Targeting Safety-Related Errors During Software Requirements Analysis

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

Abstract

This paper provides a Safety Checklist for use during the analysis of software requirements for spacecraft and other safety-critical, embedded systems. The checklist specifically targets the two most common causes of safety-related software errors: (1) inadequate interface requirements and (2) discrepancies between the documented requirements and the requirements actually needed for correct functioning of the system. The analysis criteria represented in the checklist are evaluated by application to two spacecraft projects. Use of the checklist to enhance the software-requirements analysis is shown to reduce the number of safety-related software errors.

I. Introduction

An earlier study of the causes of safety-related software errors found that those errors identified as potentially hazardous to a system tend to be produced by different error mechanisms than non-safety-related software errors [15]. Safety-related software errors found during the integration and system testing of two spacecraft arose most commonly from: (1) misunderstandings of the software's interfaces with the rest of the system, and (2) discrepancies between the documented requirements specifications and the requirements needed for correct functioning of the system.

A software error is defined to be a software-related discrepancy between a computed, observed, or measured value or condition and the true, specified, or theoretically correct value or condition [1]. A software error is classified as safety-related if, during the standard error-correction process, the systems safety analyst determines that the error represents potentially significant or catastropic failure effects.

This paper is part of an ongoing effort to improve system safety by directly targeting the known causes of safety-related software errors during the requirements phase. The main result of the paper is to provide a Safety Checklist for the analysis of software requirements that focuses specifically on interface requirements and robustness requirements.

Since system interface issues such as timing dependencies, storage capacities, noise characteristics, communication links, and expected operating environments are frequent sources of safety-related software interface errors, they need to be reflected in the software requirements specification. However, it is difficult to specify correctly the software/ system interfaces in complex, embedded systems with software distributed among various hardware components, some of which may be as-yet undetermined.

Similarly, many safety-related software errors involve inadequate software responses to extreme conditions or extreme values. Anomalous hardware behavior, unanticipated states, invalid data, signal saturation, and incorrect triggering of error-recovery responses are robustness issues which cause errors. By including requirements for robustness or what Neumann calls "defensive design" in the specifications, many safety-related errors can be avoided [18].

Jaffe, Leveson, Heimdahl, and Melhart present a set of criteria, defined in terms of an abstract state machine, to help find errors in the software requirements specifications of process-control systems [11]. They pay particular attention to the behavioral and robustness properties of control systems, making their work

*Author's 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.

an appropriate candidate for error reduction in safety-critical spacecraft systems.

Spacecraft involve embedded software distributed on several different flight computers. The spacecraft's software is safety-critical in that it monitors and controls components that can be involved in hazardous system behavior [14]. The possibility of hazardous interactions among the processes executing on different processors as well as the complexity of the timing issues across the system interfaces demand a rigorous analysis of the requirements.

This paper adapts and extends the criteria in [11] to the spacecraft domain. The work in [11] models only the controller in a process-control system. The adaptation of the criteria to spacecraft involves the consideration of additional features. Specifically, the prevalence of concurrent processes (often on distributed controllers), of redundant resources, of external command signals as inputs to the controller, and of state changes not visible in the feedback information are all features that complicate the spacecraft's requirements. The resulting Safety Checklist is shown to be useful in reducing safety-related software errors on spacecraft. It appears to be applicable to a variety of application domains involving safety-critical, embedded software.

The Safety Checklist developed here, unlike the criteria presented in [11], is appropriate for a software-development process that may not include formal specification languages or finite-state-machine modeling. The Safety Checklist described below thus is readily integrated into a wide range of software development environments.

Section IV of the paper applies the criteria represented in the checklist experimentally to two spacecraft projects. The results demonstrate a reduction in the number of safety-related errors due to inadequate software requirements regarding interfaces and robustness.

The overall goal of this work is to reduce safety-related software errors in future systems. The method is to focus, during requirements analysis, on those areas (software/system interfaces, failure modes, timing, boundary conditions and values) which in the past have caused errors that persisted until integration and system testing. The Safety Checklist has been developed as a tool to aid in this requirements analysis.

II. Related Work

Gray and Thayer [9] identify two key components of any software requirements methodology: (1) to aid in determining the software requirements and (2) to represent the software requirements specifications. The work described here is focused solely on the first of these two functions.

The paper describes a checklist by which developers can better identify and understand the requirements needed for embedded software to interact correctly with the system in all circumstances. Use of the checklist is consistent with a variety of representations of the software requirements. Regardless of the specification language or model chosen, all requirement methodologies suitable for safety-critical embedded software must include some way of confronting the issues identified in the Safety Checklist during the requirements phase. This work is thus integral to any effort to identify and eliminate software errors in safety-critical systems.

