2011 Fourth IEEE International Conference on Space Mission Challenges for Information Technology

Building a Safety Case for a Safety-Critical NASA Space Vehicle Software System

Martin S. Feather
Jet Propulsion Laboratory
California Institute of Technology
Pasadena, USA
Martin.S.Feather@jpl.nasa.gov

Lawrence Z. Markosian
SGT, Inc.
NASA Ames Research Center
Moffett Field, USA
Lawrence.Z.Markosian@nasa.gov

***Abstract*—We describe our development of a key portion of a safety case for a safety-critical piece of NASA software designed to operate on a NASA launch vehicle. The software's purpose is to make real-time determinations of the presence of catastrophic failure conditions of that vehicle and react accordingly. We show how our safety case development applies a series of generic software considerations instantiated on the specifics of the NASA software system. We conclude that this approach is applicable to a wide range of NASA software systems.**

Software safety; safety cases; V&V

I. Introduction

Our work is motivated by the observation that safety cases are in widespread use in Europe and elsewhere, but have not taken hold within NASA. Our particular interest is in software systems. Both inside and outside NASA there has been advocacy for development of safety cases for software systems, but reports of actual applications are scarce.

Our goals are to determine the feasibility of developing safety cases for NASA's safety-critical space software, and to contribute guidance to help future developers of safety cases for similar software systems. Our first step was to develop an understanding of the state of the practice of safety cases. We found plenty of discussions of the concept of safety cases in the open literature. Our next step was to try to develop a safety case ourselves. We chose to develop a safety case for a significant portion of a safety-critical NASA software system. Our experience in doing this is described in this paper, which is organized as follows:

Section II briefly introduces safety cases.

Section III further describes the NASA context motivating our study.

Section IV introduces the safety-critical software system for which we are developing a safety case.

Section V shows key steps in our development of this safety case.

Section VI summarizes our results and conclusions to date, and suggestions for future work.

II. A Brief Introduction to Safety Cases

Safety Cases are used to manage and regulate major hazard industries (e.g., nuclear power, railroads, aviation, and offshore oil platforms) in Europe and elsewhere. Their origin traces back to the nuclear industry in the UK in the 1960's – [1] provides a succinct summary of their history. The following definition of a Safety Case is taken from the UK's Defence Standard 00-56 [2]:

"The Safety Case shall consist of a structured argument, supported by a body of evidence, that provides a compelling, comprehensible and valid case that a system is safe for a given application in a given operating environment."

Observe from this definition that a safety case is an *argument*, i.e., is intended for human understanding. The argument rests on *evidence* – both "direct" evidence (e.g., the results of tests, analyses, inspections) coupled with "backing" evidence to convey the trustworthiness of the direct evidence (e.g., that inspections were performed by trained personnel following accepted practices). The *structured* nature of the argument refers to its organization, necessary for presenting the case for the safety of a large and/or complex system. Overall, the argument must be *compelling* – it must convince people that a system is safe, *comprehensible* – understandable by people (the structured nature of the argument is important in this regard, so that humans can navigate and understand the safety case for a large and complex system), and *valid* – the argument must be consistent and complete, so that the safety claims of the system indeed follow from the structure of the argument and the evidence on which it based. The phrase *safe for a given application in a given operating environment* draws attention to the need to establish the context within which the safety case will establish that a system will be safe.

III. Safety Cases and NASA

Our objective is to ascertain whether safety cases are suitable for software systems used in space. Our initial interest in safety cases stemmed from several sources:

- NASA's Constellation program several years ago recommended (but stopped short of mandating) the use of "dependability cases" [1] for software projects. The recommendation said in part: "Each project should develop and maintain a Dependability Case to show, at different stages of a project's life cycle, how computing

[1] A *dependability case* is a generalization of safety case. It addresses the aspects of dependability that are relevant for a particular system, such as availability, reliability, safety, security, real-time performance, interoperability, etc. not all of which are necessarily safety-related. [3]

978-0-7695-4446-5/11 $26.00 © 2011 IEEE
DOI 10.1109/SMC-IT.2011.17

IEEE computer society

system dependability will be, is being, and has been achieved." [4]

- In the same timeframe, a National Research Council panel on "Software for Dependable Systems: Sufficient Evidence" addressed the question of "*How can software and the systems that rely on it be made dependable in a cost-effective manner, and how can one obtain assurance that dependability has been achieved?*" The panel's overall recommendation was to focus software certification and acceptance on the dependability case for the software [5].
- In 2007 Professor John Knight from the University of Virginia visited JPL, gave a presentation advocating use of "assurance cases" for critical software, and engaged in further discussions on the subject during his visit.
- Prior to the above, several members of CMU's Software Engineering Institute had advocated dependability cases for software [3]. They used a space software system as illustration, showing portions of a dependability case for that system (our understanding is that they developed their dependability case after the system had been implemented).

