Reuse of a Formal Model for Requirements Validation

Robyn R. Lutz*

rlutz@cs.iastate.edu
Jet Propulsion Laboratory
California Institute of Technology
Pasadena, CA 91109-8099

Abstract

This paper reports experience from how a project engaged in the process of requirements analysis for evolutionary builds can reuse the formally specified design model produced for a similar, earlier project in the same domain. Two levels of reuse are described here. First, a formally specified generic design model was generated on one project to systematically capture the design commonality in a set of software monitors onboard a spacecraft. These monitors periodically check for faults and invoke recovery software when needed. The paper summarizes the use of the design model to validate the software design of the various monitors on that first project. Secondly, the paper describes how the formal design model created for the first project was reused on a second, subsequent project. The model was reused to validate the evolutionary requirements for the second project's software monitors, which were being developed in a series of builds. Some mismatches due to the very different architectures on the two projects suggested changes to make the model more generic. In addition, several advantages to the reuse of the first project's formal model on the second project are reported.

1 Introduction

In some application domains, successive software projects tackle many of the same problems. In such applications, software design from prior projects in the same domain or product family is sometimes used to guide the requirements for a current project. At the informal level this occurs when "Lessons Learned" on earlier projects are collected and considered during the requirements phase of a subsequent project. In other cases, reuse of existing software components or design patterns from a previous project may be mandated on a new project, with the reuse sometimes driving the requirements [20].

This paper investigates how a project engaged in the process of requirements analysis can exploit the formal design modeling and analysis done on a similar past project in the same application domain. The approach is outlined in Figure 1.

The paper describes two applications of reuse:

1. A formally specified design model was generated on Project 1 to systematically capture the design commonality in eighteen software monitors. This generic model was then reused to validate the software design of each of the eighteen monitors.

2. The formal model created for Project 1 (the Cassini spacecraft) was subsequently reused to analyze the requirements for similar software (i.e., fault monitors) on a second project in the same application domain.

   Each element (data item or function) of the generic formal model produced for Project 1 was either traced to the requirements for the monitors in Project 2 (the Deep Space-1 spacecraft) or the discrepant element was noted and investigated.

In addition to the anticipated benefit of validating requirements in the current build of Project 2, the work also clarified the allocation of requirements among the software elements, provided a structured way to capture design constraints and design assumptions during

---

*To be presented at the Fourth NASA Langley Formal Methods Workshop, Sept. 10–12, 1997. Mailing address is Dept. of Computer Science, Iowa State University, Ames, IA 50011. The research described in this paper was carried out by the Jet Propulsion Laboratory, California Institute of Technology, under a contract with NASA.

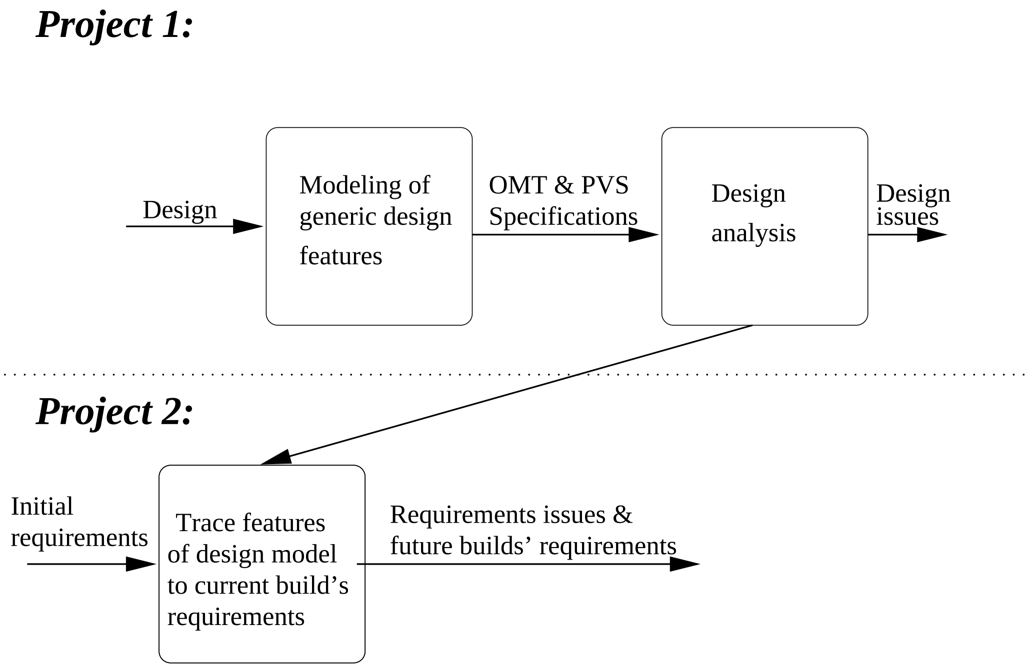

Figure 1: Reuse of Project 1's Formally Specified Design Model for Requirements Validation on Project 2

requirements analysis, and guided the identification of requirements to be added in later builds during the evolutionary software development process.

