Tool-Supported Verification of Contingency Software Design in Evolving, Autonomous Systems

Robyn Lutz
Jet Propulsion Lab/Caltech and Iowa State University
rlutz@cs.iastate.edu

Ann Patterson-Hine
NASA Ames Research Center
apatterson-hine@mail.arc.nasa.gov

Anupa Bajwa
UARC, NASA Ames Research Center
abajwa@mail.arc.nasa.gov

Abstract

Advances in software autonomy can support system robustness to a broader range of operational anomalies, called contingencies, than ever before. Contingency management includes, but goes beyond, traditional fault protection. Increased autonomy to achieve contingency management brings with it the challenge of how to verify that the software can detect and diagnose contingencies when they occur. The approach used in this work to investigate the verification was two-fold: (1) to integrate in a single model the representation of the contingencies and of the data signals and software monitors required to identify those contingencies, and (2) to use tool-supported verification of the diagnostics design to identify gaps in coverage of the contingencies. Results presented here indicate that tool-supported verification of the adequacy and correct behavior of such diagnostic software for contingency management can improve on-going contingency analysis, thereby reducing the risk that change has introduced gaps in the contingency software.

1. Introduction

Advances in software autonomy can support system robustness to a broader range of operational anomalies than ever before. Autonomy is the capability of a software system to make control decisions on its own. Autonomous systems can thus often handle not only faults but also other unexpected environmental or operational scenarios that might add operational risk.

These broader classes of anomalies that must be anticipated and handled are called contingencies [3]. Contingencies include, but go beyond, traditional fault protection. They are obstacles to the fulfillment of a system's high-level requirements that can arise during real-time operations. Two examples of critical contingencies in the flight applications described here were the lens cap having been left on a camera (causing an all-black image and consequent loss of stereo imaging) and interference with communication from other wireless devices in the area (impeding a safe landing).

Increased autonomy brings with it the challenge of how to verify that the software can detect and diagnose contingencies when they occur. The problem addressed in this paper is how to verify the adequacy of the software design to detect, diagnose and respond to contingencies in an evolving, autonomous system.

Such verification is particularly difficult and time-consuming when the system itself changes over time [7]. Especially in cases where the software is evolving (as, for example, in successive updates to an operational system) or where the software is being considered for reuse in a similar system, tool support seems to be essential in practice.

The approach used to investigate this question was two-fold: (1) to integrate in a single model the representation of the contingencies and of the data signals and software monitors required to identify those contingencies and (2) to use tool-supported verification of the diagnostics design to identify gaps in coverage of the contingencies. This paper describes our tool-based verification approach and its evaluation on key contingencies in two NASA applications. The first application concerned contingencies that can interfere with the vision subsystem on an operational unpiloted aerial vehicle (UAV). The second application concerned contingencies that can interfere with the critical pointing software on an operational spacecraft.

Supporting low-cost maintenance of the model and analysis when the software detecting the contingencies evolved or was reused was the primary motivation for automating the analysis. In the first application, the UAV undergoes rapid software evolution during

operations which makes it difficult to manually keep up with the changes. In the second application the software will be reused on another upcoming spacecraft making it essential to maintain the contingency analysis as the requirements for the new system evolve. Results reported here indicate that tool-supported verification of the completeness of such diagnostic software for contingencies can improve on-going contingency analysis, thereby reducing the risk that change has introduced gaps in the contingency software.

The paper is organized as follows. Section 2 introduces the two applications on which the approach was evaluated. Section 3 describes the approach to tool-supported modeling and verification of contingencies used here. Section 4 presents and discusses the verification results both initially and in terms of maintaining the monitorability analysis as the software evolves or is reused. Section 5 describes related work. Section 6 discusses the implications of these results for other high-reliability, autonomous systems. Section 7 offers some concluding remarks.

2. Background

This section briefly describes the two applications used to illustrate and evaluate our tool-based verification approach.

