NASA-IVV-97-015

NASA IV&V Facility, Fairmont, West Virginia

Experiences Using Lightweight Formal Methods for Requirements Modeling

Steve Easterbrook, Robyn Lutz, Rick Covington, John Kelly, Yoko Ampo and David Hamilton

October 16, 1997

This technical report is a product of the National Aeronautics and Space Administration (NASA) Software Program, an agency wide program to promote continual improvement of software engineering within NASA. The goals and strategies of this program are documented in the NASA software strategic plan, July 13, 1995.

Additional information is available from the NASA Software IV&V Facility on the World Wide Web site http://www.ivv.nasa.gov/

This research was funded under cooperative Agreement #NCC 2-979 at the NASA/WVU Software Research Laboratory.

To appear in IEEE Transactions on Software Engineering, special issue on formal methods in software practice, spring 1998

Experiences Using Lightweight Formal Methods for Requirements Modeling[1]

Steve Easterbrook
NASA IV&V Facility,
100 University Drive, Fairmont, West Virginia 26505,
steve@atlantis.ivv.nasa.gov

Robyn Lutz, Rick Covington, John Kelly
NASA Jet Propulsion Lab, Pasadena, California

Yoko Ampo
NEC Corp, Tokyo, Japan

and

David Hamilton
Hewlett Packard Corp, San Diego, California

Abstract

This paper describes three case studies in the lightweight application of formal methods to requirements modeling for spacecraft fault protection systems. The case studies differ from previously reported applications of formal methods in that formal methods were applied very early in the requirements engineering process, to validate the evolving requirements. The results were fed back into the projects, to improve the informal specifications. For each case study, we describe what methods were applied, how they were applied, how much effort was involved, and what the findings were. In all three cases, formal methods enhanced the existing verification and validation processes, by testing key properties of the evolving requirements, and helping to identify weaknesses. We conclude that the benefits gained from early modeling of unstable requirements more than outweigh the effort needed to maintain multiple representations.

[1] *The research described in this paper was carried out in part by the Jet Propulsion Laboratory, California Institute of Technology, under a contract with the National Aeronautics and Space Administration, and in part by West Virginia University under NASA cooperative agreement #NCC 2-979. Reference herein to any specific commercial product, process, or service by trade, name, trademark, manufacturer, or otherwise, does not constitute or imply its endorsement by the United States Government, the Jet Propulsion Laboratory, California Institute of Technology or West Virginia University.*

Experiences Using Lightweight Formal Methods for Requirements Modeling

Steve Easterbrook (NASA IV&V Facility, 100 University Drive, Fairmont West Virginia), **Robyn Lutz, Rick Covington, John Kelly** (NASA Jet Propulsion Lab, Pasadena, California), **Yoko Ampo** (NEC Corp, Tokyo, Japan) and **David Hamilton** (Hewlett Packard Corp, San Diego, California)

I. Introduction

In the development of embedded, mission-critical software there is a serious, unmet need for early feedback on the viability of a system in the requirements and early design stages [1]. The impact of early feedback on cost and safety has been demonstrated empirically. Boehm showed that errors are cheaper to fix the earlier they are detected in the development lifecycle [2]. In a study of 387 software errors found during integration and system testing, Lutz found that safety-related software errors arose most often from inadequate or misunderstood requirements. [3]. It is also clear that conventional techniques fail to catch many requirements errors [4]. However, studies have suggested that formal methods have tremendous potential for improving the clarity and precision of requirements specifications, and in finding important and subtle errors [5-7].

This paper presents three case studies of successful application of formal methods for requirements modeling. The studies demonstrate that a pragmatic, lightweight application of formal methods can offer a cost-effective way of improving the quality of software specifications. The studies concern the Verification and Validation (V&V) of fault protection software on the International Space Station and the Cassini deep space mission. The three studies share a number of features:

- Formal methods were applied in response to an existing development problem. In each case the problem was to provide an assurance that the fault protection requirements were correct. The informal techniques used on these projects had not been able to provide the desired level of assurance. Whilst the formal methods did not assure correctness, they improved the level of assurance by revealing errors that the informal techniques had missed.
- Formal methods were applied selectively. Only the most critical portions of the requirements were modeled, and only a selection of properties of these requirements were analyzed. The formal methods were applied by a research team working in parallel with the requirements analysts, rather than by the analysts themselves.
- In each case, formal methods offered a partial solution to the original problem. In particular, they provided a consistent requirements model, and revealed a number of errors, some of which had not been detected using inspection and traceability analysis. The studies increased the confidence in the requirements, but did not guarantee the completeness and correctness of the specifications. We argue that this is appropriate for early

modeling of requirements.

- In each case, the results of the study fed back into development process to improve the product.

We summarize observations on the utility of formal methods in these studies, and describe problems we encountered in applying them. Finally, we describe our current work exploring applications of formal methods in evolutionary design of new architectures for autonomous spacecraft control systems, and the special challenges of formally modeling evolutionary designs.

II. Background

1 Fault Protection

For NASA spacecraft, the term *fault protection* is used to describe system elements that detect and respond to perceived spacecraft faults. There are two main requirements when a fault occurs: the system needs to guarantee the completion of any time critical activities, and that the spacecraft is still safe, observable and commandable. Each spacecraft function has a pre-defined set of operating parameters, where each parameter has a normal operating range. Values beyond this range are *out-of-tolerance*. An out-of-tolerance condition may have many possible causes, so information from multiple sources must be combined to locate the fault. The normal operating range for each parameter is derived from the results of various system analyses, including failure modes and effects analysis (FMEA), hazard analysis, and safety analysis. These analyses also provide rules of inference for fault recovery.