The need for rapid, low-cost requirements analysis and the planned, steady evolution of requirements on the new project motivated the reuse of the earlier design model. The goal was to import some of the lessons learned about system-level fault protection monitor design on the earlier project into the subsequent project in a structured but informal way. The results show that, although the software architectures and the development processes for the two systems are very different (see Section 4), the design model from the earlier project provided some guidance for validating a specific build's requirements and identifying future builds' requirements in the second project. By tracing each element of the earlier formal design model to the preliminary requirements for the later system, additional insights into the assumptions underlying the requirements, the design constraints of the new system, and the criteria for design choices were gained.

The assumption underlying the experimental reuse of the first project's formal model on the second project's requirements is that similar behavior and similar data must occur in each monitor in this domain. This assumption turned out to be largely true. The data items and functions in the formal specification can often be mapped directly to a data or behavioral requirement in the second project's monitors, adding some assurance that the requirements are complete and correct. This mirrored the experience applying the generic formal model to the eighteen monitors in the first project. In that case, flagging deviations of particular monitors from the norm (i.e., the generic model) was a quick way to identify areas of concern for additional analysis or testing. Similarly, identification of instances in which the second project's monitors deviated from the first project's generic formal model provided insights into currently missing requirements (to be required in later builds) and into implied design constraints.

Both projects involved fault-monitoring software in the same domain. Project 1 is the Cassini spacecraft, set for an October, 1997 launch to Saturn and its moon, Titan. Project 2 is the Deep Space-1 (DS-1) spacecraft, which will be launched in 1998 to rendezvous with an asteroid and a comet [7]. The software described here was, on both projects, the system-level fault-protection monitoring software. In the spacecraft domain, a monitor is software that periodically checks for malfunctions and initiates a process leading to recovery when appropriate. The monitors are the "eyes and ears" of the spacecraft [15].

In both projects the software monitors are required to display similar behavior (e.g. ignoring transient faults) and to use similar data (e.g., fault thresholds against which the input data is measured to determine if a fault exists). A fault is defined to be either "a defect in a hardware device or component" or "an incorrect step, process, or data definition in a computer program" [9]. Monitors are safety-critical software in that they must autonomously detect onboard threats to the spacecraft's health and mission. Since fault monitoring software in other applications' control systems frequently displays many of the same behavior and data dependencies represented in the formal model here, the approach described in this paper may have application beyond the spacecraft domain.

The rest of the paper is organized as follows. Section 2 describes the formal specification and analysis of the generic model. Section 3 summarizes the results from its use on the first project. Section 4 surveys some relevant commonalities and differences between the two projects in terms of their software development processes (waterfall vs. spiral) and their architecture (centralized vs. remote agents). Section 5 describes the results from the application of the design model to the requirements analysis of the software builds on the second project. Section 6 briefly discusses related work and future directions. Section 7 summarizes the lessons learned from the experience reported here.

2 The Formal Model

In previous work we used two technologies, formal methods and object-oriented modeling, to analyze the software design for portions of the Cassini spacecraft's software [1, 12]. The two tools that were used were OMT, the Object Modeling Technique [17], and PVS, the Prototype Verification System tool (SRI). PVS is an integrated environment for developing and analyzing formal specifications using support tools and a theorem prover [18]. These tools allowed the modeling, formal specification, and analysis of the monitors' design in the Cassini system-level fault protection software [11].

There are eighteen monitors in the system-level fault protection onboard the Cassini spacecraft. Eight of these are overtemperature monitors that are nearly identical in their logic. The other ten fault monitors detect loss of commandability (uplink), loss of telemetry (downlink), heartbeat loss (i.e, communication between computers), overpressure, overtemperature, undervoltage, and selected other failures.