2.1 Rotorcraft

The first application was to the vision subsystem of a small unpiloted aerial vehicle (UAV), called the Autonomous Rotorcraft Project [18]. The challenge in this application was to verify the capability of the software to diagnose a set of contingencies. The verification had to be maintainable on an on-going basis because the software, and hence the software model, was evolving toward increased autonomy. As the software evolved toward additional autonomy, the set of contingencies that had to be modeled also changed.

The vision subsystem is a critical system for the UAV. For example, if there is no line of sight from the pilot on the ground to the UAV, then the camera must provide critical position information for autonomous landing and obstacle avoidance. The cameras may also provide backup when other subsystems fail. For example, in the case of a failed or degraded GPS, the cameras can provide critical position information. Similarly, in the case of a failed or degraded laser-ranging system, the cameras provide ranging (including landing-site) data. In some surveillance missions camera failure can jeopardize the success of the mission. Software contingency handling can help prevent some critical failures that threaten the UAV or its mission.

The UAV has three cameras—a stereo pair of cameras to provide input to a passive range estimation algorithm and a color camera to provide situational awareness—as well as a scanning laser range finder.

2.2 Spacecraft

The second application was to the critical pointing software developed for the MER (Mars Exploration Rovers) mission launched in 2003 and planned for reuse on the Mars Science Lab spacecraft (MSL) to be launched in 2009. This software commands the appropriate orientation (e.g., pointed at the sun or turned to the correct position prior to an engine burn) in certain critical scenarios such as prior to an autonomously controlled trajectory-correction maneuver during cruise or when the spacecraft finds itself no longer commandable from the ground station [12]. MSL will face many of the same contingency scenarios as MER [10]. While important design decisions for MSL are still being made, it appears likely that the critical-pointing software will be reused, with appropriate modifications, on MSL.

In modeling and analyzing the critical-pointing contingencies with the TEAMS (Testability Engineering and Maintenance System) toolset, we represented some potential obstacles to achievement of Critical Pointing and how those obstacles were detectable in the software. The challenge in this application was to ensure that the output from the software checks adequately distinguished among the possible contingency scenarios. The verification was intended to show that the available data signals and software could disambiguate (uniquely identify) the contingencies of concern.

For autonomous systems that are safety-critical, such as the UAV and spacecraft, some contingencies pose hazards to the system's mission or to the vehicle itself. The planned, eventual flights of UAVs over populated areas and in air traffic make contingencies that interfere with onboard imaging a special concern. Similarly, the fact that spacecraft operate too far away for operators to command real-time maneuvering makes contingencies that interfere with onboard trajectory-correction maneuvers a concern. The tool-based approach described below supports the design verification of robust diagnostic software to detect and address these contingencies.

3. Approach

This section describes the process steps for tool-supported verification of contingencies and illustrates each step with examples from the two applications.