The Safety Checklist can be integrated into the requirements-analysis process as currently practiced for many application domains [4]. The checklist format is one that is widely used and with which developers are familiar. Formal inspections of requirement specifications, for example, commonly use checklists.

The utility of the formal inspection of requirements documents is widely documented. A study by Kelly et al., of 203 formal inspections, 23 of them inspections of software requirements documents, reports that a significantly higher density of defects was found during requirements inspections than in later development phases. [13]. Work by Doolan describes the savings and quality benefits resulting from the formal inspections of the requirement specifications of a large (2 million lines of code) package of seismic-processing software [5].

The Safety Checklist presented below is compatible with the software-requirements checklist used during formal inspections at Jet Propulsion Laboratory [6]. Overlap with the formal-inspection checklists has been eliminated to increase the usefulness of the Safety Checklist. The focus of the Safety Checklist is narrower than the formal-inspection checklists, since it concentrates on working backwards from common safety-related software errors discovered during system testing to their prevention in the requirements phase. The Safety Checklist is intended to extend the requirements analysis in directions that may enhance system safety, not to replace the current checklists, which are broader and more comprehensive in scope.

A wide variety of powerful formalisms exists to model and represent the specifications and behavior of systems [21]. In addition, much work has been done in recent years on formal specification languages. Timing constraints, which are a major source of software interface errors, often can be accurately modeled and interactively checked [2, 8, 10, 17, 19].

The capability to verify that the software requirements for a system satisfy the safety constraints on that system is a focus of much recent work [7, 12, 14]. Similarly, the capability to analyze specifications by proving theorems regarding them allows verification of the safety-critical functions of a system [3, 20].

The Safety Checklist provides a possible bridge mechanism from manual or CASE analysis of requirements to the formal specification and verification of safety-related software requirements. As formulated here, the checklist can provide a first step towards specifying safety constraints formally. The checklist's formal basis, as defined by Jaffe et al., allows the checklist to be written in terms of mathematical predicates in a variety of formal specification languages. It can then be tested against a formal specification of the requirements. The Safety Checklist thus can serve as a link between current informal practices and future, formal requirements analyses for safety-critical domains.

III. The Safety Checklist

The approach in [11] is to build a formal, finite-state model of the requirement specifications and then to analyze this model to ensure that its properties match the desired behavior (e.g., determinism). The authors accomplish this by stating criteria (usually formal predicates) that must hold in the model.

The Safety Checklist was developed as a translation of the criteria into an informal, natural-language format. Sometimes the translation is extracted from the text that accompanies the formal description in [11]. Other times the checklist item is a rewording of a mathematical predicate in a less-technical vocabulary. Though this rewording inevitably involves some loss of rigor and information, these are readily recaptured if the need arises by reference to the original article.

Formatting the requirements-analysis concerns as a checklist avoids the need previously to have built a model of the requirements. The checklist thus makes the interface and robustness issues (shown in Sect. IV to be powerful identifiers of future safety-related software errors) available to a wider range of software-development environments. In some applications the checklist may complement more formal approaches to the requirements analysis.

The Safety Checklist adapts the criteria in [11] to the spacecraft by taking into account the possibility of multiple processors, of concurrently executing processes, of redundant resources (including backup components) that must be managed, of externally commanded state changes, and of state changes not visible to the controllers. These features appear to be typical of many complex, embedded systems with timing constraints and safety-critical functions. The wording of the items in the Safety Checklist takes into account the associated interface and robustness issues for such systems.

Interfaces

1. Is the software's response to *out-of-range values* specified for every input?

2. Is the software's response to *not receiving an expected input* specified? (That is, are timeouts provided?) Does the software specify the *length* of the timeout, *when to start counting* the timeout, and the *latency* of the timeout (the point past which the receipt of new inputs cannot change the output result, even if they arrive before the actual output)?

3. If *input arrives when it shouldn't*, is a response specified?

4. On a given input, will the software always follow the same path through the code (that is, is the software's behavior *deterministic*)?

5. Is each input *bounded in time*? That is, does the specification include the earliest time at which the input will be accepted and the latest time at which the data will be considered valid (to avoid making control decisions based on obsolete data)?

6. Is a minimum and maximum *arrival rate* specified for each input (for example, a capacity limit on interrupts signalling an input)? For each communication path? Are checks performed in the software to avoid signal saturation?

7. If interrupts are masked or disabled, can *events be lost*?

8. Can any output be produced faster than it can be used (absorbed) by the interfacing module? Is *overload* behavior specified?