These monitors share many of the same functions and attributes. One of the roles of the OMT models in the design analysis was to explicitly represent these common features in a way that could be readily reviewed by the Cassini engineers. This approach worked well. The OMT approach provides three viewpoints from which to represent the software design. The design of the fault protection monitor was thus represented in three OMT design diagrams [11]:

- The object-oriented design approach was represented in an object diagram. At the design phase, the object model of the monitor provided insight into the common behavior, properties, and relationships that the various monitors share. The OMT diagrams allowed the similarities in the monitors to be compactly represented.
- The functional viewpoint was represented in a data flow diagram. The data flow diagram characterized the data and data transformations common to all Cassini system-level fault protection monitors.
- The dynamic viewpoint was represented in a state diagram (Fig. 2). The state diagram for the design contained a sequence of six states that an active monitor can reach. Monitors commonly (1) test the validity of the input measurements that they receive from the sensors, (2) detect the existence of a fault condition in the various input data, (3) decide whether a fault condition in fact exists (perhaps by voting), (4) disregard transient anomalies, (5) determine whether a recovery response is appropriate, and (6) update state data, including possibly a request for a recovery response.

The formal specification in PVS of the design for the monitor software drew on the OMT diagrams to guide the formal specification of the design model. This was consistent with our earlier experience that creating OMT diagrams prior to formally specifying the requirements enhanced the accuracy of the initial formal specifications and reduced the effort required to produce them [12]; see also [2]. The formal specification in PVS of the design for the monitor consists of two theories (five pages of typechecked PVS specifications). The first theory, called `mon`, specifies the design of a system-level fault protection monitor (Fig. 3). Voting behavior (among inputs on whether a fault exists) was not represented in the formal model, although it would be a desirable addition.

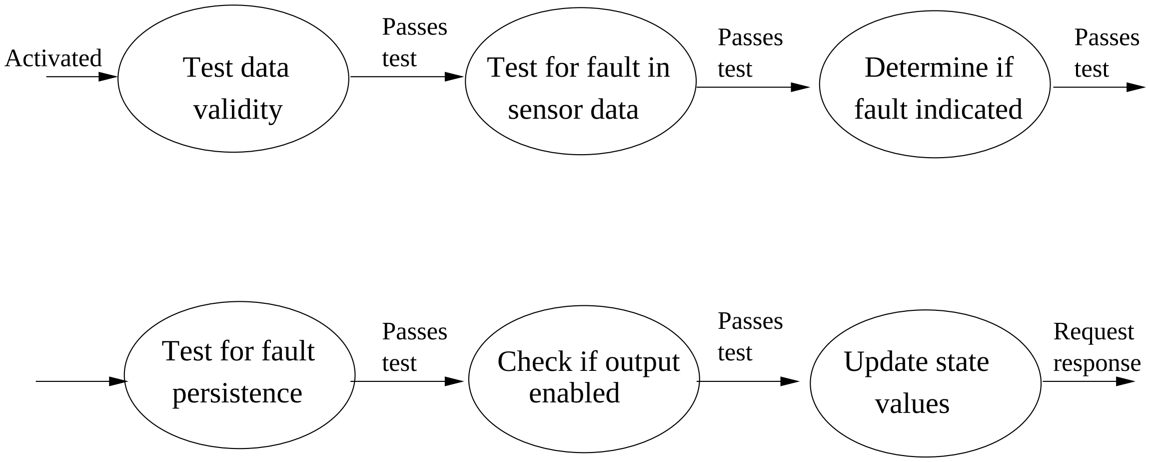

Figure 2: Dynamic Model

```
% Generic Cassini System-Level Fault Protection Monitor
% Monitor requests recovery response.

request_response (i, x, threshold, low_filter, high_filter, enabled,
                  prev_persist_ctr, persist_limit, sensor_input): bool =
                    valid_data_exists (i, low_filter, high_filter)
                       AND cond_exists  (i, x, threshold, low_filter,
                          high_filter, sensor_input)
                       AND cond_persists (i, x, threshold, low_filter,
                          high_filter, prev_persist_ctr, persist_limit,
                          sensor_input)
                       AND enabled
```

Figure 3: Excerpt from the Formal Model

The second theory, called `monlem`, states seven lemmas that specify the monitor's behavior. Two of these lemmas describe basic safety properties that the monitor is expected to obey, as documented in the project requirements ("A response is requested by a monitor only if the detected fault is not a transient fault" and "If a fault is not detected by a monitor, then the monitor doesn't request a response"). The other five lemmas concern the monitor's interfaces, i.e., the validity of the input data it uses to make a control decision as to whether to request a recovery attempt. The seven lemmas were proven, several by Martin Feather.