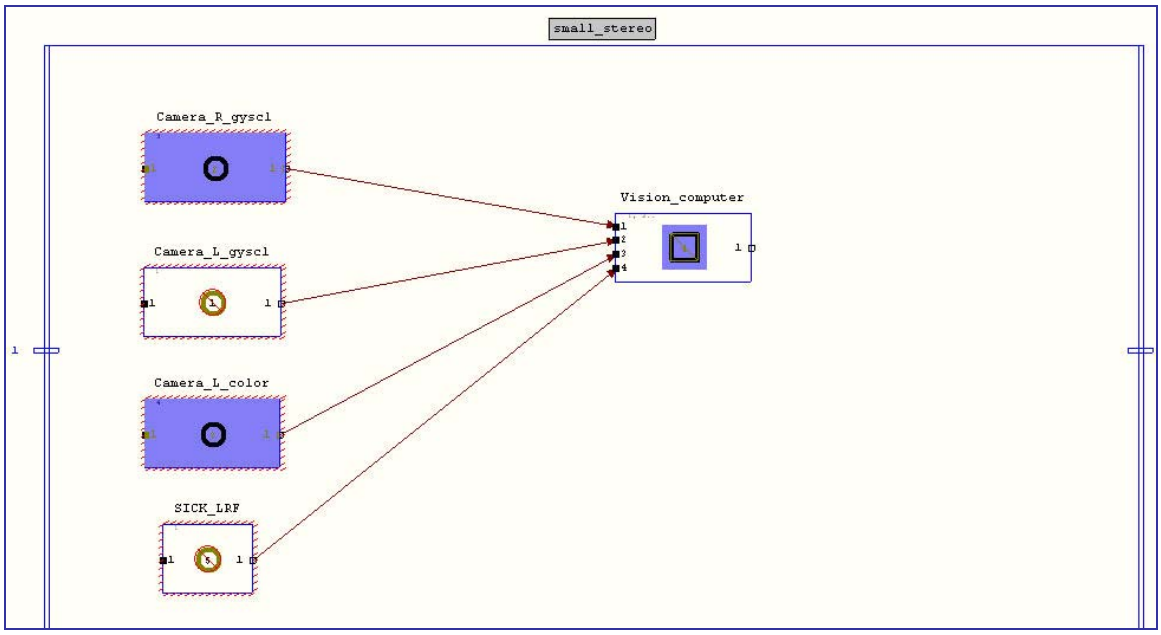

Figure 1 Camera subsystem

3.1 Identifying contingencies

The first step of the process involved identifying and modeling the contingencies of concern. We used bi-directional software analysis (including software fault tree analysis and software failure modes, effects and criticality analysis) [9] as a structured method to identify contingencies that could prevent the success of the software's goals. We then used goal-oriented obstacle analysis [17] to investigate and evaluate alternative resolutions to contingencies that could obstruct achievement of the system's goals. This structured process, as well as lessons learned regarding its effective use in a rapidly evolving operational system such as the rotorcraft, is described in detail in [8].

As noted above, some examples of identified contingencies for the UAV were the lens cap having been left on a camera (causing an all-black image and consequent loss of stereo imaging) or interference with communication from other wireless devices in the area. Examples of identified contingencies for the spacecraft pointing software were losing sun orientation or not receiving any commands from the ground station for a long period of time (indicating a problem onboard).

3.2 TEAMS modeling

The second step of the process created a model of the target subsystem and the associated contingencies, together with the data flows and software monitors available to detect the occurrence of the contingency (the event or state of concern). The level of effort for the modeling of the camera subsystem in TEAMS was less time-consuming for us than understanding the domain well enough to identify the contingencies and the appropriate responses to them. We selected the TEAMS toolset from Qualtech Systems, Inc. for our modeling and contingency analysis [14]. TEAMS is a commercial tool developed to support the modeling, diagnosability analysis and automated troubleshooting of avionics and other systems. A further advantage of selecting TEAMS was that, having won a NASA Space Act Award, it was more likely to be accepted into on-going, practical use on the projects.

Tool-based modeling using TEAMS integrated in one model the identified contingencies and the signals and monitors needed to identify those contingencies. The effect of using a single modeling language to represent both the contingencies and the means by which they can be diagnosed was to enable automated analysis of whether the available diagnostic means cover the space of the modeled contingencies.

More specifically, the modeling in TEAMS consisted of the following tasks:

(1) Identify the software architectural components and connectors associated with the software's key functionality. The connectors show the data signals that modules can use to identify whether contingencies have occurred. Note that the links along which data signals propagate also trace the dependencies among the modules. Properties of each component are described in that component's Properties box in TEAMS. For diagnostic purposes, the most important fields are the number of inputs and number of outputs, since failure signals propagate along these connectors.

The top-level model of the vision subsystem in Fig. 1 contains five components: the Right Grayscale Camera, the Left Grayscale Camera, the Left Color Camera, the SICK Laser Range Finder (SICK is the company name), and the Vision Computer that processes the images from the cameras. Connectors (arrows) between the components run between components' ports and carry signals or data.

