearly a partial solution to the long and costly development problems we are experiencing with complex control systems.[1] Development costs for such systems can reach billions of dollars, and they can take a decade to complete. In some cases, the technology has become obsolete before the systems are finished (e.g., the FAA's microwave landing system). The reuse of application software, however, has not lived up to its promises and has, at times, resulted in spectacular losses. In spacecraft, for example, NASA, the European Space Agency, and the Air Force have lost billions of dollars and important scientific missions due to software reuse and poorly designed changes to operating software [10].

[1]For a very comprehensive survey of software reuse and an extensive bibliography, see Kruger [7].

Examining the Mars Climate Orbiter (MCO) loss is instructive. The loss of the MCO involved minor changes to software that was being reused from the Mars Global Surveyor (MGS) spacecraft [3]. According to the developers (but surprisingly absent from the accident report), the original software contained 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" [3].

The problems are not simply related to safety. Goodman describes the experiences United Space Alliance had in trying to take GPS software that was created for aircraft and reuse it for spacecraft. Although they assumed that the off-the-shelf GPS units with proven design and performance would reduce acquisition costs and require minimal adaptation and minimal testing, the time, budget, and resources needed to test and resolve software issues greatly exceeded initial projections:

> Software evolves and changes over time. Many vendors have a library of software modules, many of which are used in multiple applications. Software errors that manifest in a particular application may be deemed to have "no impact" to the user and are not corrected. This causes software errors to propagate through succeeding product lines, with the potential for affecting future users in different applications. Changes in operating environment that come with a new application may invalidate assumptions made during initial requirements definition and result in software issues during testing and operation [5, p.3].

Some successful reuse reports have been premature. The modules may be reused in the next similar project, but fewer can be used in later projects. In fact, in the well-known NASA Goddard reuse experiment, while reuse was high in the next project (about 70%), it quickly dwindled over time to the point where the original extra cost of producing reusable modules was never recovered [15]. While it may appear that new versions of embedded systems do not change dramatically, in fact there are enough changes in the missions and design of the physical systems and devices the software is controlling—after all, the reason a new system is being built at all is to change or augment functionality—that changes to the software will usually be required to reuse it. This fact does not preclude application software reuse, but it does suggest some requirements for reuse, including the fact that it must be possible to make changes easily and safely.

The question is how to get the benefits of reuse without the drawbacks [19]. 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 and beginning the reuse from that phase (i.e., reuse the work performed up to that phase). Because coding is such a small part of the software engineering process, particularly for real-time embedded control software, and coding for these applications can even be partially automated from requirements specifications, the cost of repeating the coding step is insignificant compared to the potential cost of revalidating the reused code. In addition, safely changing code is easier when the changes are made at an earlier development phase.

The rest of this paper describes the requirements we believe must be satisfied for successful, large-scale reuse of embedded control software and a case study using a real system to evaluate the feasibility of satisfying these requirements. Note that this paper focuses on application software reuse, not system software although some of the requirements and solutions may be similar.

2. REQUIREMENTS FOR EFFECTIVE SOFTWARE REUSE

Knowledge capture in the specification of the software and the controlled system is critical for successful reuse of application software. That specification must contain the following:

- *Documentation of design rationale*: A basic requirement for successful and safe reuse is having thoroughly documented design rationale and design assumptions for both the system and the software design. The lack of such specifications has been cited in most of the well-known spacecraft accidents in the recent past where reuse of software components was an important factor in the loss [10]. 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 [12]. The report recommends that justification documents be given the same attention as code and that techniques for keeping code and its justifications consistent be improved. One of the conclusions of the GPS project mentioned above was that they needed insight into the rationale and requirements that governed the original design of the GPS units to successfully reuse them in a different environment from that for which they were designed [4]. Goodman notes that the documentation they received was limited to "how" and often did not include "why":

  > Software requirements documents contain equations to be used, but rarely provide insight into how the equations were derived, or how values of constants were determined. This information exists on paper at some point, in the form of informal memos and company internal letters. However, over time, this information is lost due to employee attrition, clean-out of offices, retirements and corporate takeovers. Many mathematical results used in navigation algorithms no longer exist in the open literature. Corporate knowledge loss makes it difficult for engineers to understand, evaluate, and modify software years or decades after it was written and certified [4, p.9].