The conformity of the OMT representation and the formal specifications to the actual, final software design was checked against the eighteen system-level fault protection monitors in the post-Critical Design Review document [3, 4]. One step in evaluating that the model accurately represented the design was to classify the Data Lists provided in [3] for each of the monitors, and then to map those data classifications to the model. Toward this goal, the 162 data items in the Data Lists were classified into eight categories. In descending order of frequency, the eight categories were: Measurements (input data from sensors), Enabled/Disabled flags (monitors can be disabled), Thresholds (limits beyond which a fault condition exists), Filters (persistence requirements so that transient faults are ignored), State Updates (e.g., "high-water"—the highest measurement seen to date), Validity ranges (measurements outside these ranges are assumed to be from failed sensors), Heartbeat/Messages (from other software), and State Currently Commanded (information about the current configuration). The eight categories of data found in the document were reflected in the OMT design model and the formal specifications.

3 Reuse of the Formal Model on Project 1

The design analysis involved in constructing the model, in formally specifying the design of the monitor, in stating and proving the lemmas, and in confirming the accuracy of the model and specifications, identified eight deviations of Project 1's individual fault monitors from the formal specification of the generic design. Four of the eight discrepancies were found through the design analysis needed to develop the formally specified generic model. The other four of the eight discrepancies were found by comparing the Data Lists for each of the eighteen fault monitors with the data in the formal specification. The issues found were reported back to the Cassini project. In each case, either a decision was made to retain the current design for the reason listed below or the issue was still being worked when this phase of the analysis ended.

None of these discrepancies involved errors in the design logic, but several yielded discussions about the most robust way to design the software or how best to flag these items for attention during testing. Among the more interesting issues found by the analysis were:

- Unnecessary coupling. In a single case, a monitor cancels a response. This appears to be unnecessary coupling of the monitor and the response, since in all other cases, the Fault Protection Manager performs this supervisory function. Conversations with the software designers indicated that, since there is no known problem with the current unique design for this monitor, that it will be retained.
- Unique design. In a single case, a monitor disables itself (permanently or until ground operators interfere) if it even once receives bad data. In all other cases, the response disables its related monitor. The design ratonale is that past flight data indicates that once a transducer (the hardware component providing the data) provides unhealthy data, that it is likely to be unreliable from then on. However, other monitors do not cease to accept data from transducers that have once provided unhealthy data.
- Missing data validity checks? Inputs from the transducers on the propellant tanks and temperature transducers are checked for validity (i.e., whether the values are outside a range of credible values, which would indicate a failed transducer); however, inputs from other sensors do not undergo similar validity checks. At the time the study ended, this design issue was being worked as an action item by the design team.
- Misleading data name. In one monitor, the monitor can update the value of two global variables even when the monitor's "Enabled flags" indicate that its output is disabled. Thus, in this one case, a disabled monitor can produce output, contrary to what the flags' names suggest.

Several other discrepancies involved inconsistent documentation practices (e.g., a passed variable was in one case inaccurately listed as a global variable). By raising these issues while the design was still being defined, the Cassini design team was able to resolve them (by change or by documenting a rationale for the inconsistency) prior to implementation and testing.

The key benefit of abstracting from the documentation of individual monitors to model and formally specify the design of a general-purpose monitor in this study was to support design analysis. Flagging the design deviations of specific monitors from the general pattern was useful to the design verification process. These deviations are more likely to be design errors, more likely to be implemented incorrectly (because they are exceptions to the norm), and more likely to be overlooked in the selection of test cases than other software. For example, Software FMEA's (Failure Modes and Effects Analyses) that were being performed at the same time on the Cassini system-level fault protection software incorporated the design deviations found via the formal methods analysis into the SFMEA process [13].

4 Similarities and Differences in the Two Projects

The fault monitors on the two spacecraft have much in common. In both systems the fault protection software is divided into software monitors, that detect when a fault condition exists, and software responses, that take autonomous action to command the spacecraft to a known safe state.

Despite the similarities in the responsibilities of the fault monitors in the two systems, there are major differences in both the software architectures and in the development processes. System-level fault protection on Cassini is managed by a Fault Protection Executive that runs in a separate virtual machine. This is consistent with the basic fault protection design of prior spacecraft from which Cassini inherited portions of its fault-protection architecture. DS-1, currently under development, instead uses an innovative architecture based on recent advances in remote agent software, artificial intelligence, and robotics to monitor and recover from faults.