9. Is all data output to the buses from the sensors *used* by the software? If not, it is likely that some required function has been omitted from the specification.

10. Can input that is received *before startup, while offline, or after shutdown* influence the software's startup behavior? For example, are the values of any counters, timers, or signals retained in software or hardware during shutdown? If so, is the earliest or most-recent value retained?

Robustness

11. In cases where performance degradation is the chosen error response, is the degradation *predictable* (for example, lower accuracy, longer response time)?

12. Are there *sufficient delays* incorporated into the error-recovery responses, e.g., to avoid returning to the normal state too quickly?

13. Are *feedback loops* (including echoes) specified, where appropriate, to compare the actual effects of outputs on the system with the predicted effects?

14. Are all modes and modules of the specified software *reachable* (used in some path through the code)? If not, the specification may include superfluous items.

15. If a hazards analysis has been done, does every path from a hazardous state (a failure-mode) *lead to a low-risk state*?

16. Are the inputs identified which, *if not received* (for example, due to sensor failure), can lead to a hazardous state or can prevent recovery (single-point failures)?

IV. Results

Two applications of the Safety Checklist are described below. The first application looks at the safety-related software errors that were actually found on two spacecraft and evaluates whether use of the checklist during requirements analysis could have forestalled those errors. The second application uses the Safety Checklist to analyze part of a requirements document for safety-critical software.

A. Targeting Safety-Related Errors

The efficacy of the Safety Checklist is first analyzed by examining 192 safety-related software errors documented during integration and system testing of two spacecraft, Voyager and Galileo. Each of the 192 errors is classified according to which item, if any, in the Safety Checklist addresses the issue causing the error. Table 1 reports the results (see Appendix for tables).

Of the 192 errors, 149 have causes addressed by the checklist. This indicates that the checklist does, in fact, "ask the right questions." The usefulness of the Safety Checklist lies in its use as a prompter for better recognition of requirements. Asking the right questions during the requirements-analysis phase clearly is not sufficient to preclude the introduction of safety-related software errors into the system. However, since misunderstanding of the interface requirements and lack of detailed requirements for robustness are the primary causes of errors, asking the right questions seems to be a necessary condition for avoiding safety-related software errors in complex systems.

Table 1 shows that the issue most frequently involved in safety-related software errors is item 15, "Does every path from a hazardous state (a failure mode) lead to a low-risk state?" The prevalence of this issue reflects the fact that many of the safety-related software errors (20% on Galileo) involved the onboard autonomous error-recovery software. Some of the required error-recovery responses incorrectly included or omitted actions that allowed hazardous states to be entered or re-entered. Examples of such actions are turning off gyros, switching to backup memory, or disabling certain software processes in a particular mode. The additional analysis needed during the requirements phase to answer "Yes" to item 15 of the checklist might have precluded some of these errors.

The second most common issue producing safety-related software errors is item 12, "Are there sufficient delays incorporated into the the error-recovery responses, e.g., to avoid returning to the normal state too quickly?" Failure to recognize timing constraints such as the time required to complete recovery activity (e.g., to point the sensor at the sun), the delay required to avoid transient values (e.g., power transients or warm-up delays), or the correct persistence limit at which to trigger a response are common requirements inadequacies that cause subsequent interface errors [16].

Both the third and fourth most common errors queried by the checklist involve the arrival of input. The third most common error-producing issue is item 3, "If input arrives when it shouldn't, is a response specified?" This issue causes safety- related software errors when essential input is ignored. Often this involves subtle timing issues across the software/system interfaces (e.g., commands arriving before a process is in the correct mode to receive them, unexpected duplicate commands that are mishandled, or unforeseen race conditions).

The fourth most common issue is item 1, "Is the software's response to out-of-range values specified for every input?" This becomes a safety issue when error responses are erroneously triggered by incorrectly defined ranges or thresholds. Additionally, the lack of software requirements to handle large errors (e.g., large pitch disturbances, unexpected spin rates) caused several software errors on each spacecraft.

A related robustness issue is the fifth most common error, item 8, "Can any output be produced faster than it can be used by the interfacing module?" This item, together with item 6 (arrival rates), checks for erroneous assumptions regarding the possibility of, and appropriate response to, data overflow, signal saturation, and duplicate commands. By including requirements for overflow protection and out-of-range checks in the specifications, the subsequent design is more likely to be robust with regard to boundary conditions and values.

In all, eleven of the sixteen items on the Safety Checklist produced safety-related software errors. Those items not cited are either adequately handled during the development process with other methods (e.g., "Is all data used?" is checked by various means) or have not been documented as a problem with these particular systems (e.g., "Is the software's behavior deterministic?"). Of the safety-related software errors on the two spacecraft, 77% have their causes addressed by the Safety Checklist. The Safety Checklist thus appears to be useful for targeting the causes of safety-related software errors.