Despite these recurring expressions of advocacy from independent sources, we could not find any evidence of safety cases being developed, or planned to be developed, for NASA software systems. We attributed this to hesitation deriving from a lack of experience with safety cases within NASA, and more generally within the United States as a whole. Since no-one had constructed a safety case for a NASA software system, there was understandable reluctance to be the first to do so.[2] We also could find plenty of descriptions of the concept of safety cases, a few examples of actual safety cases[3], but almost no examples of safety cases that specifically address details of software's internals.[4] We recently found some helpful examples in [10].

In response, we crafted a proposal to NASA's Software Assurance Research Program to prototype the development of a safety case for a safety-critical NASA software system, and on the basis of that experience develop a NASA Safety Case Guide for evaluating the applicability of safety cases in NASA, and providing guidance and training to future NASA developers of safety cases.[5]

IV. FOCUS OF OUR STUDY – A NASA SAFETY-CRITICAL SOFTWARE SYSTEM

For our study of how safety cases can be developed we picked as our target a software subsystem to operate on board a NASA launch vehicle. The subsystem in question is the Abort Failure Detection, Notification and Response (AFDNR) system [11]. Roughly speaking, AFDNR's purpose is to make real-time determinations of the presence of catastrophic failure conditions of that vehicle from readings provided from sensors distributed throughout the space vehicle, and react accordingly. Dwight Sanderfer, lead of the AFDNR design effort at NASA Ames, provided us the following more thorough description of AFDNR's purpose:

> The AFDNR function is a part of the Ares I Flight Computer flight software designed to recognize Ares I conditions that require Orion to manually or automatically issue the abort command to initiate the abort sequence to separate Orion from the Ares I vehicle, and notify Orion of the abort recommendation with sufficient time to safely separate the Orion from the Ares I.
>
> Sensor data from the launch vehicle is first "qualified" before sending on to AFDNR – this step identifies failed sensors, and disqualifies their data; thereafter AFDNR only considers qualified sensor data.
>
> AFDNR monitors qualified vehicle sensor data against a defined set of abort triggers (measures of a system's state and/or behavior indicating an abort condition exists). Two abort triggers are defined for each abort condition; both are required for AFDNR to recognize this as the need for an abort.
>
> Upon recognition of the need for an abort, AFDNR notifies Orion (including crew), Mission Systems and Ground Systems (pre-launch only). Furthermore, if the abort condition is time-critical, AFDNR contains logic to autosafe the system [12].

Responses based on AFDNR's abort recommendations have potentially safety-critical implications. As per NASA's Software Safety Standard [13], AFDNR is therefore classified as *safety-critical* software ("Processes data … that lead directly to safety decisions…"). Our safety case study encompasses both the "qualification" of the sensor data provided to AFDNR and AFDNR's determination of whether there is an abort condition, but does not encompass AFDNR's response to abort conditions.

As described above, the scope of our safety case is AFDNR software. The scope is further limited primarily to the AFDNR algorithms (i.e., executable prototype). This focus was selected because we had the best access to the algorithm developers and design documentation, and because the *flight* software had not been developed to the point where a representative safety case could be constructed within our project resources.[6] In fact, all AFDNR software development

---

[2] Discussions with Constellation personnel conveyed the following arguments that arose in opposition to requiring safety cases: (1) no experience for estimating the level of effort was available to the Constellation program; (2) proponents of safety cases could not produce cost models; and (3) it was believed that the required safety analysis is already adequately performed, i.e., the added value of a safety case was not apparent [6], [7].

[3] A repository of safety cases is at [8].

[4] An exception is the NASA Ames' Robust Software Engineering group's research on automated generation of safety cases for model-based development [9].

[5] More recently, NASA's Aeronautics Research Mission Directorate established a research effort under the Aviation Safety Research Program to investigate the use of safety cases to support verification and validation of flight-critical systems. A number of such research projects have been funded under this research effort, which is expected to continue over a three-year period.

[6] This is *not* to suggest that safety cases need or even *should* be developed only when the flight software development is well underway. In fact, the *opposite* is likely to be the case, and the literature indicates that safety case

was significantly scaled back half-way into our project with the termination of the Constellation Program.