Fault protection software initiates appropriate responses when out-of-tolerance conditions are detected in hardware and software components. Responses to loss of function include recovery (e.g. switch to a redundant backup), or retry (e.g. re-start a device in an attempt to restore functionality where no backup is available). Hazardous conditions generally require a *safing* response, to isolate the problem and minimize damage. For unmanned spacecraft, a typical safing response is to shut down all non-critical functions, ensure the antenna is pointing towards Earth, and await further commands. On Cassini, there is a requirement to be able to maintain such a safe state for up to two weeks. For manned spacecraft there is a possibility of crew intervention, so a further requirement is to isolate the fault to the smallest possible replaceable unit.

Because of the need to maintain a safe, habitable environment for the crew, fault protection on the space station has additional requirements over those for unmanned craft, and is referred to as Fault Detection, Isolation and Recovery (FDIR). Responsibility for FDIR is divided up into five layers, or *domains*. The lowest domain is the individual device. The next layer is the function that uses the device, followed by the subsystem and system control layers. The highest layer is manual FDIR. If a domain cannot provide FDIR for some conditions, a higher layer must provide it. For example, the subsystem layer, rather than the device layer, might handle an error condition involving the

interaction of two separate devices. Validation of the space station FDIR is particularly problematic, as FDIR functionality is distributed across many flight computers. The development and construction schedule for the space station does not permit full integration testing of the entire architecture prior to on-orbit assembly. Hence, FDIR functionality must be validated through a combination of inspection, simulation and analysis.

Fault protection operates asynchronously, and may be invoked at any time. Hence, the addition of fault protection software to a spacecraft system significantly increases the complexity of the software. An error in the fault protection software may compound an existing failure. This occurred during the launch of Ariane 5, when the fault protection software erroneously shut down two healthy processors, in response to an unhandled floating point overflow exception in a non-critical software function [8]. If the spacecraft is executing a critical function (e.g. an orbital maneuver) when the failure occurs, the fault protection must respond quickly to allow the critical function to proceed.

2 The Need for Formal Methods

Current requirements engineering processes within NASA rely extensively on informal processes, largely based on inspection. Inspection helps to remove a large number of specification errors, but cannot provide the desired level of assurance for the new generation of software-intensive spacecraft [4]. Remaining errors are detected throughout the lifecycle as the developers attempt to implement and test the system. There is a significant lack of effective methods and tool support for the requirements phase in comparison to those available for detailed design and coding.

The lack of rigorous requirements engineering techniques is well illustrated in the fault protection area. Fault protection requirements are more volatile than most other requirements, as they are sensitive to any change during the development of the primary system. Interactions between requirements can be hard to identify, let alone validate. Formal methods can help provide this validation in a number of ways. The process of formalizing a specification provides a simple validation check, as it forces a level of explicitness far beyond that needed for informal representations. Once a formal specification is available, it can be formally challenged [9], by defining properties that should hold, and proving that they do indeed hold. Formal challenges may be achieved both through theorem proving, and through state exploration or 'model checking'.

Rushby [9] points out that there is considerable scope for selective application of formal methods. Formal methods can be applied just to selected components of a system, and can be used just to check selected properties of that system. Most importantly, a great deal of benefit can be derived from formal methods without committing a project to the use of formal notations for baseline specifications. In the studies described in this paper, we used formal modeling to find errors in critical parts of existing informal specifications, but did not replace the informal specifications with their formal counterparts. We use the term 'lightweight' to indicate that the methods can be used

to perform partial analysis on partial specifications, without a commitment to developing and baselining complete, consistent formal specifications. This approach is also consistent with the advocacy of multiple representations as a way of overcoming analysis bias [10].

3 Methodology

The authors are (or were) members of a multi-center team within NASA, funded primarily by the NASA Office of Safety and Mission Assurance, to explore the potential of formal methods for increasing safety and reducing cost of mission-critical software [11, 12]. The team combines personnel with experience in formal methods, in the domains where formal methods are being applied, in software assurance and V&V, and in technology transfer. We have explored formal methods on a number of NASA programs, including Space Shuttle [6], Space Station [13, 14], and Cassini [15]. Throughout these studies, the emphasis has been on pragmatic application of formal methods in areas where there appears to be the greatest need. Experiences gained from these studies have been used to develop two NASA guidebooks [16, 17].

Although some development of the methods themselves has been necessary in order to fit them to our purpose, this has not been the main focus of the studies. Rather, we have concentrated on addressing issues such as:

- Can formal methods provide a cost-effective addition to existing techniques to improve the quality of requirements specifications?
- Can formal methods increase confidence in the validity of the requirements?
- Can early application of formal methods be beneficial even while requirements are volatile?
- How much effort is needed to apply formal methods, and what is the most appropriate process for applying them?
- Within any particular formal methods process, which activities require more effort, and which activities yield the greatest benefits?
- Which formal methods and tools are useful for which tasks?

In this paper we describe three studies that were implemented in the early requirements phase for new systems. These studies were responses to real needs on the projects. The requirements were often still volatile, and hence some effort was needed to ensure the formal analysis was kept up to date. Our goal was to demonstrate that formal methods could be applied and could add value in this context.

Although the three studies described here used different tools and notations, the basic approach was the same:

1) Re-state the requirements in a clear, precise and unambiguous format.
2) Identify & correct internal inconsistencies.