B. Analyzing Software Requirements

The Safety Checklist also was used to analyze part of a draft version of a Software Requirements Document for a spacecraft currently being developed. The portion chosen for analysis was the requirements specifications for data collection by the remote (distributed) engineering subsystems (e.g., power, propulsion, science and radio instruments).

This portion of the software specifications was chosen because each remote subsystem has many interfaces (both periodic and aperiodic) with other subsystems and because the safety-critical, error-recovery processes depend on the results of the data collection. The purpose of applying the Safety Checklist to these specifications was to evaluate the usefulness and ease-of-use of the checklist, not the correctness or completeness of what was provided as only a preliminary requirements document.

The data collection functions as follows. Each remote engineering subsystem receives various inputs over a shared bus from the central control computer as well as from other subsystems, performs certain actions in response to these inputs, and places various outputs (primarily engineering data) on a bus. Engineering data are gathered by each remote engineering subsystem and stored in its Bus Interface Unit's memory until the data are packaged and output on the bus to the central control processor. Some of the data also are extracted and sent to other subsystems. In addition, data required by the error-recovery processes are extracted and output separately.

Table 2 (see Appendix) shows the results of applying the Safety Checklist to the software requirements specifications for the remote engineering data collection. Six of the sixteen items in the checklist are thoroughly addressed in the preliminary software requirements document. Three of the sixteen items are explicitly deferred (since error-recovery responses and interrupt behavior are still being defined). The remaining seven of the sixteen items prompt additional questions involving the analysis of the requirements.

The seven items prompting additional requirements analysis do not necessarily need further specification in the document. Instead, they raise questions about possibly vulnerable areas ("what if's") and possibly hazardous circumstances. The questions raised by the checklist are useful in focusing the requirements-analysis process on the interface and robustness issues that have been shown to cause safety-related software errors in other complex, embedded systems.

For example, in accordance with the checklist, there is a requirement specified for data freshness. However, the timestamp in the header of the relevant data item records only the current time, making it uncertain whether obsolete data could be identified. The concern is not at this point with how the obsolete data could be identified, but with whether a requirement to identify obsolete data is in conflict with other requirements regarding header information. The checklist also allowed identification of possible race conditions, of possible starvation of low-priority data transfers, and of inputs which if not received might result in hazardous states (e.g., notification that error-recovery is underway).

Two extensions to the checklist were suggested by the application of the Safety Checklist to the software requirements specification.

1. *Data consistency.* When multiple copies of the same data items are kept, the possibility exists that the copies may have different values at any point in time. This inconsistency can occur through asynchronous update or through data corruption (e.g., as data is transferred across the bus or during a power-on reset response). This issue has significant safety consequences since error recovery often involves the management of redundant resources. This leads to the following extension to the Safety Checklist:
"Are checks for consistent data performed before control decisions are made based on that data?"

2. *Generic structures.* An important aspect of defensive design is that, as much as possible, modules and data objects should be generic, similar in format and in use. Special cases and exceptions increase the number of states and the opportunities for design er-

rors, especially during changes. In particular, restricting the number of possible hazardous states makes the validation of safety constraints more feasible. This leads to the following additional item for the checklist:
"Are generic structures used whenever appropriate to restrict the number of possible hazardous modes and states?"

V. Conclusion

The Safety Checklist has been shown to be useful in analyzing software requirements, particularly with regard to interfaces and robustness. By targeting those features which have proven to be the most common causes of safety-related software errors, the checklist contributes to a safer system. The criteria in [11] have been adapted to spacecraft software by taking into account the interface and robustness issues associated with the spacecraft's concurrent processes, interacting controllers, redundant resources, and externally commandable states. The resulting Safety Checklist aids in analyzing failure modes, in uncovering hidden assumptions and misunderstandings, and in identifying potential areas of vulnerability.

The Safety Checklist focuses extra attention on historically troublesome aspects of safety-critical, embedded software (timing dependencies, triggers for error-recovery responses, the handling of overload and saturation, the use of obsolete data for control decisions) without causing overspecification of well-understood or low-risk requirements. The checklist thus allows the depth of the requirements analysis to be tailored to the level of risk (technical or historical) associated with a component.

Because the checklist emphasizes requirements for software/system interfaces and robust responses to anomalous circumstances, many of the items it identifies are system hazards. It thus can be used as a first step towards specifying and checking safety constraints, either informally or formally. As developed here, the checklist can be readily incorporated into the requirements analysis, e.g., as a supplement to the formal inspection of requirements specifications.