Given our project's scope, we focused on several documents related to AFDNR to support development of the safety case. We also consulted with AFDNR project personnel at NASA Ames Research Center.

The principal AFDNR resource used in developing the safety case has been the system definition and specifically the descriptions of AFDNR algorithms (and, to a lesser extent, sensor data qualification system (SDQS) functions). The flow diagrams for detection and confirmation functions serve as the specification for these functions, and were used in developing and testing the executable algorithms.

We also referenced the AFDNR project's mathematical framework used by project personnel to derive the allocation of false positives and false negatives from a module to its submodules given assumptions about common cause failures and the probability that the module will fail at a defined limit. Other relevant AFDNR references were the functional fault analysis (FFA) and the failure modes & effects analysis (FMEA). All of the resources mentioned in this paragraph contribute to the context for, but, in general, do not provide data for direct incorporation into, the software safety case.

Also relevant to the software safety case is the testing that was performed on the prototype algorithms. Information on testing was provided in conversations with software engineers who performed the testing.

For this paper we limit our description to aspects of AFDNR that have been previously published. This level of detail is sufficient for illustrating the points of this paper, but of course our actual safety case drives into greater detail of AFDNR itself.

V. Development of the Safety Case

A. System Level to Software Level

The first step in our construction of the safety case was to identify the system-level safety requirements relevant to the AFDNR software subsystem, make those the top-level goals of our safety case, and decompose them to get to the level of the AFDNR software itself.

As described in [11], the safety challenge pertinent to determination of an abort condition is: "… the need to reduce the false positive and false negative indications for abort. A *false negative* describes a situation where an abort is necessary to avoid loss of crew but is not detected by the system. A *false positive* describes an incorrect indication for the need to abort, where none is necessary. During launch and ascent, this could lead to loss of mission and loss of vehicle, and possibly loss of crew since the abort process is not without its own risk."

The launch vehicle's requirements state upper-bounds on the probabilities of false positives and false negatives. These are decomposed into requirements on the vehicle's major elements (e.g., first stage), leading to requirements on AFDNR's determination of abort conditions, also expressed as upper-bounds on the probabilities of false positives and false negatives. We mirror this decomposition in our safety case, leading to safety goals on AFDNR itself.

Our initial focus has been on the false positive goal. Our choice was driven by the knowledge that the vehicle itself is designed to be reliable, hence false positives are the predominant concern (by comparison, only during the infrequent occasions when an abort is necessary could a false negative occur). There is commensurately more complexity in the design of AFDNR to minimize the probability of false positives, increasing the interest in how a safety case might apply to this.

B. Quantifiable goals and the Software Safety Case Decomposition

Since the safety requirements are provided as probabilistic assertions, we began our work on the AFDNR safety case by examining a number of publicly-available safety cases and papers that used analogous assertions. In these safety cases, the high level safety requirements are stated in terms of the probability or frequency of injury or fatality. For example, the documentation for reduced vertical separation minima (RVSM) post-implementation safety case in European air space gives the following high-level safety requirements [15]:

> (i) the vertical collision risk in RVSM airspace meets the ICAO Target Level of Safety (TLS) of 5 x 10^{-9} fatal accidents per flight hour.
>
> (ii) The vertical collision risk in RVSM airspace due solely to technical height-keeping performance meets the ICAO TLS of 2.5 x 10^{-9} fatal accidents per flight hour.

The architecture requirements for the Constellation program include analogous high-level safety requirements on various Constellation mission classes and flight phases. These safety requirements are allocated to various systems, subsystems, etc. The mathematical framework for allocating false positives and false negatives in Ares is based on probabilistic risk assessment (PRA). While well-established for hardware and systems, PRA's applicability to software is questionable. (See, for example, [14] and [16]).

Regardless of the applicability of PRA and other methods of allocating false positive and false negative requirements, allocation of these requirements to and within software, and the verification of such requirements, was not performed by the AFDNR design and algorithm development team and we have not attempted to carry out such an analysis for the safety case.[7]

Another approach frequently used to address risk in safety cases is to use the "as low as reasonably practical" (ALARP) concept. [17] provides a safety case pattern for the argument. While this concept does not by itself fully address the question of allocating and verifying probabilistic safety requirements, it provides guidance for constructing the safety case argument in a way that allows experts to

development should be integrated with the other activities in the software development lifecycle [14]

[7] The need to do so is not specific to safety cases, but should be addressed in any software safety analysis.

judge whether a target risk level (*not* expressed quantitatively in the safety case) is achieved according to a specific argument strategy and evidence set, at what the experts consider "reasonable" cost. On the other hand, this approach has been criticized as unable to capture, in the safety case structure, the information required for a quantitative modeling and analysis [18].