- *Documentation of the assumptions about the operational environment that are implicit in the software*: These assumptions include interfaces with other components and other structural features, but also include assumptions about behavior. In the SOHO (SOlar Heliospheric Observatory) mission interruption [13], critical assumptions about the operation of the gyros were not recorded anywhere and were violated when the ground control software modules were reused. The GPS integration attempt described earlier found that vendors tend to perform the minimum amount of lab testing necessary to ensure that the unit meets contract specifications. Without specification of the assumptions of the environment in which the system was developed, tested, and used, it is not possible to determine what additional testing and analysis needs to be performed or what changes may be necessary to meet the conditions in a different operational environment. For example, many of the software issues that arose with the GPS units involved design problems that are not manifested in aviation applications where flight times may last minutes or hours but may appear in the much longer space flight applications. Goodman notes that the documentation United Space Alliance received about the GPS units did not include the assumptions made in designing the GPS receiver for terrestrial applications that were invalid for the new space applications [4].

- *Traceability from high-level system requirements to system design to software requirements to code and vice versa*: By traceability, we do not mean simply including the standard traceability matrix showing the mapping between high-level requirements and software modules or CSCIs but instead traceability to system design features and decisions. Such traceability allows those planning reuse to make sure that the requirements and assumptions about the operation of the component fit the new use and to determine any interactions with other components that need to be considered. In the SOHO mission interruption, the accident report cites one factor as a lack of system knowledge by the person modifying the software procedure. Clearly the same was true for MCO. Not having this system knowledge and traceability can lead to fear of modifying reused software modules, which was a factor in both the Ariane and Milstar launch accidents [14]. Traceability potentially provides a way to acquire the system knowledge necessary to successfully reuse software.

- *Documentation of hazard analysis and safety information*: A hazard analysis needs to be performed for each safety-critical system. Without information about the original hazard analysis and the specific safety constraints related to the reused software component, it is very difficult to perform this analysis. This difficulty, coupled with lack of documentation of reused software module and common issues with proprietary information, makes reuse very hazardous. While software may be perfectly safe in one environment, it can lead to accidents in another. The Ariane 501 loss is an example, where a difference in the Ariane 5 trajectory from the Ariane 4 trajectory triggered the unsafe software behavior. The same software error that led to the Therac-25 deaths had a benign manifestation in the earlier Therac-20. The blackbox (externally visible) behavior of a component can only be determined to be safe by analyzing its effects on the system in which it will be operating. Cost-effective reuse of safety-critical software requires clear documentation of the assumptions and procedures underlying the original hazard analysis.

Simply including this necessary information in the specifications is, however, not enough. The information must be specified in a way that is easy to find, use, and change. Most projects have voluminous documentation, but it is often difficult to understand and is incomplete, inconsistent, and ambiguous. In addition, validation and verification of the new system and the reused components within it will require the collaborative efforts of many types of domain experts. The language used in the specifications must be easily readable and reviewable with minimal training by non-software engineers. Accomplishing this goal requires minimizing semantic distance between the specification and the engineers' mental models, incorporating standard notations when possible, minimizing obscure notations, and including animation and visualization tools to assist in understanding the behavior specified or modeled.

Some other characteristics and practices can, in our experience, assist with reuse, but are not necessarily required. One is model-based development using blackbox models of software behavior. Such model-based development is not by itself enough to make reuse safe, but combined with the required characteristics above, reuse can start with the models, necessary changes can be made, the changes validated, and then code generated (either manually or automatically) from the models. In fact, there is evidence that this process works: TCAS has been maintained and gone through multiple versions since our first model was created in the early 1990s [11]. Each time a new version is needed, the developers go back to the RSML specification, make changes to the model, and then validate the newly changed model. When they are convinced that the changes are effective and safe, new code is created by incorporating any changes that have been made in the models. Because our specification includes complete traceability from the TCAS models to the code, the process of making and verifying changes to the code is greatly simplified.

