Experience Report: Using Formal Methods for Requirements Analysis of Critical Spacecraft Software

Robyn R. Lutz *
Jet Propulsion Laboratory
California Institute of Technology
Pasadena, CA 91109

Yoko Ampo †
NEC Corporation
Tokyo, Japan

May 15, 1996

Formal specification and analysis of requirements continues to gain support as a method for producing more reliable software. However, the introduction of formal methods to a large software project is difficult, due in part to the unfamiliarity of the specification languages and the lack of graphics. This paper reports results of an investigation into the effectiveness of formal methods as an aid to the requirements analysis of critical, system-level fault-protection software on a spacecraft currently under development. Our experience indicates that formal specification and analysis can enhance the accuracy of the requirements and add assurance prior to design development in this domain.

The work described here is part of a larger, NASA-funded research project whose purpose is to use formal-methods techniques to improve the quality of software in space applications [2]. The demonstration project described here is part of the effort to evaluate experimentally the effectiveness of supplementing traditional engineering approaches to requirements specification with the more rigorous specification and analysis available with formal methods.

The approach taken in this investigation was to:

1. Select the application domain. The primary criteria were, first, to select portions of the *requirements* of an large, embedded software project currently under development, and, secondly, to select *mission-critical software*, meaning that its failure could jeopardize the spacecraft system or mission.

   The selected applications were the requirements for portions of the Cassini spacecraft's system-level fault-protection software. This on-board software autonomously detects and responds to faults that occur during operations. About 85 pages of documented requirements describing the software that commands the spacecraft to a known safe state and a software executive that manages the fault protection were involved in the

*First author's mailing address is Dept. of Computer Science, Iowa State University, Ames, IA 50011.
†Second author's mailing address is Space Station Systems Division, NEC Corporation, 4035 Ikebe-cho, Midori-ku, Yokohama 226, Japan. This work was performed while the author was a visiting researcher at Jet Propulsion Laboratory, Pasadena, CA 91109. It appears in the *Proceedings of the 19th Annual Software Engineering Workshop*, Goddard Space Flight Center, Dec. 1994, SEL-94-006.

study. System-level fault protection was targeted as a domain which merited the extra assurance possible with formal specification and analysis.

2. Model the selected applications using object-oriented diagrams. The object-oriented modeling tool used in this work was Paradigm Plus, an implementation of OMT, the Object Modeling Technique [6] [1]. This effort built on earlier work in this research project in which OMT diagrams were found to be a useful complement to formal specification in a reverse-engineering application [1]. Our work differs in that we applied OMT to software currently in the process of being developed, with formal proofs as well as formal specifications being created.

3. Develop formal specifications. The formal specification language used in this study was that of PVS, the Prototype Verification System [8]. PVS is an integrated environment for developing and analyzing formal specifications including support tools and a theorem prover.

4. Prove required properties. We determined properties that must hold for the target software to be hazard-free and function correctly, specified them in PVS as lemmas (claims), and proved or disproved them using the interactive theorem-prover.

5. Feedback results to the Project. Because we were analyzing requirements that were still being updated, part of our task was to keep current with the changes and to provide timely feedback to the Project as they resolved the remaining requirements issues and began design development.

The experiment described here produced 25 pages of PVS specifications and 15 pages of OMT diagrams. 37 lemmas were specified. Of these, 21 were proven to be true and 3 were disproven. An additional 13 lemmas were stated but not proven. Five of these unproven lemmas were obviously true from the formal specifications; four were out of the scope of our application; and four remain to be proven. The lemmas that were proved were claims or challenges which must be true if the specifications are accurate and the requirements are hazard-free.

The lemmas were divided into three categories: requirements-met, safety, and liveness properties. Requirements-met lemmas traced the documented requirements to the formal specifications. For example, a documented requirement "If a response can be initiated by more than one monitor, each monitor shall include an enable/disable mechanism" led to a lemma demonstrating that the specifications satisfied this requirement. We proved or disproved 10 such requirements-met lemmas.