The software development process on Cassini is essentially a waterfall software development process tailored to the needs and constraints of the overall system development. The software development process on DS-1 uses an evolutionary model, similar to the spiral process model, with rapid development of an initial system and three subsequent builds to add incremental functionality. The requirements for a next build are derived in large part from the testing of the previous build. Consequently, the distinction between requirements and design is blurred. Risk identification during testing, and risk resolution during the definition of the next build's goals and constraints, drive the evolution of the requirements.

This evolutionary development of requirements, together with the shift from a familiar to a relatively new spacecraft architecture, motivated the decision to reuse the Cassini design model. Concerns with requirements completeness and consistency for each build's baseline could be addressed in part by reference to the previous project's formally specified generic model.

On DS-1, two monitors have been established as C++ classes which can be reused by various subsystems [5, 14, 15, 16]. The two monitors are a Threshold Monitor, which closely parallels the functionality of the generic Cassini monitor in detecting when a persistent fault occurred, and a Transaction Monitor, which reports on the status of a transaction (e.g, the successful or failed attempt to take a picture). On Cassini the generic monitor model was created strictly for independent design validation; on DS-1 the monitor class is incorporated directly into the implementation. The two DS-1 monitors are specified by means of state transition diagrams, supplemented by descriptions of the data. It is these specifications of monitors to which the Cassini generic design model was mapped.

One difficulty in mapping the generic formal model to DS-1's requirements was that the states in DS-1's monitors represent the software's knowledge of the hardware device it is monitoring, whereas the states in Cassini's monitors represent the monitor's own state of execution. This difference complicated the mapping since the same condition may lead to different states in the two systems. For example, on Cassini when the predicate "sensor surpasses threshold" becomes true, the software changes state from "Determine if fault indicated" to "Test for fault persistence." In DS-1 the same predicate causes the state to change from, e.g., the nominal state ("Looks-OK") to the state indicating that a fault may exist.

5 Reuse of the Formal Model on Project 2

Nine threshold monitors and seven transaction monitors are included in a recent, intermediate build of DS-1 [15, 16]. An early estimate was that there will be at least thirty monitor instances in the final launch code [6]. In addition, aggregate monitors will be composed from several Threshold Monitors.

The formal specification in PVS of the generic design model of the Cassini system-level fault protection monitors was used to build two tables, one for the data items in the formal model and one for the functions in the formal model. In these tables each element of the formal generic model (data and functions) was traced to the current DS-1 build's requirements for the Threshold and Transaction monitors.

The tables list nineteen elements of the design model (the external inputs to the monitor, the prior state inputs and next state outputs, the external outputs, and the functions). Some simple data items and functions needed only for correct PVS specification or to simplify proofs (e.g., a variable that specifies the ordinal number of an input within a set), but that were not present in the OMT models, are excluded from the tables. Figure 4 shows an example from the Data Table of the input data item, "persistence_limit". Figure 5 shows a sample function, "condition_persists". In the tables, "S" is a structure holding the internal monitor state, with S.Persistence being the limit on the number of unacceptable values, beyond which an error is declared; and S.Count being the number of unacceptable values [15]. (The actual tables are turned sideways with each row of the table describing a data item or function, and the Transaction and Threshold monitors described separately, but are reformatted here for readability of the excerpts.)

Those elements that are present in the Cassini design model but are neither in a current DS-1 build nor have a clear rationale for being excluded are candidates for software requirements for future DS-1 builds. On the other hand, because the monitors in DS-1 have less spacecraft redundancy to manage, some data and behavior present in Cassini monitors are not required on DS-1. For example, the DS-1 monitors do not receive redundant sensor data, so do not need the "num_sensors" data item present in the Cassini design model.

The following types of issues arose during the development and inspection of the tables:

1. *Evolutionary requirements.* The tables were most useful in identifying some requirements that needed to be added in later builds. The tables allowed the current status of the requirements to be tracked against the generic design model for similar software. The tables thus serve as a partial checklist against which the evolving requirements can be measured. The tables also allow some possible requirements for future builds to be inferred. For example, the collection of data required for ground diagnosis of the monitor's state was explicitly deferred to a later build (e.g., the collection by the Threshold Monitors of the highest and lowest values that each monitor ever sees for downlinked telemetry). The tables capture this intent for later builds' requirements.