Perhaps more controversial, we believe that object-oriented system design may inhibit safe reuse of embedded application software and has other important drawbacks. The authors' experience and that of others on complex projects (e.g., Pathfinder [17] and Iridium [8]) is that functional decomposition of embedded control software is safer and more effective than object-oriented system design, particularly with respect to establishing the very high level of confidence necessary in control software and in its maintenance and reuse. Problems in OOD arise particularly from the tight functional coupling involved in spreading control functions among the objects rather than providing the functionally cohesive modules that result from functional decomposition of the system requirements.

Verification of the safety of the original, reused, or changed code is basically impossible for object-oriented system design. Those building such systems in the aviation realm have suggested that object-oriented technology is inappropriate for safety-critical applications [6]. Safety problems arise from interactions among components involved in achieving safety-critical functions. Safety analysis, therefore, focuses on the safety-critical system functions, which in object-oriented system design can potentially be spread throughout so many objects (and involve multiple inheritance levels) that high confidence in the system behaving safely is impossible to provide. Note that only programming-in-the-large is being considered here. Object-oriented design within the individual models is perfectly reasonable and usually safe.

3. A CASE STUDY

The approach to reuse employed in the case study starts from a library of generic executable component specifications describing the component's blackbox hardware and software behavior. Only externally observable behavior is specified, not the internal design. The developer then selects appropriate components, assembles them into subsystems and system specifications, and uses simulation (the specifications are executable) and analysis (the specifications are formal) to validate the design and perform at least partial system testing. For embedded software, testing and formal analysis are not adequate to find all the errors so human review by a varied group of domain experts is also required. Human review can be enhanced by visualization tools. Later verification and testing of the reused and perhaps changed modules and system can be assisted by automatic generation of test data from the component models, and the executable specifications can act as a test oracle during the code testing process.

Although several different toolsets can be used to implement such a approach, Weiss used a commercial requirements specification tool called SpecTRM [16] to perform a case study of reuse on a family of autonomous spacecraft (nanosatellites) called SPHERES [18]. SPHERES (Synchronous Position Hold Engage Reorient Experimental Satellites) was created by MIT's Space Systems Laboratory to provide NASA and the Air Force with a reusable, space-based testbed for high-risk metrology, control, and autonomy technologies critical to the operation of distributed satellite and docking missions, such as the Terrestrial Planet Finder and Orbital Express. The SPHERES testbed was designed to operate in the micro-gravity conditions on the International Space Station although deployment has been delayed due to the Columbia accident. In addition, it is planned for guest scientists from around the world to have access to this testbed to independently design and evaluate estimation, control, and autonomy algorithms for autonomous coordination and synchronization of multiple spacecraft in tightly controlled spatial configurations. The very nature of the purpose of

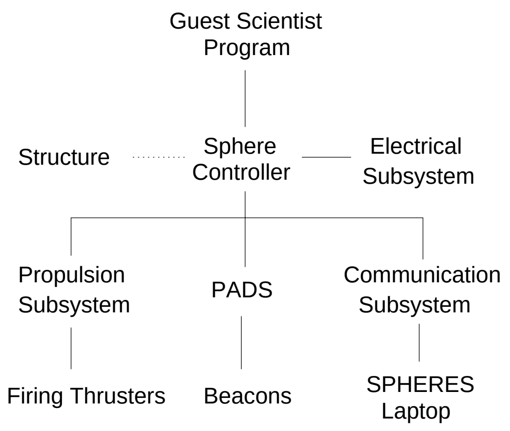


Figure 1: The Functional Structure of SPHERES

SPHERES implies that reuse of software could save enormous amounts of programming effort by the guest scientists.

Figure 1 shows the functional decomposition of SPHERES. The Sphere controller provides the overall coordination of the actions of the onboard components as well as determining the overall operating mode of the Sphere. Each Sphere controller provides the interface for and control of that Sphere's propulsion, position and attitude determination, communication, guest scientist program, electrical, and structural subsystems. Because the focus of the case study was on subsystems that contain both hardware and software, the structural and electrical subsystems were not modeled, although they easily could have been included.