Building on this objection, [19] factors the problem into a *safety argument* and a *confidence argument*, where the confidence argument is based on subjective probability. We believe that this is a promising area of safety case research. However, without the data to construct the confidence argument, we have focused on the safety argument in our AFDNR safety case.

C. *Generic concerns for a software system - corresponding step in the safety case*

We decompose the goal that AFDNR system causes sufficiently few false positives by considering each of a generic set of concerns for any software system that performs a calculation[8], and instantiating them on AFDNR. These generic concerns are:

- Interference: Nothing external to the software will interfere with the correctness of its calculations.
- Internals: The software itself performs its calculations correctly.
- Inputs: The software's inputs (upon which its calculations are based) are correct, or errors in those inputs will not cause the software to calculate incorrect results.
- Outputs: The results of the software's calculations are conveyed correctly and interpreted correctly wherever they are to be used.

When instantiated on AFDNR's calculation goal, these generic concerns become the following four subgoals:

- The Interference subgoal: "AFDNR's computing platform (including other software that might be executing upon it) does not interfere with AFDNR to cause it to generate false positives".
- The Internals subgoal: "AFDNR's calculations do not generate false positives".
- The Inputs subgoal: "AFDNR's inputs are correct or, if incorrect, will not lead AFDNR to generate false positives".
- The Outputs subgoal: "The results of AFDNR's calculations are not corrupted to, or misinterpreted as, false positives".

The Interference and Outputs subgoals quickly lead to consideration of the broader context in which AFDNR resides. For example, the Interference subgoal we take one step further and subdivide into a subgoal that the computing platform on which AFDNR executes will not cause AFDNR to generate false positives, and another subgoal that all other executing software will not cause AFDNR to generate false positives. Since detailed consideration of AFDNR's computing platform and other software lies outside the scope of our effort, we simply capture the dependencies of the AFDNR safety case on these externalities in our safety case as *assumptions* that their claims are true.

The Internals and Inputs subgoals form the heart of our safety case, since our focus is on the design of AFDNR itself. We look at each of these subgoals in the following subsections.

D. *AFNDR's inputs*

To address the goal "AFDNR's inputs are correct or, if incorrect, will not lead AFDNR to generate false positives" we first introduce a simple but necessary subgoal that "AFDNR's inputs are not confused with one another", and then consider each of the kinds of inputs that the software receives, introducing one subgoal for each kind. These give rise to quite different treatments in our safety case.

AFDNR has two kinds of inputs: the input indicating the current mission phase (e.g., a mission phase might be "ascent before first state separation has occurred"), and the sensor readings.

The first kind of input is the mission phase. To represent this in the safety case we add a subgoal "Mission phase input to AFDNR is correct, or if incorrect will not lead AFDNR to generate false positives". Further explanation of what correct means is linked to this subgoal. Mission phase input is relied upon by AFDNR to be correct as-is (i.e., AFDNR takes no action to validate such inputs). To represent this in the safety case we refine the previous subgoal into a stricter subgoal "Mission phase input is correct". The provider of this mission phase input is external to AFDNR, and hence outside our scope, so we attach an *assumption* to this subgoal asserting that the input is correct. This makes it clear that the safety case depends upon this assumption.

The second kind of input to AFDNR is sensor data. To represent this in the safety case we add a subgoal "Sensor inputs to AFDNR are correct, or if incorrect will not lead AFDNR to generate false positives". In contrast to the mission phase input, AFDNR does *not* assume these inputs are correct. Sensors are known to be fallible devices whose own failures lead to their reporting incorrect readings, and a significant portion of AFDNR's design is devoted to correctly discerning the state of the vehicle despite the possibility of incorrect sensor inputs. To represent this in the safety case we refine the previous subgoal into a stricter subgoal "Incorrect sensor inputs to AFDNR will not lead AFDNR to generate false positives". The AFDNR design achieves this by determining whether a sensor reading can be relied upon to be correct (a process referred to as "data qualification" in [21]), and excluding non-qualified sensor readings in its calculation of the condition of the vehicle. To represent this in the safety case we refine the previous subgoal into "AFDNR excludes incorrect sensor inputs from

[8] These generic considerations correspond to requirements found in prescriptive standards such as *IEEE Standard for Software Verification and Validation 1012-2004*, which states "Determine… that the software interfaces correctly with other software components in the system in accordance with requirements and that errors are not propagated between software components of the system." [20]

its calculation of positives”. We will consider how AFDNR does this in the next section.