3) Test the requirements by proving statements about expected behavior.

4) Discuss the results with the requirements' authors.

The formal methods used in the studies were chosen according to need. PVS [18] was chosen for two of the studies, because it offers automated support for proof construction, and because the specification language appeared to be readily understandable to engineers and programmers. SCR [19] was chosen for the remaining study as it offered a tabular notation that corresponded well to the structure of the requirements, and provided tool support for consistency checking. In each study, an intermediate notation was used as a prelude to translating the requirements into the formal specification language. The first study used an annotated flowchart notation, the second used AND/OR tables [20], whilst the third used OMT (Object Modeling Technique) diagrams [21]. The intermediate notations helped to clarify ambiguities, and gain a better understanding of the structure of the requirements. This in turn helped to determine how the formal notation would be used.

Study 1: High level FDIR requirements for Space Station

This study was commissioned by the space station independent assessment[2] panel, who were seeking some assurance that the high level FDIR concept was clearly defined and validated, before detailed requirements were derived from it. Subsequent changes to the FDIR concept would have significant impacts throughout the requirements and design of the entire system. The study analyzed 18 pages of FDIR requirements, and was conducted over a period of two months, by two people working part-time. The total effort was approximately 2 person-months.

1 Approach

Three views of the FDIR had been documented: the functional concept diagram (FCD) which is a flowchart-like representation of the generic FDIR algorithm; baseline FDIR requirements; and capabilities, in which the requirements are grouped into related functional areas. This study concentrated on the first two of these views, developing a formal model of each, and testing traceability between them.

The four-step approach described above was used as follows:

1) The FCD was restated by abstracting out common features. The 53 processing steps of the original FCD were partitioned, in order to reduce the detail. For example, the first 12 steps check parameters for out-of-tolerance conditions, the next 7 deal with safing, the next 8 check for functional failure, and so on. Each step was labeled as one of three procedural categories: performing automated procedures, checking for anomalous conditions, and

[2] Independent assessment is an oversight activity, covering all aspects of the system, including hardware, software and operational procedures.

```
message: type =
{         parameter_OK,
          parameter_verified,
          safing_not_allowed,
          safing_executed,
          ...
}

% parameter is ok when its tolerance
% check has just ran and the parameter
% is OK (i.e. within tolerance)
rr_parameter_ok: axiom
          forall (t: tolerance_check):
          (  on(just_ran(t, time) and
             OK?(t(time)))
          iff
             record_check(time)(parameter_OK, t)
          )
```

Figure 1: Fragments of PVS specification, showing type definitions and axioms used to express FDIR concepts

recording/reporting results. Finally, six classes of condition under which control is passed to higher level FDIR domains were identified. The result of this initial analysis was a more structured (informal) model of the FDIR processes. This model was informally checked for reasonableness and for traceability to the original FCD. All the objects and attributes referenced in the FCD were then translated to PVS. Figure 1 shows two fragments of PVS generated at this stage.

The baseline requirements were then translated directly into PVS, using the definitions and types from the formalized FCD. This translation concentrated only on the FDIR system itself; we did not model the primary system that the FDIR monitors. Translation of these requirements into PVS proved to be relatively straightforward. Figure 2 gives an example.

2) The resulting definitions were typechecked using the PVS tool. Typechecking helped to eliminate several types of errors in the specification, including typos, syntax errors and type consistency errors.

3) The PVS specification was validated by using the PVS proof assistant to prove claims based on the specification. An example of such a claim is "at any domain level, if a failure occurs then it will always be recovered at some domain level". Although this claim was not very profound, several missing assumptions were detected in the

Requirement: automatic hazard and hazardous condition detection: ISSA shall automatically detect any out-of-tolerance condition or functional performance parameter that exhibits a time to catastrophic or critical effect of less than 24 hours.

```
automatic_hazard_condition_detection: axiom
          forall (p:parameter)
             param_out_of_tol?(p) AND time_to_effect(p)<24 =>
             exists(d:fdir_domain): detection(p,d) = automatic
```

Figure 2: An example FDIR requirement, and its PVS translation

process of proving it. For example, several sequencing constraints needed to be defined explicitly, even though the FDIR documentation states that no such constraints should be inferred from the requirements. A total of 14 claims were defined and proved.

4) A total of fifteen issues were documented and discussed with the requirements' authors. We had planned to explore traceability between the FDIR concept diagram and the baseline requirements. However, an initial analysis indicated that there was little traceability. The requirements' authors confirmed that the two documents expressed different kinds of requirements. The FCD describes the processing that is performed within an FDIR domain, while the baseline requirements describe a higher level view of the kinds of FDIR that must be provided.

2 Findings

In general, the FDIR requirements were well thought out. However, there was some question over whether the documentation was sufficient so that system developers and other stakeholders would understand them. Most of the fifteen issues were minor ambiguities, inconsistent use of terms, and missing assumptions, discovered during the process of formalization. These reduce the ability of developers to understand the requirements. For example, the distinction between the primary system and the FDIR system was not clear in the original requirements. Other ambiguities surrounded the use of terms such as "anomaly", "out-of-tolerance" and "functional failure". Three of the issues were classed as "high-major":

a) There were inconsistencies in the FCD over reporting the status of safing, recovery and retry procedures. The intention was that the FDIR processes should report their status before, during and after execution of each procedure. However, some of the procedures were missing requirements for some of the reporting activities, so that most of them did not have requirements to report status at all three points. This problem was detected during the initial reformulation of the FCD diagram.