Each Sphere receives attitude and position information from its PADS (Position and Attitude Determination Subsystem). The Sphere controller uses this information to calculate the position and orientation of each Sphere. The guest scientist program running on board a Sphere performs either state estimation, control calculations, or both to determine which control actions need to be performed to achieve a new position and/or attitude. Two guest scientist programs were created for this study: a rate damper and a rate matcher.

Sphere control is actuated through the propulsion subsystem and the Spheres coordinate action through their communication subsystems. Astronauts on the ISS load and unload programs and information to and from the Spheres through a laptop computer.

3.1 Modeling and Analysis of the SPHERES Components

The SpecTRM specification/modeling tool used in the case study is based on intent specifications [9]. Briefly, an intent specification differs from a standard specification primarily in its structure: the intent specification is structured as a hierarchy of "models" designed to describe the system from different viewpoints, with complete traceability between models (see Figure 2). Levels do not represent the more familiar refinement abstraction, but a *why* or *intent abstraction* with higher levels providing the rationale (why) for the lower levels. Rather than each level representing more detailed information than the higher levels, 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.

This structure is designed to facilitate several activities necessary for successful reuse. First, hyperlinks are used to map between levels and facilitate reasoning across hierarchical levels. They also implement the tracing of system-level requirements and design constraints to related design decisions, and vice versa, to explain why the design decisions were made. Note that the structure of the specification does not imply that the development must proceed from the top levels down to the lower 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, as argued above, is critical to successful reuse. The necessary design rationale information, 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 being captured and maintained in separate documents.

To avoid accidents and mission losses, reused components must be analyzed to determine whether they violate the design rationale, assumptions, and safety constraints 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 blackbox model specification level and above (Levels 1 to 3), where most of the original safety analysis is done.

During operations, if changes are made to any physical system component (or if software is to be reused in a different system), potential violation of assumptions underlying the original system design must trigger re-analysis of the software. To accomplish this goal, 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 in order to reduce the costs of revalidating correctness and safety. Intent specifications are designed to make that process feasible and practical.

Level 0

The top level (Level 0) of an intent specification provides a project management view and insight into the relationship between the plans and the project development status through links to the other parts of the intent specification. One problem in managing large projects is simply getting visibility into the progress of the project, particularly when a lot of software in involved. The project management level might include project plans, such as risk management plans, pert charts, system safety plans, etc., with embedded hyperlinks to the various parts of the intent specification where the plans are implemented.

Level 1

Level 1 is the customer view of the project and includes the high-level goals, contractual requirements (the *shall* statements or functional requirements), design constraints, environmental constraints and assumptions, operator and interface requirements, system hazard analyses and hazard lists, and system limitations. Links from these parts of the specification down to other levels provides understanding of design rationale and the ability to determine how the requirements,