Fig. 1 shows the portion of our safety case that decomposes the goal “AFDNR’s inputs are correct, or if incorrect, will not lead AFDNR to generate false positives” up to this point. This figure shows our decomposition diagrammatically, using the Goal Structured Notation developed at the University of York, UK [22]. The rectangular boxes denote subgoals, the oval (with a letter “A” to its bottom right) denotes an assumption, the rounded-ended boxes denote context (additional information), and the parallelogram-shaped box denotes a strategy (an explanation of how the subgoal above the parallelogram is refined into the subgoals below it). The small diamond below the bottom of the “AFDNR’s inputs are not confused with one another” indicates that there is further elaboration of the safety case is needed to address this goal.

Note that our entire safety case starts at the system level with the goal to show that the requirement placing an upper-bound on the probability of false positives is met by AFDNR. Fig. 1 shows a portion from the interior of the entire safety case.

We are using Adelard LLP’s ASCE™ software tool to edit and maintain our safety case. The figures in this paper are screenshots taken from our use of ASCE.

E. The “Calculate/Convey/Use” Software Safety Case Pattern

Excluding use of incorrect inputs from calculations obviously requires both recognizing when an input is incorrect, and then avoiding its use. As described in [11], in the architecture of AFDNR the recognition is done by a separate subsystem, called the “Sensor Data Qualification System” (SDQS) [21].

To represent this in our safety case we split the goal “AFDNR excludes incorrect sensor inputs from its calculation of positives” into *three* subgoals:

- “Incorrect sensor inputs are identified by SDQS” – this is the *calculation* of a result (of the correctness of sensor inputs),
- “Identifications of incorrect sensor inputs are conveyed from SDQS to AFDNR” – this is the *conveyance* of that result from where it is calculated to where it is used, and
- “AFDNR excludes sensor inputs identified as incorrect from its calculation of positives” – this is the *use* of the calculated result (note that the use made of this result is to exclude the incorrect sensor input from AFDNR’s calculations).

Fig. 2 shows this portion of our safety case. We believe that calculation/conveyance/use is a simple yet useful

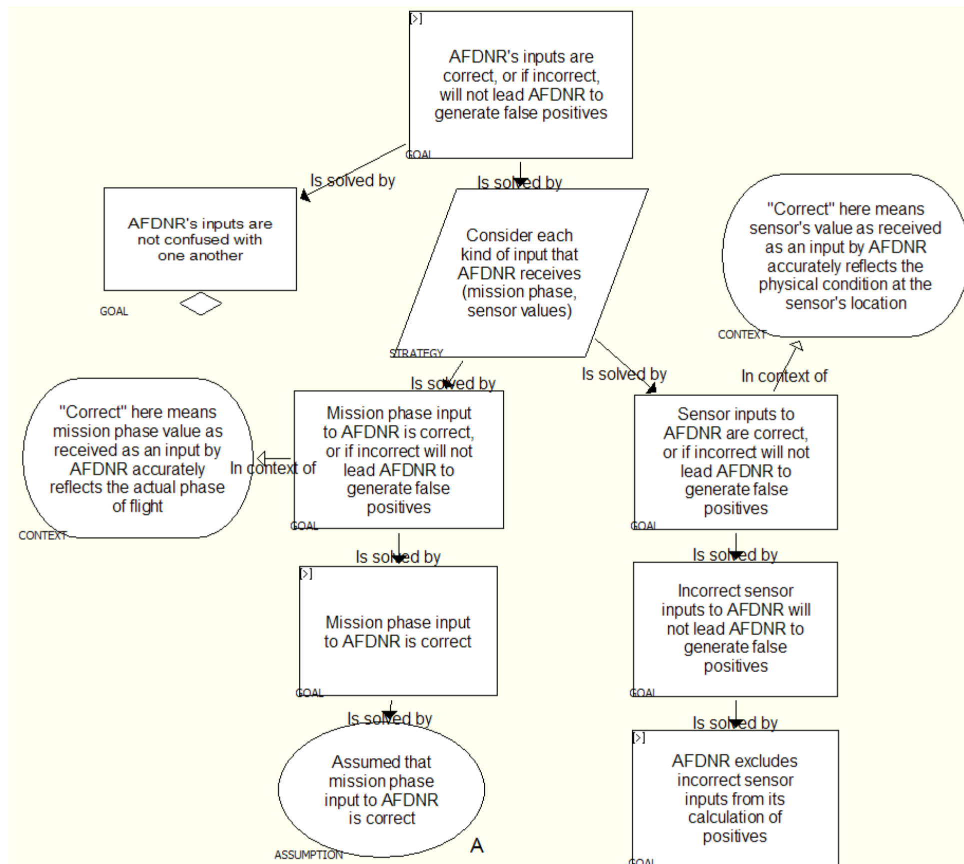

Figure 1. Decomposition of the goal “AFDNR’s inputs are correct, or if incorrect, will not lead AFDNR to generate false positives”