b) The proper sequencing of FDIR processing is not clear from the FCD. Although the FCD looks like a flowchart, the accompanying text stipulates that it should not be interpreted as a sequential process. However, some important requirements can only be inferred by treating the flowchart as a sequential process. For example, it is not clear whether safing should be performed before isolation, although the diagram seems to imply it should be. This problem was detected during the proof process: some of the sequencing requirements had to be stated explicitly in order to prove necessary properties of the FDIR model.

c) No requirements are given for checking inconsistencies between parameters. The requirements only mention limit checking of individual parameters. The requirements team clearly intended that inconsistency checking should be included. This problem was discovered during the process of formalizing the baseline requirements.

```
(2.16.3.f) While acting as the bus controller, the C&C MDM CSCI shall set the
e,c,w, indicator identified in Table 3.2.16-II for the corresponding RT to
"failed" and set the failure status to "failed" for all RT's on the bus upon
detection of transaction errors of selected messages to RTs whose 1553 FDIR is
not inhibited in two consecutive processing frames within 100 millisec of
detection of the second transaction error if; a backup BC is available, the BC
has been switched in the last 20 sec, the SPD card reset capability is
inhibited, or the SPD card has been reset in the last 10 major (10-second)
frames, and either:
1. the transaction errors are from multiple RT's, the current channel has been
reset within the last major frame, or
2. the transaction errors are from multiple RT's, the bus channel's reset
capability is inhibited, and the current channel has not been reset within the
last major frame.
```

Figure 3: An example of a level 3 requirement for Bus FDIR. This requirement specifies the circumstances under which all remote terminals (RTs) on the bus should be switched to their backups.

Study 2: Detailed Bus FDIR requirements for Space Station

The purpose of this study was to analyze the detailed FDIR requirements associated with the bus controller for the main communications bus on the space station. These requirements represent a concrete implementation of the high level FDIR concepts addressed in the first study. The study was initiated by an Independent Verification and Validation (IV&V)[3] team. The IV&V team was having difficulty validating the bus FDIR requirements, as some of the properties that the IV&V team wished to test could not be established using existing informal methods.

The requirements for Bus FDIR are expressed in natural language, with a supporting flowchart showing the processing steps involved. The flowchart does not have the status of a requirement, but was merely provided for guidance; the intention was that the prose completely expressed the requirements (E.g. figure 3). The IV&V team had recommended that to improve clarity, the requirements should be re-written in a tabular form (E.g. table 1). This recommendation had been rejected because of the cost involved in re-writing them all. Hence, the IV&V team generated their own tabular versions, in order to facilitate the kinds of analysis they wished to perform.

The study analyzed 15 pages of level 3 requirements, and was conducted over a period of four months, by one person working part time. The total effort was approximately 1.5 person months.

1 Approach

The four-step approach was used as follows:

1) Each individual requirement was restated as an AND/OR table, to clarify the logic (see table 1). The generation of a tabular interpretation of each individual requirement proved to be hard, as there are a number of ambiguities

[3] IV&V is a practice in which a separate contractor is hired to analyze the products and process of the software development contractor [22]. The IV&V team reports to the Independent Assessment panel.

			OR		
	C&C MDM acting as the bus controller	T	T	T	T
	Detection of transaction errors in two consecutive processing frames	T	T	T	T
	errors are on selected messages	T	T	T	T
	the RT's 1553 FDIR is not inhibited	T	T	T	T
	A backup BC is available	T	T	T	T
A	The BC has been switched in the last 20 seconds	T	T	T	T
N	The SPD card reset capability is inhibited	T	T	•	•
D	The SPD card has been reset in the last 10 major (10 second) frames	•	•	T	T
	The transaction errors are from multiple RTs	T	T	T	T
	The current channel has been reset within the last major frame	T	F	T	F
	The bus channel's reset capability is inhibited	•	T	•	T

Table 1: The tabular version of the requirement shown in figure 3, showing the four conditions (the four columns) under which the action should be carried out. A dot indicates "don't care".

concerning the associativity of 'and' and 'or' in English, and the correct binding of subclauses of long sentences. For example, in figure 3, it is not clear what the phrase "in two consecutive processing frames" refers to. When the requirement shown in figure 3 was given to four different people to translate, we obtained four semantically different tables. By comparing these different interpretations, an extensive list of ambiguities was compiled. The ambiguities were resolved through detailed reading of the documentation, and questioning the original authors. This process also revealed some inconsistencies in the way in which terminology was used. The individual tables were then combined into a single SCR state-machine model (see table 2).

2) The SCR model was type-checked using the SCR toolset.

3) Properties of the SCR model were tested in two ways. Static properties of the state model, such as disjointness and coverage, were tested using the built-in checker in the SCR tool. Example properties are "for each combination of failure conditions, there is an FDIR response specified" and "for each combination of failure conditions there is at most one FDIR response specified". Dynamic properties of the model were tested by translating the SCR state machine model into PROMELA [23], and applying the SPIN model checker to explore its behavior. For example, some of the requirements express conditions to test whether various recovery actions have already been tried. These conditions were validated by exploring the dynamic behavior of the model in the face of multiple failures, and recurring failures. An example property is "if an error persists after all recovery actions have been tried, the bus FDIR will eventually report failure of itself to a higher level FDIR domain".

4) The findings were discussed with the IV&V team, and fed back to the development team through the normal IV&V reporting process.