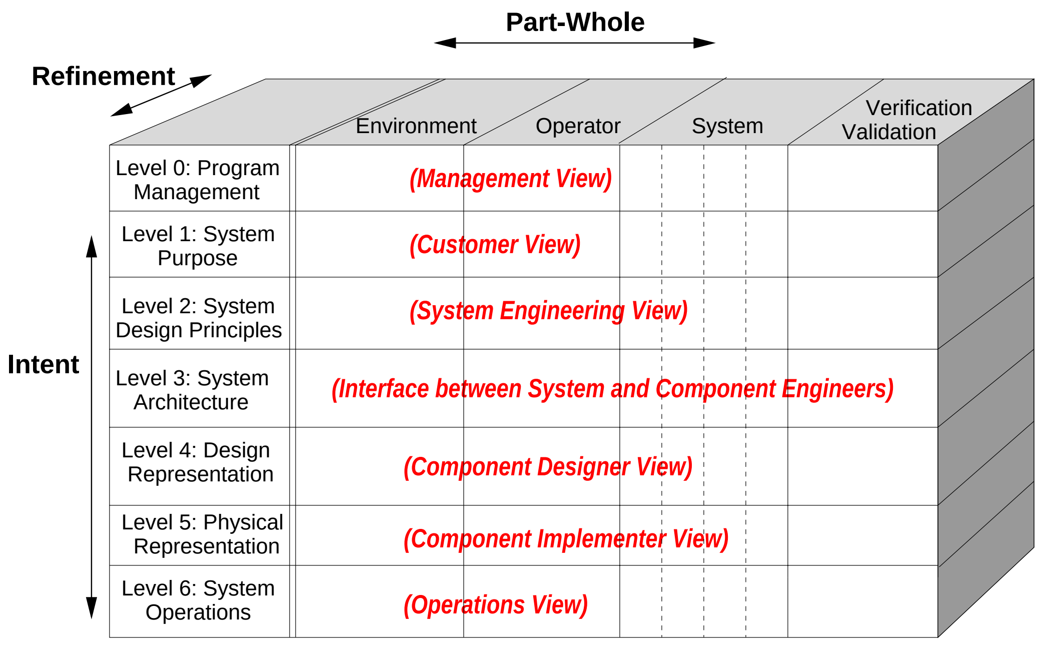
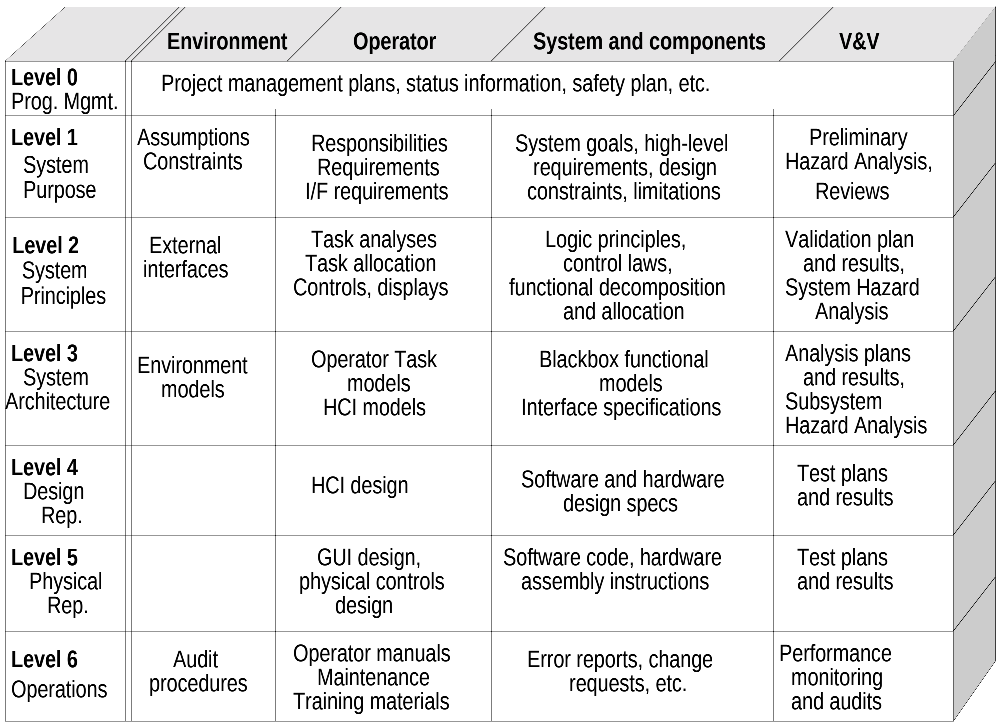

Part-Whole
Refinement
Environment
Operator
System
Verification Validation
Intent
Level 0: Program Management
(Management View)
Level 1: System Purpose
(Customer View)
Level 2: System Design Principles
(System Engineering View)
Level 3: System Architecture
(Interface between System and Component Engineers)
Level 4: Design Representation
(Component Designer View)
Level 5: Physical Representation
(Component Implementer View)
Level 6: System Operations
(Operations View)

Figure 2: The Structure of an Intent Specification



	Environment	Operator	System and components	V&V
Level 0 Prog. Mgmt.	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 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 System Architecture	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 3: Example Contents of an Intent Specification

environmental assumptions, and hazard analysis information are implemented in the system design.