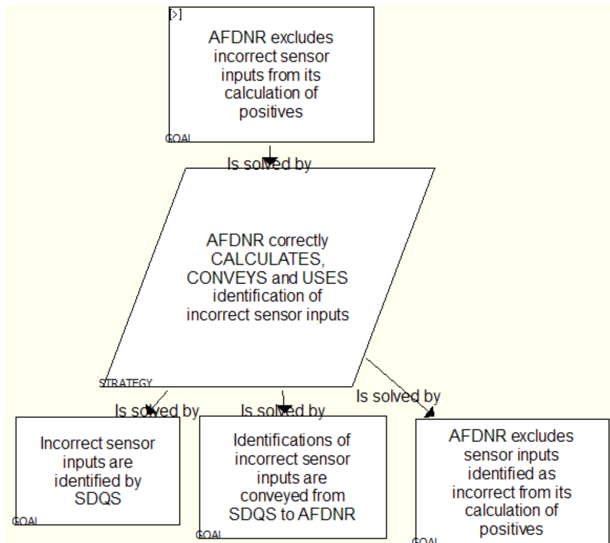

Figure 2. An instance of the Calculate, Convey, Use Pattern

example of the concept of a "safety case pattern" [17], which says of them: "As with Design Patterns, Safety Case Patterns are intended to describe partial solutions, i.e., for safety cases – attacking just one aspect of the overall structure of the safety argument contained within a safety case". In software systems there will clearly be many occasions when the calculation/conveyance/use pattern could apply.

We think this pattern is useful to apply when, as here in the AFDNR system, safety hinges on key information being computed by one software component and used by another.

F. *AFDNR's internals*

We now briefly consider the Internals subgoal: "AFDNR's calculations do not generate false positives". As to be expected, the decomposition of this subgoal depends on the specifics of AFDNR, details of which are beyond the scope of this paper to elucidate. We here give a brief overview of the range of considerations addressed at this level.

Within AFDNR the existence of abort conditions is determined based on sensor readings exceeding pre-established thresholds. Avoiding false positives during these determinations depends on:

- Threshold settings being designed correctly – "the choice of abort detection thresholds can influence the rates of false negatives and false positives" [11]
- Those settings being encoded correctly in the software.
- The AFDNR software correctly implements the logic of tests of sensor readings against thresholds.
- The AFDNR software executing those tests at the right times.

Each of these gives rise to an equivalent subgoal in our safety case.

In addition, AFDNR requires that an abort condition must be both detected and either "confirmed" or "corroborated". As stated in [11]: "…*confirmation* is defined as affirmation of the failure condition by measuring similar sensors multiple times, and *corroboration* is defined as affirmation of the fault or failure condition by measuring dissimilar sensors or assessing the health status of other vehicle components or subsystems." This scheme further mitigates the risk of false positives, and therefore is represented in our safety case.

G. *Evidence (testing, simulation, MCDC etc)*

AFDNR's developers had implemented and tested executable prototypes of some of the AFDNR fault detection algorithms specified by flow diagrams and other means in the system design document. Each algorithm implemented the detector for one abort condition. The developers used Mathworks® SystemTest™ to generate test cases, covering the full range of inputs (sequences of sensor inputs and qualification flags from SDQS). These inputs included nominal sensor data that should not cause AFDNR to respond with a fault detection, as well as data that should cause one of the faults of interest. The results were compared against the Matlab™ model in the system design document, which served as an oracle independent of the flow diagram specifications. The automated test case generator was used to generate tests for modified condition-decision coverage (MCDC) coverage of each abort condition handler. Given that each detection algorithm that was tested had a low cyclomatic complexity, no recursion and only bounded iteration, the test team lead believed that full path coverage would have been feasible given a reasonable level of funding and appropriate test tools.