2 Findings

In addition to a number of minor problems with inconsistent use of terminology, the following major problems were reported:

a) There were significant ambiguities in the prose requirements, as a result of the complex sentence structure. Some of these ambiguities could be resolved by studying the higher level FDIR requirements, and the specifications for the bus architecture. Some of the ambiguities that arose from the sentence structure could not be resolved in this way, and could lead to mistakes in the design. These ambiguities were detected in the initial reformulation of the requirements as AND/OR tables.

b) There was one missing requirement to test the value of the Bus Switch Inhibit Flag before attempting to switch to the backup bus. This was detected during the test for disjointness in the SCR specification.

c) The requirements were missing a number of preconditions that enforce the ordering of the inference rules. The accompanying flowchart for these requirements implied a sequence for these rules. An attempt had been made in the prose requirements to express this sequence as a set of preconditions for each rule, to ensure that all the earlier rules have been tested and have failed. The preconditions did not completely capture the precedences implied by flowchart. This problem was found during the test for disjointness in the SCR specification.

d) The timing constraints expressed in the requirements were incorrect. Several of the failure isolation tests referred to testing whether certain FDIR actions had already been tried "in the previous processing frame".

Current Mode	Conditions											Next Mode
	errors in two cons. frames	bus swch'd last frame	bus switch inhibit	bus swch'd this frame	backup BC avail.	BC swch'd in last 20 sec	card reset inhibit	card reset last 10 frames	errors from mult. RTs	channel reset last frame	channel reset inhibit	
Normal	@T	-	-	F	-	-	-	-	-	-	-	switch buses
	@T	-	T	F	-	-	-	-	-	-	F	reset the channel
	@T	T	-	F	-	-	-	-	-	-	F	
	@T	-	-	-	-	-	F	F	T	T	-	reset the card
	@T	-	-	-	-	-	F	F	T	F	T	
	@T	T	-	-	-	-	-	-	F	T	-	switch RT to backup
	@T	F	T	-	-	-	-	-	F	T	-	
	@T	T	-	-	-	-	-	-	F	F	T	
	@T	F	T	-	-	-	-	-	F	F	T	
	@T	-	-	-	T	F	T	-	T	T	-	switch BC to backup
	@T	-	-	-	T	F	T	-	T	F	T	
	@T	-	-	-	T	F	-	T	T	T	-	
	@T	-	-	-	T	F	-	T	T	F	T	
	@T	-	-	-	T	T	T	-	T	T	-	switch all RTs
	@T	-	-	-	T	T	T	-	T	F	T	
	@T	-	-	-	T	T	-	T	T	T	-	
	@T	-	-	-	T	T	-	T	T	F	T	

Table 2: An SCR Mode transition table. Each of the central columns represents a condition, showing whether it should be true or false; '-' means "don't care"; '@T' indicates a trigger condition for the mode transition. The four columns of table 1 correspond to the last four rows of this table. The semantics of SCR require this table to represent a function, so that the disjunction of all the rows covers all possible conditions (coverage), and the conjunction of any two rows is false (disjointness).

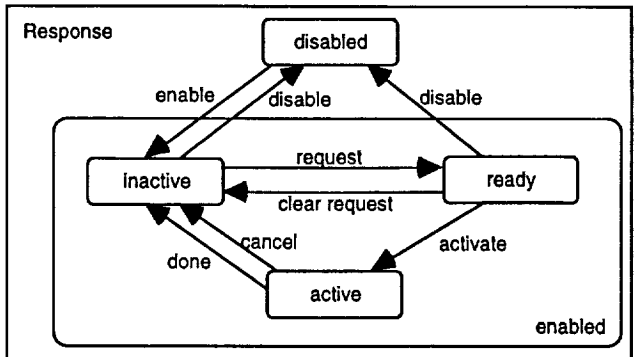

Figure 5: An example OMT state diagram for fault protection

However, as each FDIR recovery action is followed by a time-out while the action takes effect, and as further FDIR intervention is only initiated on occurrence of errors in two consecutive processing frames, these conditions can never be true. This was discovered during model checking of the PROMELA model.

Study 3: Fault Protection on Cassini

The third study concerns the system level fault protection software for the Cassini deep space probe. System reliability is a major concern for Cassini, due to the duration of its mission to Saturn. Fault protection is a major factor in providing the required levels of reliability. The study examined the requirements for the software executive that manages fault protection and requirements for putting the spacecraft into a safe state. The Cassini project was interested in the potential of formal methods to provide an assurance that the fault protection requirements were correct.

This study analyzed eighty-five pages of documented requirements. Fifteen pages of OMT diagrams [21] were produced, followed by twenty-five pages of PVS specifications. Twenty-four lemmas were proven. The study was conducted over the period of a year by two people working part-time, with a total effort of approximately twelve person-months.

1 Approach

The four-step model was applied as follows:

1) The first step was the production of OMT diagrams representing the prose requirements (see figure 5). The production of object diagrams, state diagrams and dataflow diagrams, according to the OMT method, helped to define the boundaries and interfaces of the fault protection requirements, and helped to crystallize some of the issues that arose in the initial close reading of the requirements. A PVS model was then produced directly from the OMT models – the elements of the OMT model often mapped onto elements of the formal model in a relatively straightforward way. For example, object classes mapped onto type definitions in PVS, while state