2. *Validation of requirements in the current build.* Instances in which the DS-1 monitors were not required to exhibit behaviors that were present in the generic model were fed back to the project for confirmation. In most cases investigation yielded a rationale for the exclusion (e.g., there is not the same redundancy in sensor input on DS-1 as on Cassini, so DS-1 monitors do not need the "number of sensors" data item that the Cassini monitor used) or a better understanding of the spacecraft interfaces (e.g., that on DS-1 some lower-level filtering of the data occurs before the data are made available to the monitors). In a few cases deviations of a DS-1 monitor from the generic model led to continued discussions of some requirements decisions. For example, the use of dual thresholds ("too-high" and "too-low") in the Threshold Monitor raised questions about whether a single persistence counter suffices, since it masks whether the upper threshold, lower threshold, or some combination of both thesholds has been repeatedly surpassed.

3. *Clarifications of requirements allocation.* Given DS-1's innovative architecture and increased autonomy, there is an ongoing focus by the project on providing clean interfaces among software components and on avoiding the possibility that requirements drop through the cracks. The process of systematically either mapping each element of Cassini's design model elements to DS-1's monitors or of documenting why that element was not needed in DS-1 assists in this effort. The production of the tables prompted several questions about requirements allocation that crossed subsystem boundaries, leading to useful clarification by project personnel. Examples are where validity checks and noise filtering on input data are performed (externally or internally to the monitor); how outdated sensor inputs are handled; and where and under what conditions various data items should be reinitialized (e.g. should one good reading reset the fault counter to zero?).

4. *Design constraints/rationales.* The tables which trace Cassini's generic formal model of the fault monitor to DS-1's requirements for the monitors

Information for each Data Element	*Example of a Data Entry*
Data Item in Formal Specification	persist_limit
Data Type in Formal Specification	nat
Explanation	Fault must persist for a specified duration
Cassini Design Rationale/Assumptions	Transient faults should be ignored
Data Item in DS-1 Transaction Monitor	S.Persistence
Data Item in DS-1 Threshold Monitor	S.Persistence
DS-1 Build in which Implemented, and/or Rationale for Inclusion/Exclusion	Build x

Figure 4: Reuse of Formal Model for Requirements Validation of Data

Information for each Function	*Example of a Function Entry*
Function Name in Formal Specification	cond_persists
Cassini Design Rationale/Assumptions	Ignores transient fault condition
Function in DS-1 Transaction Monitor	S.Count >= S.Persistence
Function in DS-1 Theshold Monitor	S.Count > S.Persistence
DS-1 Build in which Implemented and/or Rationale for Inclusion/Exclusion	Build y; note that difference in trigger mechanism reflects individual requirements of the two monitors on DS-1

Figure 5: Reuse of Formal Model for Requirements Validation of Functions

document both the rationale and assumptions made by Cassini in adopting their design, and the reason for deviations of DS-1's requirements from that baseline. For example, on Cassini there was a design decision to allow monitors to remain enabled but to disable their output under some circumstances. The table notes that this distinction has no meaning for DS-1, where the design is constrained to handle monitors as subsystem function calls.

Initially an effort was made to trace each element in the OMT diagrams to the DS-1 monitors. This was useful for better understanding the DS-1 monitors, especially with regard to requirements allocation. However, the subsequent tracing of the PVS elements to the DS-1 monitors subsumed this earlier work. Moreover, the imprecision and repetition in the OMT diagrams was removed by focusing on the PVS specifications. The tracing of the OMT diagrams to the DS-1 monitors is thus omitted here.

6 Discussion

The discussion up to this point has primarily described the reuse of the formally specified generic model from the first project on the requirements validation of the second project's fault monitors. This effort at reuse was also instructive for validating the formal model itself and for pointing toward possible areas of future work.

6.1 Evaluating the formal model

There were data items in DS-1's monitors that did not appear in the generic design model, namely dual upper and lower thresholds for detecting fault conditions. On Cassini only one threshold was tested in any single monitor–e.g., overpressure and underpressure would be tested in separate monitors since different responses would be triggered. However, the deviation of the DS-1 Threshold Monitor from this pattern was a reminder that any extension of this generic design monitor to other applications should also handle a design decision to test dual thresholds in a single monitor.