The safety case captures the kinds of evidence and justification that are required to show the adequacy of the validation efforts. A high level goal is "The testing methodology is appropriate to make false positives sufficiently unlikely." This requires a subgoal to justify the use of each tool. Another subgoal is needed to justify the use of MCDC testing. Another high level goal is to justify the capability (credentials, experience) of the test team.

Many other issues would need to be addressed in the safety case for the *flight* software, for example, the safety of the abort executive, which orchestrates the execution of each of the abort condition algorithms, and the safety of the execution platform. We anticipate that the flight software implementations of the algorithms are likely to be more complex and difficult to test to an adequate level of assurance. Automated verification tools such as static analyzers and worst-case execution time analyzers might be required. Autocoding might be used. The use of such tools requires justification in the safety case. One issue with the use of such tools is that a safety case must provide a "*comprehensible*… case that a system is safe…" Automated tools such as static analyzers and code generators are generally not able to produce *comprehensible* evidence. For example, a static analyzer may be able to generate comprehensible evidence, in the form of "execution" traces that demonstrate that a claimed defect is, in fact, a defect; however, even if such tools can prove the *lack* of defects of a certain class, they typically are not able to provide *comprehensible* evidence that these defects are absent. An

example of work that addresses this problem by aiming to generate *explainable* verification is seen in [23].

VI. Results/Impacts/Future Directions

A. Conclusions

This was our first time at developing a safety case, and we make the following observations about our effort:

- We had available to us voluminous documentation of the AFDNR design and its system context, totaling several hundreds of pages. Although we had some prior familiarity of AFDNR from modest involvement in its conception, we nevertheless found it challenging to try to gain the understanding of AFDNR that we would need to develop its safety case. We believe it would have been easier for us had we been involved in developing the safety case from the very start.
- In developing the safety case itself we started from a "blank sheet of paper" (actually, the blank drawing canvas of the Adelard's ASCE™ safety case tool). Despite the well-written guidance on how to go about development of safety cases (including material from Adelard [24], and some training materials that Tim Kelly of the University of York, UK kindly shared with us), we found it daunting to get started. We believe that the availability of examples of safety cases for software systems, especially an example crafted for training in the use of safety cases, would have helped us.
- We have iterated several times on the safety case itself, significantly reorganizing it as we improved our understanding of how safety cases should be structured (for example, in an early version we failed to organize the overall safety case as an argument for why the system would be safe, instead lapsing into a more traditional fault tree-like structure). We may yet find the need to make additional such reorganizations as our understanding improves further.
- In retrospect we see our safety case development applies a series of generic software considerations (e.g., the generic concerns of "interference", "internals", "inputs" and "outputs" described in section V.C; the "Calculate/Convey/Use" pattern in section V.E) instantiated on the specifics of the AFDNR software system. We believe that this approach has potential for use on a wide range of software systems.

B. Future work

We would like to continue our safety case to explore what would be the next step in AFDNR's development, where architectural choices are made on how to realize the its design. This would give us experience with what John Knight calls "Assurance Based Design" [25], whereby consideration of the assurance implications of design alternatives can be useful to guide choice among those alternatives.

We plan to include in our safety case the results of some of the testing that has already been performed on the prototype of AFDNR's design. In the safety case, the results of such testing would form evidence supporting some of the goals of the safety case.

We also plan to provide guidance on developing and using safety cases within the existing NASA software lifecycle.

Acknowledgment

This research was carried out at the Jet Propulsion Laboratory, California Institute of Technology under a contract with the National Aeronautics and Space Administration, and at NASA Ames Research Center.

The work was sponsored by the NASA Office of Safety and Mission Assurance under the Software Assurance Research Program led by the NASA Software IV&V Facility. This activity is managed locally at JPL through the Assurance and Technology Program Office and at NASA Ames Research Center by the Robust Software Engineering group.

We also thank Adelard LLP for allowing us extended use of its ASCE™ software tool to edit and maintain our safety case.

Tom Pressburger of NASA Ames Research Center provided NASA Ames management of the project and has given feedback on our safety case research and specifically on earlier drafts of this paper. The authors held numerous conversations with the NASA Ames FDNR team including Dwight Sanderfer, Masoud Mansouri-Samani and Anupa Bajwa. The authors held meetings with John Rushby of SRI International, who reviewed safety case development and provided numerous references on safety case research. Josef Pohl and Ibrahim Habli provided helpful feedback on the paper and portions of the safety case shown herein. However, any errors in this paper are the sole responsibility of the authors.