Cassini Requirement: If Spacecraft Safing is requested via a CDS (Command and Data Subsystem) internal request while the spacecraft is in a critical attitude, then no change is commanded to the AACS (Attitude and Articulation Control Subsystem) attitude. Otherwise, the AACS is commanded to the homebase attitude.

```
saf: THEORY
% Example is excerpted from saf theory.
% Spacecraft safing commands the AACS to homebase mode, thereby
% stopping delta-v's and desat's.
BEGIN

aacs_mode:  TYPE = {homebase, detumble}
attitude:   TYPE

cds_internal_request:   VAR bool
critical_attitude:   VAR bool
prev_aacs_mode:      VAR aacs_mode

aacs_stop_fnc (critical_attitude, cds_internal_request, prev_aacs_mode):
         aacs_mode =
         IF critical_attitude
            THEN IF cds_internal_request
               THEN prev_aacs_mode
               ELSE homebase
               ENDIF
            ELSE homebase
         ENDIF
```

```
aacs_safing_req_met_1:  LEMMA
         (critical_attitude AND cds_internal_request)
         OR (aacs_stop_fnc (critical_attitude, cds_internal_request,
            prev_aacs_mode) = homebase)

END saf
```

Figure 6: An example Cassini fault protection requirement, a fragment of PVS representing this requirement, and an associated 'requirements-met' lemma.

transitions mapped onto functions and axioms.

2) The PVS model was checked for internal consistency and traceability to the original requirements. Lemmas were defined to ensure that the model accurately captured the documented requirements. Figure 6 shows an example. The function expressed in this requirement is represented as part of the PVS theory for safing procedures. The requirement is also defined declaratively as a lemma, as a consistency check. Seven such lemmas were proved, and three disproved.

3) The PVS model was then checked for safety and liveness conditions. Safety lemmas represent conditions that should not arise. For example, "A fault protection response shall not change the instrument's status during a critical sequence of commands". Seven such lemmas were proved. Liveness lemmas ensure that required functions will eventually be performed. An example is "If a response has the highest priority among the candidates and does not finish in the current cycle, it will be active in the next cycle". Seven such lemmas were proved.

4) The results were discussed with Cassini project personnel. In some cases where requirements issues were still being worked by the project, the formal methods effort was able to assist by formalizing undocumented concerns (e.g., whether starvation of tasks would be possible) clearly and unambiguously. This facilitated rapid response to proposed changes or alternatives by the Cassini Project.

2 Findings

A total of 37 issues were identified during the study. These were classified as follows:

11 undocumented assumptions: None resulted in errors, but some significant ones needed documentation, to prevent future errors, especially at interfaces. These assumptions were identified during the process of formalizing the requirements.

10 cases of inadequate requirements for off-nominal or boundary cases: Such cases usually involved unlikely scenarios, and the spacecraft engineers had to help decide which were credible. An example case is when several monitors with the same priority level detect faults in the same cycle. Documentation of such cases is useful, as it helps to verify the robustness of the system.

9 traceability/inconsistency problems: The study uncovered a number of traceability problems between different levels of requirements, and inconsistencies between requirements and subsystem designs. Many of the latter were significant, as the correct functioning of the system depends on choosing the correct interpretation. For example, in the high-level requirements, the assumption is made that if multiple faults are detected within the response time of the first fault, they are all symptoms of the original fault. In the low-level requirements, a fault response will be cancelled if a fault of higher priority is detected, in order to handle the higher-priority fault.

6 cases of imprecise terminology: These were largely documentation problems, including synonyms and related terms. They were revealed during the process of defining the PVS model.

1 logical error: This was a problem of starvation when a request for service is pre-empted by a higher priority request. The issue was first spotted during initial close reading, and confirmed by disproving a lemma.

V. Discussion

The majority of published case studies of the use of formal methods are post hoc applications to on-going or finished projects. Such studies demonstrate what formal methods can do, and help to refine the methods, but they do not help to answer questions of how such methods can be integrated with existing practices on large projects. A few notable exceptions have used formal methods 'live' during the development of real systems [20, 24-26]. However, in all these cases, the emphasis was on the adoption of formal notations as baseline specifications, from which varying degrees of formal verification of the resulting design and implementation are possible.

In contrast, we applied formal methods only in the early stages of requirements engineering, during which the requirements were still volatile. Rather than treating formal specification as an end product of the requirements phase, we used it to answer questions and improve the quality of the existing specifications.

Our approach does not fit with any of the three process models suggested by Kemmerer [26] as ways of applying formal methods. Kemmerer offers three alternatives: *after-the-fact*, in which a formal specification is produced at the end of the development process to assist with testing and certification; *parallel*, in which formal specifications are developed alongside a conventional development process, and used to perform verification of code, design and requirements; and *integrated*, in which formal specification is used in place of conventional approaches. Our studies suggest a fourth model, in which formal modeling is used to increase quality during the requirements and high level design phases, without necessarily producing a baseline formal specification, or verifying low level design and code.

Our studies also demonstrate that questions of tool support need not be a barrier to the adoption of formal methods. We conducted sophisticated validation of our models, via theorem proving and model checking, using tools that are essentially still research prototypes. In the 12 case studies surveyed by Gerhart *et. al.* [25], tool support was generally only used for syntax checking of specifications, and Gerhart suggests tool impoverishment is a barrier to wider use of formal methods. This may be true for the more complete process models used in case studies of the kinds described by Kemmerer [26], Hall [24] and Gerhart [25], but is not true of the 'lightweight' application of the kind we adopted.

Although we have not attempted any detailed quantitative analysis of the costs and benefits of the application of formal methods in these studies, in each case the study added value to the project by clarifying the requirements and identifying important errors very early in the lifecycle. The costs, in terms of time and effort, were consistent with existing V&V tasks on these projects. Formalization of the requirements was the most time consuming part of the process, and in each case it revealed a large number of minor problems. Formalization also helps to focus attention on areas that are more susceptible to errors [27]. Consistency checking of the models was inexpensive, as it is largely done through automated typechecking. Formally challenging the models required a great deal of expertise, and it was often difficult to find suitable properties to test. This step uncovered a smaller number of more subtle errors, of the kind that are very hard to detect informally.