Similarly, although the value of the fault persistence counter in Cassini never exceeds the value of the persistence limit, in DS-1's Transaction Monitor the counter can increase beyond this limit. Again, any further generalization of the generic design monitor should take this possibility into account. Most interesting is the inclusion in the DS-1 monitors of a confidence level, a number against which the number of successful or non-failed iterations of the monitor is compared. Too few successful iterations lead the monitor to report that it knows too little to accurately report the situation, i.e., the status is "unknown" [16].

Several of these mismatches between the second project's requirements and the model could be resolved by revisions or extensions to the formal model. Such work is currently underway by others in anticipation of reuse on subsequent projects.

Other mismatches are not readily resolvable due to the very different architectures on the two projects. For example, the second project's transaction monitors must report all changes, good or bad, rather than just faults, due to the increased on-board state modeling required by the remote agents. The formal model did a better job of validating the requirements for the Threshold Monitor on DS-1 than the Transaction Monitor on DS-1, since the Threshold Monitor's requirements were closer to the behavior of the Cassini monitors. This suggests that a set of generic models of fault monitors, rather than the single monolithic model developed for Cassini, may be needed if the generic model is to be reused for design (rather than requirements) validation.

Despite these limitations, the reuse of the formal model performed well in providing an early reasonableness check on the completeness and the correctness of the requirements for the second project's monitors. At the fairly high level of abstraction in the model, a high degree of correspondence between the required behaviors or responsibilities of the two project's monitors, and between the data they need to do their jobs, was evident. Required functionality currently deferred to later builds in the second project became evident and could be made explicit in the tables.

Architectural differences, which will lead to very different implementations of the fault monitors on the two spacecraft are, for the most part, masked at the model's high level of abstraction. A more detailed formal model might be less appropriate for reuse in requirements validation of the second project since it might be more likely to contain architectural dependencies.

Of the seven lemmas that were formally specified and proved in PVS for the Cassini generic design model, five involved the validity of the input data used for the control decision of whether to request a recovery response. All five of the properties specified in these lemmas (e.g., "The valid data from a sensor is within the range low-filter to high-filter") are required behavior of the lower-level "reflex" or fault detection behavior incorporated into each component in DS-1. One of the remaining two lemmas ("A response is requested by a monitor only if the detected fault is not a transient fault") describes required behavior of the Threshold and Transaction Monitors. The other lemma ("If a fault is not detected by a monitor, then the monitor doesn't request a response") describes required behavior of the Threshold Monitor but not of the Transaction Monitor. This is because the Transaction Monitor reports any change in status, including the first successful transaction after a string of failed transactions.

6.2 Related Work

Recent work in design patterns contains much in common with the process of defining and reusing the generic model described in this paper. Both approaches emphasize the specification of "the core of a solution to a recurring problem" [10]. However, the use of the generic model differs in two significant ways from the use of design patterns.

First, the reuse of the generic model provided a mechanism for requirements validation rather than a design mechanism. The formally specified and validated components of the generic model were traced to the components of the second project to check for gaps in functionality, robustness, and environmental assumptions. The generic model thus provided a reasonableness check on requirements rather than a design pattern for the new project's software.

Secondly, design patterns are often tightly linked to an architectural model. In contrast, one of the interesting features of the reuse of the generic model is that the second project had a significantly different architecture (remote agents) from the first project (centralized control). This is discussed in more detail in Section 4.

6.3 Future Work

Future work on generic monitors may involve identifying and specifying one or more multimission fault protection monitor for use in future spacecraft development. Since monitoring software similar to that on the spacecraft is part of many other safety-critical control systems, the monitor is also being investigated as a possible design pattern [19]. Possible benefits of such an effort include:

- Reducing design complexity. The specification of a generic design supports functional abstraction by identifying shared properties. It supports data abstraction by identifying the common objects and classes, and operations on them. By organizing similiar objects into classes and similar classes into superclasses, a generic design can help uncover underlying similarities and promote generalization and inheritance of shared attributes. In general, a common design specification keeps the internal logic of the individual modules as simple and general-purpose as possible.

- Encouraging gradual design refinement. Use of a generic design may encourage hierarchical design development (successive refinement). For example, some monitors vote on whether a fault exists, but the voting strategy (2 of 3, etc.) varies among the monitors. Design updates often change the voting strategy in a particular monitor. With a generic design the details of the different voting strategies can be cleanly deferred to the detailed design stage of each monitor.

- Tracking and evaluating design changes. The existence of a model and formal specification may allow more rapid response to proposed design changes by keeping the program structure evident. This might help avoid "design recovery" problems in the maintenance of existing software.