Safety properties were "shall-not" claims, which can be stated informally as "nothing bad ever happens [9]." Examples are, "The software shall not activate any response that is not requested by a monitor" and "The response shall not change the instrument's status during a critical sequence of commands." We were able to prove 7 such safety properties, adding assurance that the software did not introduce hazards into the system.

Liveness properties described the positive aspects of the correct behavior of the software: "something good eventually happens [9]." Examples are, "If a response has the highest

[1] Paradigm Plus is a registered trademark of Protosoft, Inc.

priority among the candidates and does not finish in the current cycle, it will be active in the next cycle" and "If the response occurs during a non-critical sequence of commands, then the instrument is turned on." We proved 7 such liveness properties, adding assurance that no hidden assumptions were required for the software to function correctly.

The results obtained from the specification and analysis (including proofs) of the requirements were of two types: issues found in the requirements and an evaluation of the process itself.

A total of 37 issues were found in the requirements. These were categorized as follows:

- Undocumented assumptions: 11. The formalization of the requirements revealed several assumptions that were not explicit in the documentation. An example of such an assumption is, "if the spacecraft is in a critical attitude, then the software is executing a critical sequence of commands." Frequently, these assumptions involved interface issues between software modules or subsystems, historically a frequent source of errors that persist until system testing [4]. In almost every case, the hidden assumption was currently correct. However, several assumptions merited documentation, especially since future changes can invalidate current assumptions.

- Inadequate requirements for off-nominal or boundary cases: 10. These issues usually involved unlikely scenarios in which a pre-condition could be false. We often had to consult spacecraft engineers to know whether such boundary cases were credible. For example, the case in which several monitors with the same priority level detect faults in the same cycle was not described. By concretely specifying the possibility of off-nominal scenarios, the formal analysis can contribute added robustness to the system.

- Traceability and inconsistency: 9. These issues included lack of traceability between the high-level requirements and low-level requirements, as well as inconsistency between the software requirements and the design of subsystems. Many of these issues were significant in that they could affect both the logic and the correctness of the formal specifications. An example is that although the high-level requirements assume that multiple detections of faults occuring within the response time of the first fault detected are symptoms of the original fault, the lower-level requirements (correctly) cancel a lower-priority fault response to handle a higher-priority response.

- Imprecise terminology: 6. These were documentation issues, frequently involving synonyms or related terms. The definition of types in PVS enforced their resolution.

- Logical error: 1. The logical error involved the handling of a request for service from a monitor in the case that a higher-priority request occurred. The question as to whether such a request could face starvation was first raised during the initial close reading. The formalization of the issue as a lemma which could be disproven provided insight and certainty.

The evaluation of the process we used to specify and analyze the requirements led us to three conclusions:

1. *Using object-oriented models.* For the target applications, object-oriented modeling offered several advantages as an initial step in developing formal specifications. First,

the object-oriented modeling defined the boundaries and interfaces of the embedded software applications at the level of abstraction chosen as appropriate by the specifiers. In addition, the modeling offered a quick way to gain multiple perspectives on the requirements. Finally, the graphical diagrams served as a frame upon which to base the subsequent formal specification and guided the steps of its development. Since the elements of the diagrammatic model often mapped in a straightforward way to elements of the formal specifications, this reduced the effort involved in producing an initial formal specification. We also found that the object-oriented models did not always represent the "why," of the requirements, i.e., the underlying intent or strategy of the software. In contrast, the formal specification often clearly revealed the intent of the requirements.

2. *Using formal methods for requirements analysis.* Unlike earlier work in this research project on software in which the requirements were very mature and stable and the formal specification entailed reverse engineering (Space Shuttle's Jet Select Subsystem), the work on Cassini's fault-protection subsystem analyzed requirements at a much earlier phase of development. Consequently, the requirements that we analyzed were known to be in flux, with several key issues still being worked (e.g., timing details, number of priority levels). A negative effect of the lack of stability was that time was spent staying current with changes. A positive effect was that issues identified during our analysis could be readily fed back into the development process before the design was frozen.