References

[1] P. Wilkinson, "Safety Cases: Success or Failure?," Seminar Paper 2 at the National Research Centre for OHS regulation. http://ohs.anu.edu.au/publications/pdf/seminar_paper_2.pdf

[2] "Safety Management Requirements for Defence Systems": "Part 1 Requirements," UK MOD Defence Standard 00-56. http://www.dstan.mod.uk/standards/defstans/00/056/01000400.pdf

[3] C.B. Weinstock, J.B. Goodenough, and J.J. Hudak, "Dependability Cases," Technical Note CMU/SEI-2004-TN-016, 2004.

[4] "Constellation Program Computing System Requirements," CxP 70065, May 25, 2010.

[5] D. Jackson, M. Thomas, and L.I. Millett (eds). "Software for Dependable Systems: Sufficient Evidence?", National Research Council, 2007, National Academies Press.

[6] M. Barry, private communication to the authors, October 30, 2009.

[7] M. Lowry, private communications to the authors, 2008 – 2010.

[8] University of Virginia Dependability Research Group, "Safety Cases:Repository", http://dependability.cs.virginia.edu/info/Safety_Cases:Repository

[9] N. Basir, E. Denney, and B. Fischer, "Deriving Safety Cases for Hierarchical Structure in Model-based Development" in Proceedings

of the 29th International Conference on Computer Safety, Reliability and Security (SAFECOMP 2010), September 2010.

[10] A. Ellis and E. Nguyen, "Assurance Case Patterns for Flight Software", Workshop on Spacecraft Flight Software, 2010. http://flightsoftware.jhuapl.edu/files/2010/FSW10_Ellis.pdf

[11] G. Pisanich, A. Bajwa, D. Sanderfer, and M.D. Watson,"An Abort Failure Detection, Notification, & Response System: Overview of an ISHM Development Process," IEEE Aerospace Conference, March 2008. IEEAC paper #1404.

[12] M. Feather and L. Markosian, "Safety Case for NASA Ares Abort Fault Detection, Notification & Response", NASA Software Assurance Symposium, Fairmont WV, September 2009.

[13] "Software Safety Standard," NASA-STD-8719.13B w/Change 1, July 2004, available from http://www.hq.nasa.gov/office/codeq/doctree/871913B.pdf

[14] D. Jackson, "A Direct Path to Dependable Software," Communications of the ACM, vol. 52, no. 4, April, 2009, pp. 78-88, doi: 10.1145/1498765.1498787

[15] "The EUR RVSM Post-Implementation Safety Case," EUROCONTROL RVSM A1190, July, 2004, p. 17.

[16] M. Stamatelatos, "Probabilistic Risk Assessment Procedures Guide for NASA Managers and Practitioners, Version 1.1," NASA, August 2002, pp. 186 – 205. http://www.hq.nasa.gov/office/codeq/doctree/praguide.pdf

[17] T. Kelly, "Arguing Safety – A Systematic Approach to Managing Safety Cases," September, 1998, available from http://www-users.cs.york.ac.uk/~tpk/tpkthesis.pdf

[18] R. Alexander and T. Kelly, "Escaping the Non-Quantitative Trap", in Proceedings of the 27th International System Safety Conference (ISSC '09), August 2009.

[19] R. Hawkins, T. Kelly, J. Knight and P. Graydon, "A New Approach to Creating Clear Safety Arguments," Proceedings of the 19th Safety Critical Systems Symposium, February 2011

[20] IEEE Standard for Software Verification and Validation, IEEE Std 1012-2004. IEEE Computer Society, June 1005, p. 3.

[21] E. Wong, C. Fulton, W. Maul, and K. Melcher, "Sensor Data Qualification System (SDQS) Implementation Study," International Conference on Prognostics and Health Management 2008 (PHM08) Denver, Colorado, October 6–9, 2008.

[22] T. Kelly and R. Weaver, "The Goal Structuring Notation – A Safety Argument Notation," Proceedings of the Dependable Systems and Networks 2004 Workshop on Assurance Cases, July 2004.

[23] E. Denney and B. Fischer, "Generating code review documentation for auto-generated mission-critical software" Proc. Third IEEE International Conference on Space Mission Challenges for Information Technology (SMC-IT), Pasadena, California, Jul. 19-23, 2009.

[24] "ASCAD Adelard Safety Case Development Manual", available for download, after registration, at Adelard's website http://www.edelard.com/resources/ascad

[25] P.J. Graydon, J.C. Knight & E.A. Strunk, "Assurance Based Development of Critical Systems" in Proceedings of the 2007 International Symposium on Dependable Systems and Networks (DSN), June 2007