- Reducing test time by reducing coupling and increasing cohesion. General-purpose designs keeps the interfaces simple and the interdependence among modules minimal. The attention paid to abstraction creates more tightly bound modules with a clear sequence of tasks.

Work is underway to investigate whether a similar strategy for requirements reuse in a software product family offers these same advantages.

7 Conclusion

Formally modeling and analyzing the design of a generic fault protection monitor articulated the many commonalities among the data and functions of the eighteen Cassini software monitors. Having a formally specified design of a general-purpose fault protection monitor then allowed the occasional design deviations of individual monitors from the general pattern to be readily flagged for further analysis. This was useful because the discrepancies (1) may be design errors, (2) may need additional documentation of their design rationale (to preserve project awareness of their uniqueness), and (3) may require special attention during testing (since erroneous implementation of these exceptions from the pattern is easy).

The formally specified design model provided a baseline against which to measure the completeness of the requirements for similar fault-monitoring software on DS-1. Tracing data items and functions in the PVS design model to the requirements for the monitors in the second project helped (1) validate requirements in the current build, (2) identify requirements for future builds, (3) clarify the allocation of requirements among software components, and (4) document possible design rationales and constraints.

Acknowledgments

The author thanks Martin Feather, Gary Leavens, Judy Crow, and Jack Callahan for their insightful comments on an earlier draft of this paper, and Sarah Gavit, John Kelly, and Nicolas Rouquette for their assistance and suggestions. The work described in this paper was carried out at the Jet Propulsion Laboratory, California Institute of Technology, under a contract with the National Aeronautics and Space Administration.

Reference herein to any specific commercial product, process, or service by tradename, trademark, manufacturer, or otherwise, does not constitute or imply its endorsement by the United States Government or the Jet Propulsion Laboratory, California Institute of Technology.

References

[1] Y. Ampo and R. Lutz, "Evaluation of Software Safety Analysis Using Formal Methods", *Workshop for Foundation of Software Engineering (FOSE)*, Hamana-Ko, Japan, Dec, 1995.

[2] R. H. Bourdeau and B. H. C. Cheng, "A Formal Semantics for Object Model Diagrams," *IEEE Transactions on Software Engineering*, 21(10): pp. 799-821, October, 1995.

[3] *Cassini Orbiter Functional Requirements Book, System Fault Protection Algorithms*, CAS-3-331, Jet Propulsion Laboratory, June 7, 1995.

[4] *Cassini System Fault Protection Final Design Review*, Jet Propulsion Laboratory, Pasadena, CA, June, 1995.

[5] D. Dvorak, N. Rouquette, Q. Vu, "Monitors: How To Design, Build, Test," JPL internal posting, August, 1996.

[6] L. Fesq and D. Bernard, "DS-1 Fault Protection," in *New Millenium Interim Design Concurrence DS1 Autonomy/FSW,"* June, 1996.

[7] L. Fesq, A. Aljabri, C. Anderson, R. Connerton, R. Doyle, M. Hoffman, and G. Man, "Spacecraft Autonomy in the New Millenium," *Proceedings of the 19th AAS Guidance and Control Conference, Advances in the Astronautical Sciences*, ed. R. D. Culp, Breckinridge, CO, February, 1996.

[8] E. Gamma, R. Helm, R. Johnson, and J. Vlissides. *Design Patterns: Elements of Reusable Object-Oriented Software.* Addison-Wesley, 1995.

[9] *IEEE Standard Glossary of Software Engineering Terminology*, IEEE Std 610.12-1990, IEEE, New York, 1990.

[10] N. Islam and M. Devarakonda, "An Essential Design Pattern for Fault-Tolerant Distributed State Sharing," *CACM, Special Issue on Software Patterns*, 39(10): pp. 65-71, October, 1996.

[11] R. Lutz, "Design Analysis of Cassini Fault-Protection Montiors Using Formal Methods," JPL Document D-13431, May 1, 1996.

[12] R. Lutz and Y. Ampo, "Experience Report: Using Formal Methods for Requirements Analysis of Critical Spacecraft Software," *Proceedings of the 19th Annual Software Engineering Workshop*, NASA Goddard Space Flight Center, pp. 231-248, Greenbelt, MD, December, 1994.

[13] R. Lutz and R. Woodhouse, "Requirements Analysis Using Forward and Backward Search," *Annals of Software Engineering, Special Volume on Requirements Engineering*, forthcoming, 1997.

[14] "Mode Identification, Reconfiguration,