A number of observations arising from these studies are worth further discussion:

Who should apply the methods?

In each of the studies, the formal analysis was conducted by experts in formal methods, who were external to the development project. There was a simple reason for this: it is easier to bring in a small team of formal methods experts than it is to train members of the development team.

There are some interesting consequences of our use of external experts. Developing formal models of informal specifications involves a great deal of effort in understanding the domain, and figuring out how to interpret the documentation. As our external experts were unfamiliar with the projects prior to the studies, they did not share the assumptions that the requirements' authors had made. Our experts questioned everything, spurred on by the explicitness needed to build the formal models. They also needed to present parts of their models back to the developers, in order to check the accuracy of their interpretations. The result was a healthy dialogue between the developers and our formal methods experts. This dialogue exposed many minor problems, especially unstated assumptions and inconsistent use of terminology. This dialogue was clearly an important benefit.

Another aspect of this dialogue was that some of the issues that were raised were the result of misunderstandings by our experts, rather than genuine errors. The requirements' authors therefore had to filter the issues, to pick out those for which the benefits of changing the requirements out-weighed the cost. This was especially true when the analysis revealed "interesting" off-nominal cases. A great deal of domain knowledge was needed to judge whether such cases were reasonable. The need for such filtering would be greatly reduced if the analysis were conducted by domain experts; however, the risk of analysis bias would then increase.

Is formal modeling of volatile requirements worthwhile?

During early stages of the requirements process, there may be a great deal of volatility. In each case study, some effort was needed to keep the formal model up to date with evolving requirements. For example, in the second study new drafts of the requirements document were being released approximately every two months. In at least one case (study 2, finding c), the error had already been fixed by the time we discovered it. We mitigated the problem of fluctuating requirements by only doing the minimum amount of modeling necessary to test the properties that were of interest.

Our results indicate that there is no need to wait for the requirements to stabilize before applying formal methods. Early formalization allowed us to crystallize some of the outstanding issues, and explore different options. Most importantly, during this early phase the development team is more receptive to the issues raised from the formal modeling. This again emphasizes the importance of lightweight formal methods: the formal model itself can be discarded if the requirements change significantly, while the experience and lessons learned from it are retained.

Were intermediate representations useful?

Like Hall [24], we found that the use of intermediate, structured representations facilitated the process of formalizing the requirements. The type of intermediate representation varied across the studies: the first study used an annotated version of the original FCD flowchart, the second study made use of AND/OR tables to clarify complex predicates, while the final study made extensive use of OMT diagrams. A large part of the effort in the formalization process

lies in understanding the existing requirements. These intermediate representations helped to refine this understanding, and therefore reduced the effort needed to generate and debug the formal models.

The intermediate representations also helped to create some initial structure for the formal models. Since the elements of the intermediate representations often mapped directly onto elements of the formal specifications, the subsequent effort of formalization was reduced. This also facilitated traceability between the formal and informal specifications, making it simpler to keep the formal model current. For example, in the third study, the OMT diagrams offered multiple perspectives on the requirements, and were easy for project personnel to review for accuracy. In effect the OMT model provided a higher level structural view of the requirements, while the PVS models filled in the processing details, and allowed detailed behavioral analysis.

From our experience, it seems that this benefit more than outweighs the extra cost of maintaining several representations, at least for high levels of abstraction, even when requirements are still unstable.

VI. Conclusions

The three studies described here were conducted as pilot studies to demonstrate the utility of formal methods and to help us understand how to promote their use across NASA. An important characteristic of these studies is that in each case the formal modeling was carried out by a small team of experts who were not part of the development team. Results from the formal modeling were fed back into the requirements analysis phase, but formal specification languages were not adopted for baseline specifications.

We have shown that lightweight formal methods complemented existing development and assurance practices in these projects. If formal methods is seen as an additional tool in the V&V toolbox, then selected application to existing large projects becomes feasible.

As a follow-up to the studies described here, we have begun to investigate the role of formal methods in the development of new spacecraft technology. As part of NASA's New Millennium program, new architectures are being developed using knowledge-based systems to reduce the reliance of the spacecraft on ground support. Rather than produce a detailed statement of requirements, the project is using a rapid prototyping approach to explore the capabilities of the technology. The prototypes are tested against high level objectives, using a set of scenarios for guidance. We are exploring how to use lightweight formal analysis on rapidly changing information, in such a way as to provide useful and timely feedback. In particular, we are exploring the use of model checking to verify the fidelity between a formal model and the prototype. The model checker tests whether the formal model behaves in the same way as the prototype for a given scenario, while the formal model can be used to find interesting new scenarios on which to exercise the prototype.

Acknowledgements

The authors would like to thank Chris Jones, Sarah Gavit, Jan Berkeley, John Day, Jack Callahan, Chuck Neppach, Dan McCaugherty, John Hinkle, Larry Roberts, Alice Lee, Ernie Fridge, Kathryn Kemp, George Sabolish, Rick Butler and Alice Robinson for assistance in setting up these case studies and for helpful discussions as the work proceeded. We are also grateful to Ben Di Vito, Martin Feather, Frank Schneider, Judith Crow, Laurie Dillon, and the anonymous reviewers for their comments on earlier drafts of this paper. This research is partially a product of NASA's Software Program, an agency-wide program that promotes continual improvement in software engineering and assurance within NASA. The funding for this program is provided by NASA's Office of Safety and Mission Assurance.