Future work in this area will be directed at identifying how the use of the Safety Checklist during the requirements phase can be used to predict which factors in a particular system are likely to cause subsequent safety-related software errors.

Acknowledgments

The author thanks the referees and Linda L. Welz for remarks that improved the exposition of this paper.

References

[1] ANSI/IEEE Standard Glossary of Software Engineering Terminology. New York: IEEE, 1983.

[2] *Proceedings of the Berkeley Workshop on Temporal and Real-Time Specification.* Eds. P. B. Ladkin and F. H. Vogt. Berkeley, CA: International Computer Science Institute, 1990, TR-90-060.

[3] J. Cullyer, "Safety-critical Control Systems," *Computing and Control Engineering Journal*, Vol. 2, No. 5, Sept 1991, pp. 202–210.

[4] A. M. Davis, *Software Requirements, Analysis and Specification.* Englewood Cliffs, N.J.: Prentice Hall, 1990.

[5] E. P. Doolan, "Experience with Fagan's Inspection Method," *Software–Practice and Experience*, Vol. 22(2), Feb 1992, pp. 173–182.

[6] *S/W Development Formal Inspections Course.* Version H, Sept, 1992, D-8925, Software Product Assurance, Sect. 522, Jet Propulsion Laboratory.

[7] M. K. Franklin and A. Gabrelian, "A Transformational Method for Verifying Safety Properties in Real-Time Systems," in *Proceedings of the Real-Time Systems Symposium*, 1989, pp. 112–123.

[8] C. Ghezzi et al., "A Unified High-Level Petri Net Formalism for Time-Critical Systems," *IEEE Transactions on Software Engineering*, Vol. 17, No. 2, Feb 1991, pp. 160–172.

[9] E. M. Gray and R. H. Thayer, "Requirements," in *Aerospace Software Engineering, A Collection of Concepts.* Ed. C. Anderson and M. Dorfman. Washington: AIAA, 1991, pp. 89–121.

[10] T. A. Henziger, Z. Manna, and A. Pnueli,"Temporal Proof Methodologies for Real-Time Systems," in *Proceedings of the 18th ACM Symposium on Principles of Programming Languages*, 1991, pp. 353–366.

[11] M. S. Jaffe, N. G. Leveson, M. P. E. Heimdahl, and B. E. Melhart, "Software Requirements Analysis for Real-Time Process-Control

Systems," *IEEE Transactions on Software Engineering*, Vol. 17, No. 3, March 1991, pp. 241–258.

[12] F. Jahanian and A. K.-L. Mok, "Safety Analysis of Timing Properties in Real-Time Systems," *IEEE Transactions on Software Engineering*, Vol. SE-12, Sept 1986, pp. 890–904.

[13] J. Kelly, J. S. Sherif, and J. Hops, "An Analysis of Defect Densities Found During Software Inspections," *Journal of Systems Software*, Vol. 17, 1992, pp. 111–117.

[14] N. G. Leveson, "Software Safety in Embedded Computer Systems," *Communications of the ACM*, Vol. 34, No. 2, Feb 1991, pp. 35–46.

[15] R. Lutz, "Analyzing Software Requirements Errors in Safety-Critical, Embedded Systems," *Proceedings of the IEEE International Symposium on Requirements Engineering*, Jan 1993, pp. 126-133.

[16] R. Lutz and J. S. K. Wong, "Detecting Unsafe Error Recovery Schedules," *IEEE Transactions on Software Engineering* , Vol, 18, No. 8, Aug 1992, pp. 749–760.

[17] N. Lynch and H. Attiya, "Using Mappings to Prove Timing Properties," *MIT/LCS/TM-412.b*, Dec 1989.

[18] P. G. Neumann, "The Computer-Related Risk of the Year: Weak Links and Correlated Events," in *Proceedings of the Sixth Annual Conference on Computer Assurance*. NIST/IEEE, 1991, pp. 5–8.

[19] R. R. Razouk and M. M. Gorlick, "A Real-Time Interval Logic for Reasoning About Executions of Real-Time Programs," in *SIGSOFT '89 Third Symposium on Software Testing, Analysis and Verification*, Dec 1989, pp. 10–19.

[20] J. Rushby and F. von Henke, "Formal Verification of Algorithms for Critical Systems," in *SIGSOFT '91 Software for Critical Systems*, Dec 1991, pp. 1–15.

[21] J. M. Wing, "A Specifier's Introduction to Formal Methods," *Computer*, Vol. 23, Sept 1990, pp. 8–26.