   We were concerned as to whether it was a waste of time to formally specify requirements while they were still likely to change. Certainly, there was inefficiency in rewriting specifications to conform to changes that occurred during the experiment. However, based on our experience with this trial project, the formal specification of unstable requirements had the following advantages:

   - Laid the foundation for future work.
   - Allowed rapid review of proposed changes and alternatives.
   - Clarified requirements issues still being worked by elevating undocumented concerns to clear, objective dilemmas.
   - Complemented the lower-level FMEA (Failure Modes and Effects Analysis) already being perfomed on the software, by providing higher-level verification of system properties.
   - Added confidence in the adequacy of the requirements that had been analyzed using formal methods.

   Rushby's recent study of formal methods for airborne systems reached the similar but even stronger conclusion that formal methods can be most effectively applied early in the lifecycle [7].

3. *Using formal methods for safety-critical software.* For a safety analysis it is important to ensure that a hazardous situation does not occur, as well as that the correct behavior

does occur [5]. Fault Tree Analysis, which backtracks from a hazard to its possible causes, is one method used for this kind of hazards analysis [3]. However, unlike formal methods of specification and proof, Fault Tree Analysis is an informal method which in practice permits ambiguous or inadequate descriptions.

Formal methods helped us find hazardous scenarios by forcing us to show every condition and prompting us to define new, undocumented assumptions. The process of developing formal specifications and proofs led us to think about the full range of cases, some of which were unanticipated.

In conclusion, one of the goals of the larger research project within which this investigation was performed is to evaluate the effectiveness and practicality of formal methods for enhancing the development process and the reliability of the end product. Our main contributions to this work in the Cassini demonstration project have been:

- Applying formal methods to the software requirements analysis of a project currently under development,
- Using object-oriented diagrams to guide the formal specification of software requirements,
- Formally specifying and proving a set of properties essential for the correct and hazard-free behavior of the software, and
- Demonstrating that formal methods can be used to specify and analyze an application involving critical software.

Acknowledgments

Other contributors to the formal methods work at Jet Propulsion Laboratory are Rick Covington, John Kelly, and Allen Nikora. Ken Abernethy contributed to this work while visiting JPL. The authors also thank Sarah Gavit and Jan Berkeley for helpful discussions.

The work described in this paper was carried out by 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 trade name, trademark, manufacturer, or otherwise, does not constitue or imply its endorsement by the United States Government or the Jet Propulsion Laboratory, California Institute of Technology.

References

[1] B. H. C. Cheng and B. Auernheimer, "Applying Formal Methods and Object-Oriented Analysis to Existing Flight Software," *Proc 18th Annual Software Eng Workshop 1993*, NASA/Goddard Space Flight Center, SEL, Dec 1993, 274–282.

[2] *Formal Methods Demonstration Project for Space Applications, Phase I Case Study: Space Shuttle Orbit DAP Jet Select*, JPL, JSC, and LARC, December 1993.

[3] N. G. Leveson, "Software Safety in Embedded Computer Systems," *Commun ACM*, 34, 2, Feb 1991, 35–46.

[4] R. Lutz, "Analyzing Software Requirements Errors in Safety-Critical, Embedded Systems," *Proc IEEE Internat Symp on Requirements Eng.* IEEE Computer Society Press, 1993, 126-133.

[5] *NASA Software Safety Standard*, NSS 1740.13, Interim, June, 1994.

[6] J. Rumbaugh, M. Blaha, W. Premerlani, F. Eddy, and W. Lorensen, *Object-Oriented Modeling and Design.* Prentice Hall, 1991.

[7] J. Rushby, *Formal Methods and Digital Systems Validation for Airborne Systems*, SRI-CSL-93-07, Nov 1993.

[8] N. Shankar, S. Owre, and J. M. Rushby, *The PVS Specification and Verification System*, SRI, March, 1993.

[9] J. M. Wing. "A Specifier's Introduction to Formal Methods," *IEEE Computer*, 23, 9, Sept 1990, 8–24.