Bibliography

[1] N. G. Leveson, *Safeware: System Safety and Computers*. Reading, MA: Addison Wesley, 1995.

[2] B. W. Boehm, *Software Engineering Economics*. Englewood Cliffs, NJ: Prentice-Hall, 1981.

[3] R. R. Lutz, "Analyzing Software Requirements Errors in Safety-Critical, Embedded Systems," *Proceedings of the IEEE International Symposium on Requirements Engineering*, San Diego, CA, Jan 1993.

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

[5] J. Crow, "Finite-State Analysis of Space Shuttle Contingency Guidance Requirements," Computer Science Laboratory, SRI International, Menlo Park, CA, Technical Report SRI-CSL-95-17, 1995.

[6] J. Crow and B. L. Di Vito, "Formalizing Space Shuttle Software Requirements," *Workshop on Formal Methods in Software Practice (FMSP '96)*, San Diego, California, January 1996.

[7] B. L. Di Vito, "Formalizing New Navigation Requirements for NASA's Space Shuttle," *Formal Methods Europe (FME '96)*, Oxford, England, March 1996.

[8] J. L. Lions, "ARIANE 5 Flight 501 Failure: Report by the Enquiry Board," European Space Agency, Paris 1996.

[9] J. Rushby, "Formal Methods and Their Role in the Certification of Critical Systems," Computer Science Laboratory, SRI International, Menlo Park, CA, Technical Report CSL-95-1, 1995.

[10] A. C. W. Finkelstein, D. Gabbay, A. Hunter, J. Kramer, and B. Nuseibeh, "Inconsistency Handling in Multi-Perspective Specifications," *IEEE Transactions on Software Engineering*, vol. 20, pp. 569-578, 1994.

[11] D. Hamilton, R. Covington, and J. C. Kelly, "Experiences in Applying Formal Methods to the Analysis of Software and System Requirements," *IEEE Workshop on Industrial-Strength Formal Specification*

Techniques (WIFT '95), Boca Raton, FL, April 1995.

[12] R. W. Butler, J. L. Caldwell, V. A. Carreno, C. M. Holloway, P. S. Miner, and B. L. Di Vito, "NASA Langley's Research and Technology Transfer Program in Formal Methods," *Tenth Annual Conference on Computer Assurance (COMPASS 95)*, Gaithersburg, MD, June 1995.

[13] D. Hamilton, R. Covington, and A. Lee, "An Experience Report on Requirements Reliability Engineering Using Formal Methods," *IEEE International Conference on Software Reliability Engineering*, France, October 1995.

[14] S. Easterbrook and J. Callahan, "Formal Methods for V&V of partial specifications: An experience report," *Proceedings, Third IEEE Symposium on Requirements Engineering (RE'97)*, Annapolis, Maryland, 5-8 January 1997.

[15] Y. Ampo and R. R. Lutz, "Evaluation of Software Safety Analysis using Formal Methods," *Foundation of Software Engineering '95*, Hamana-ko, Japan, Dec. 14-16, 1995.

[16] NASA, "Formal Methods Specification and Verification Guidebook for Software and Computer Systems. Vol 1: Planning and Technology Insertion," NASA Office of Safety and Mission Assurance, Washington DC, Report NASA-GB-002-95, 1995.

[17] NASA, "Formal Methods Specification and Analysis Guidebook for the Verification of Software and Computer Systems. Volume 2: A Practitioner's Companion," NASA Office of Safety and Mission Assurance, Washington DC, Report NASA-GB-001-97, 1997.

[18] S. Owre, J. Rushby, N. Shankar, and F. von Henke, "Formal Verification for Fault Tolerant Architectures: Prolegomena to the Design of PVS," *IEEE Transactions on Software Engineering*, vol. 21, pp. 107-125, 1995.

[19] C. L. Heitmeyer, B. Labaw, and D. Kiskis, "Consistency Checking of SCR-Style Requirements Specifications," *Second IEEE Symposium on Requirements Engineering*, York, UK, March 27-29, 1995.

[20] M. Heimdahl and N. Leveson, "Completeness and Consistency Analysis of State-Based Requirements," *IEEE Transactions on Software Engineering*, vol. 22, pp. 363-377, 1996.

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

[22] S. Easterbrook and J. Callahan, "Independent Validation of Specifications: A coordination headache," *Proceedings, IEEE Fifth Workshops on Enabling Technologies: Infrastructure for Collaborative Enterprises (WETICE'96)*, Stanford, CA, Jun 19-21 1996.

[23] G. J. Holtzmann, *Design and Validation of Computer Protocols*: Prentice Hall, 1991.

[24] A. Hall, "Using formal methods to develop an ATC Information System," *IEEE Software*, vol. 13, pp. 66-76, 1996.

[25] D. Craigen, S. L. Gerhart, and T. Ralston, "Formal Methods Reality Check: Industrial Usage," *IEEE Transactions on Software Engineering*, vol. 21, pp. 90-98, 1995.

[26] R. A. Kemmerer, "Integrating Formal Methods into the Development Process," *IEEE Software*, vol. 7, pp. 37-50, 1990.

[27] P. G. Larsen, J. Fitzgerald, and T. Brookes, "Applying Formal Specification in Industry," *IEEE Software*, vol. 13, pp. 48-56, 1996.