An example requirement for the propulsion subsystem (referred to below) is:

> [FR.4] If force and torque vectors are received from the Sphere Controller, the propulsion subsystem shall determine on and off times for each thruster based on the thruster's location and the force and torque needed [↓ **DP.3.2**].
>
> > *Rationale: The guest scientists should have the ability to stock compute firing times for the thrusters based on desired force and torque vectors.*
>
> [FR.7] The Propulsion Subsystem shall include enough thrusters to provide actuation throughout the six-degrees-of-freedom [↓ **DP.1.3.**].
>
> > *Rationale: The SPHERES system is designed to operate in space and therefore must be able to translate in three dimensions and rotate in three dimensions.*

A propulsion subsystem hazard is:

> [H.1] A pressure rise in the propulsion subsystem above 4500 psi [← **SC.3**].
>
> > *Rationale: Such a pressure rise may result in an explosion that could either injure an astronaut or damage the Sphere.*

The safety constraint at Level 1 related to this hazard is:

> [SC.3] A mechanical system must be included in each Sphere that will mitigate a pressure rise in the propulsion subsystem [→ **H.1**, ↓ **DP.2.4**].

Level 2

Level 2 is the system engineering view and assists engineers in recording and reasoning about the system in terms of the physical principles and application design principles upon which the system design is based. It describes how the Level 1 requirements are achieved in the overall system design, including any "derived" requirements and design features not related to the Level 1 requirements, and how the Level 1 design constraints are enforced. It is at this level that the user of the intent specification can get an overview of the system design and determine why the design decisions were made, either by information included at this level or via the hyperlinks to Level 1. Examples from Level 2:

> [DP.1.3] The thrusters are arranged on the Sphere to provide pure body-axis force or torque using only two thrusters, assuming uniform mass and inertia properties. The twelve thrusters are arranged in six back-to-back pairs, allowing for full six-degrees-of-freedom actuation. Figure 2 shows the Sphere thruster configuration [↑ **FR.7**].
>
> > *Rationale: It is expected that the majority of maneuvers will involve primarily body-axis rotations, and the flight thruster geometry is significantly more propellant-efficient than other geometries for these maneuvers.*
>
> [DP.2.4] There are two pressure release mechanisms, or burst disks, in the propulsion subsystem. One is attached to the tank coupling and one is on the regulator itself. These mechanisms burst if the pressure builds to greater than 4500 psi [↑ **H.1**, **SC.3**].
>
> > *Rationale: The burst disks will rupture before the tank reaches a hazardous pressure of greater than 4500 psi, thereby releasing the pressure buildup.*
>
> [DP.2.5] The regulator is used to expand the liquid CO2 into a gas and simultaneously decrease the thruster feed pressure to between 0 and 35 psig.
>
> > *Rationale: At 35 psig, the average thruster force is approximately 0.1N, which is the desired operating thrust for each thruster.*
>
> [DP.3.2.1] The Pulse Modulation software calculates the duration that thrusters should be opened based on body-referenced force and torque vectors from the Sphere controller using the following equations . . . [↑ **FR.4**, ↓ **Thruster Pair 17 Calculation**, . . .].

Level 3

Level 3 is the System Architecture Level. It includes information about the allocation of the design decisions at Level 2 to individual system components (hardware, software, and operators) and the component blackbox behavior that implements those decisions. Level 3 in turn has links to the implementation of the specified behavior (i.e., behavioral requirements) in the design of the software, hardware, and operator procedures. If it is necessary to make a change to a component or to reuse it in a different system, it is possible to trace the function implemented by that component upward in the intent specification to determine its design assumptions, the requirements it is satisfying, related design principles, the design rationale, the safety-critical constraints enforced, and its role in the overall system design.

Level 3 essentially serves as an unambiguous interface between system engineers and component engineers. At Level 3, the system functions defined at Level 2 are allocated to components and specified rigorously. Blackbox behavioral component models are used to specify and reason about the logical design of the system as a whole and the interactions among individual system components without being distracted by component design and implementation details. We believe the models at this level are the most effective place to start most reuse efforts. For the case study, we created generic Level 3 models that can be instantiated for any particular system in which the component is to be reused.

The design of the formal language at this level, called SpecTRM-RL, is the result of lessons learned from the use of RSML and later variants of RSML on real projects and in laboratory experiments on specification language design, particularly lessons about error-prone features. For example, although internally broadcast events were included in RSML, they caused so many errors and so much difficulty in review of the models that they are not used in SpecTRM-RL. The primary goals for the new language were readability and reviewability (ability to find errors), completeness with respect to safety, and assisting with system safety analysis of the requirements.

SpecTRM-RL has a formal foundation (a simple state machine) so it can be executed and subjected to some types olenoid Valve 1 State

= Unknown

System Start	T	*
Propulsion Subsystem in mode Startup	*	T

= Open

System Start	F	F	F
Propulsion Subsystem in mode Direct	T	F	F
Direct Control Thruster 1 is On	T	*	*
Propulsion Subsystem in mode Force Torque	F	T	T
Thruster Pair 17 Calculation () > 0 nanoseconds	*	T	T
Time since Desired Thruster 1 State last entered Open < Thruster Pair 17 Calculation ()	*	T	*
Previous value of Desired Thruster 1 State in state Closed	*	T	T
Desired Thruster 1 State has never entered Open	*	*	T

= Closed

System Start	F	F	F
Propulsion Subsystem in mode Direct	T	F	F
Direct Control Thruster 1 is Off	T	*	*
Propulsion Subsystem in mode Force Torque	F	T	T
Thruster Pair 17 Calculation () = 0 nanoseconds	*	T	*
Time since Desired Thruster 1 State entered Open >= Thruster Pair 17 Calculation ()	*	*	T
Previous value of Desired Thruster 1 State in state Open	*	*	T

Figure 4: The AND/OR tables used to specify the logic under which the state variable *Solenoid Valve 1 State* is assigned the values *Unknown*, *Open*, and *Closed*. Tables evaluate to *true* (and the state transition is taken) if any of the columns in the table evaluate to *true*, i.e., each row of the column is true. Asterisks represent "don't care" conditions. For example, the first column in the logic describing when the *Solenoid Valve 1 State* will have the value *Open* says that the state variable should have this value if the propulsion subsystem is not in the startup state, its current control mode is *Direct Mode*, and it has received an input to turn the thruster *On*. When the propulsion system is in *Force Torque Mode*, it receives a force/torque vector from the Sphere Controller that specifies the forces and torques needed to accomplish a required maneuver. The Propulsion Subsystem then calculates how long the thrusters need to be on to achieve the actuation (the Thruster Pair 17 Calculation, the details of which are not shown). Solenoid Valve 1 stays open until the time since the valve was opened is greater than or equal to the calculated *Open* duration for the thruster.

of formal analysis, such as completeness and consistency analysis, while being designed to be readable with minimal training—the models can be read and reviewed by engineers and operators after about 10-15 minutes instruction. Figure 4 shows an example of the specification of detailed logic using AND/OR tables. These tables essentially incorporate formal propositional logic in disjoint normal form in a way that is easily reviewed (and written) by those not trained in formal logic. The tables can be machine parsed and executed.

We have also experimented with visualization to understand whether usability of formal specification languages might also be improved through the use of interactive visualizations automatically generated from the underlying formal model [1, 2].

Levels 4, 5, and 6

The Design Representation and Physical Representation levels provide the information necessary to reason about individual component design and implementation issues. Some parts of Level 4 may not be needed if at least portions of the physical design can be generated automatically from the models at Level 3. 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 operational environment envisioned during design and the *actual* operational environment. Level 6 provides a place to accumulate the operational information helpful in reusing the component in a different system or in a new use of the same system.

Levels 4, 5 were not created for the case study (although the Spheres code obviously exists) because they were not needed to evaluate the potential for reuse at the higher levels. Spheres operation has been delayed due to the Columbia accident (it was scheduled to go up to the ISS on the next shuttle flight after Columbia) so operational information is not available for Level 6.

3.2 Evaluation of Reuse in the Case Study

For the case study, Weiss produced a library of generic intent specifications (Levels 1, 2, and 3) that could be reused by the guest scientists to create new algorithmic test environments and by the SPHERES team itself for future versions of the system. The effort took about six person-weeks, including developing a simulation environment and an animation of the Spheres moving in space as the SpecTRM-RL models are "executed" together in a simulation environment.

The first version of the system included only one Sphere and a guest scientist program called a Rate Damper. Later, to demonstrate reuse, a second Sphere was assembled from the preexisting components and a second Rate Matcher guest scientist program integrated into the two-Sphere configuration—all in under an hour. The changes were then simulated to evaluate their correctness. A new version of the basic SPHERES platform is planned, and we will evaluate whether and how well the generic models and intent specifications assist with reuse of the original code.

4. CONCLUSIONS

This paper described a set of requirements we believe are necessary for reuse of embedded application software to be both practical and safe. A description of a case study using a commercial system engineering development environment on a real spacecraft followed. We believe that similar types of engineering environments, where reuse is attempted at the software behavioral requirements level rather than the code level, will allow engineering teams to tailor reused components and designs to fit their needs rather than requiring them to fit their needs to a particular piece of code. It also makes the process of validating and verifying the correctness and safety of reused software practical and, in turn, may allow more safety-critical software reuse. This hypothesis, of course, needs to be more thoroughly validated in realistic development environments.

5. ACKNOWLEDGEMENTS

This research was partially supported by NSF ITR Grant CCR-0085829 and NASA Engineering for Complex Systems Grant NAG2-1543.

6. REFERENCES

[1] Nicolas Dulac. *Empirical Evaluation of Design Principles for Increasing Reviewability of Formal Requirements Specifications through Visualization.* Master's Thesis, Aeronautics and Astronautics, MIT, August 2003.

[2] Nicholas Dulac, Thomas Viguier, Nancy Leveson, and Margaret-Anne Storey, On the use of visualization in formal requirements specification. *International Conference on Requirements Engineering*, Essen, Germany, September 2002.

[3] E.E. Euler, S.D. Jolly, and H.H. Curtis. The failures of the Mars Climate Orbiter and Mars Polar Lander: A perspective from the people involved. *Proceedings of Guidance and Control 2001*, American Astronautical Society, paper AAS 01-074, 2001.

[4] John L. Goodman. Lessons learned from flights of 'off-the-shelf' aviation navigation units on the Space Shuttle. *Joint Navigation Conference*, Orlando Florida, May 2002.

[5] John L. Goodman. A software perspective on GNSS receiver integration and Operation. *Satellite Navigation Systems: Policy, Commercial, and Technical Interaction*, International Space University, Strasbourg, France, May 2003.

[6] Kelly J. Hayhurst and C. Michael Holloway. Considering object-oriented technology in aviation applications. *Digital Avionics Systems Conference*, 2003.

[7] Charles Kruger. Software reuse. *ACM Computing Surveys*, Vol. 24, No. 2, June 1992, pp. 131–183.

[8] Ray Leopold. Personal communication. May 2004.

[9] Nancy G. Leveson Intent specifications: An approach to building human-centered specifications. *IEEE Transactions on Software Engineering*, Vol. SE-26, No. 1, January 2000.

[10] Nancy G. Leveson The role of software in spacecraft accidents. *AIAA Journal of Spacecraft and Rockets*, Vol 41, No. 4, July 2004.

[11] Nancy G. Leveson, Mats P.E. Heimdahl, Holly Hildreth, and Jon Damon Reese. Requirements specification for process-control systems. *IEEE Transactions on Software Engineering*, SE-20, No. 9, September, 1994.

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

[13] NASA/ESA Investigation Board. SOHO Mission Interruption. NASA, August 31, 1998.

[14] J.G. Pavlovich (Chair). Formal Report of Investigation of the 30 April Titan IV B/Centaur TC-14/Milstar-3 (B-32) Space Launch Mishap. U.S. Air Force, 1999.

[15] Linda Rosenberg. Personal communication. August 2001.

[16] Safeware Engineering Corporation. *SpecTRM User Manual.* 2003.

[17] Steven A. Stolper. Streamlined design approach lands Mars Pathfinder. *IEEE Software*, Vol. 16, No. 5, September/October 1999.

[18] Kathryn Anne Weiss. Component-Based Systems Engineering for Autonomous Spacecraft. Master's Thesis, Aeronautics and Astronautics, MIT, August 2003.

[19] Elaine Weyuker. Testing component-based software: A cautionary tale. *IEEE Software*, September/October 1998.