The model is hierarchical. The key functional requirements implemented by software inside the Vision Computer are Stereo Processing, Stereo Range Processing, and Laser Range Processing.

(2) For each module, identify the contingencies (anomalous events or states) that can obstruct the software from achieving its goal. For example, a contingency of special concern in the camera subsystem on the UAV was losing stereo vision.

Stereo vision is essential to achieving a UAV's functional requirement to land autonomously at an unknown site. Stereo vision combines the images output by the left and right wing-mounted grayscale cameras. If stereo vision were lost due to the unavailability/malfunction of the Left Grayscale Camera, the possibility exists to instead use output from the Left Color Camera to achieve stereo vision. The quality of the combined image would likely be much poorer, but would still permit automated landing to continue.

(3) Identify the software checks or monitors, called tests in TEAMS, by which the contingencies of concern can be detected. Requirements on the ordering of the tests can be modeled in TEAMS by specifying the level of each test. The test level is an integer used to specify precedence restrictions on tests. This is especially useful for ensuring that the most-likely diagnoses will be examined first.

Recovery actions to handle identified contingencies can also be modeled in TEAMS. For example, the recovery action for the contingency "No Left Grayscale Image" uses the left color camera instead. Similarly, the recovery action for the contingency "No Laser Range Data" is to use stereo range data. Contingency-handling software behavior is represented by annotating the Technical Data block of the relevant tests with a description of the associated contingency response.

(4) Identify the architectural test-points for each of these software monitors. These test-points are placed in the model at the positions where checks can be run to detect failures or other contingencies. For example, one test-point in the camera subsystem uses the results output from the Stereo Processing component to check whether or not there is a good (un-failed) Stereo Image. There are thus two software checks run at this test-point, namely whether there is a Left Image and whether there is a Right Image. Both of these software checks need to be passed for a Stereo Image to be available.

The usefulness of having multiple software checks at a test-point is that our goal is both to detect that a problematic contingency exists (e.g., an all-black, left grayscale image) and to isolate the problem. The two checks at this test-point serve to isolate the problem down to the camera level.

To isolate the problem down to a specific failure mode--e.g., whether the lack of a left grayscale image is due to the camera having failed, to the lens cap having accidentally been left on, or to the camera having been given incorrect setup parameters--requires modeling the left grayscale camera. That is, each component at this level of the model is refined down to the next level, and so on, until the bottom level, composed of atomic, root causes, is reached. We modeled the camera subsystem at this lower level of detail, but retreated to a less detailed version to verify the diagnosability analysis and to provide a simpler XML to Planner translation target for the project, as described in Sect. 4.4.

4. Results of Tool-based Verification

This section describes the results provided by the automated analysis of the contingency model. In each case, we used the TEAMS toolset to produce a sequential trouble-shooting strategy in the form of a diagnostic tree. This step verified whether the necessary data signals and software diagnostic checks existed to detect each modeled contingency. The diagnostic tree described the sequence of software checks to be performed based on the results of prior software checks. It thus contained an optimized sequence of contingency-detection and isolation software checks. Hardware-in-the-loop simulation of the diagnostic tree contingencies on the grounded rotorcraft then provided dynamic verification of the results.

As an example, Fig. 2 shows an excerpt from a multi-screen diagnostic tree for the camera subsystem. The box labeled "5" links the check shown here to the earlier software checks in higher-level windows. At this point the diagnosis has already found that the Right Image has failed. If we now check the Left Image and find that it is good (left branch), then we have isolated the problem to the Right Camera. However, if the Left Image is also bad (right branch), then the Stereo Processing software is isolated as the problem source.

4.1 Identifying undetectable contingencies

The TEAMS tool models a system as a directed graph model and internally runs a reachability analysis to

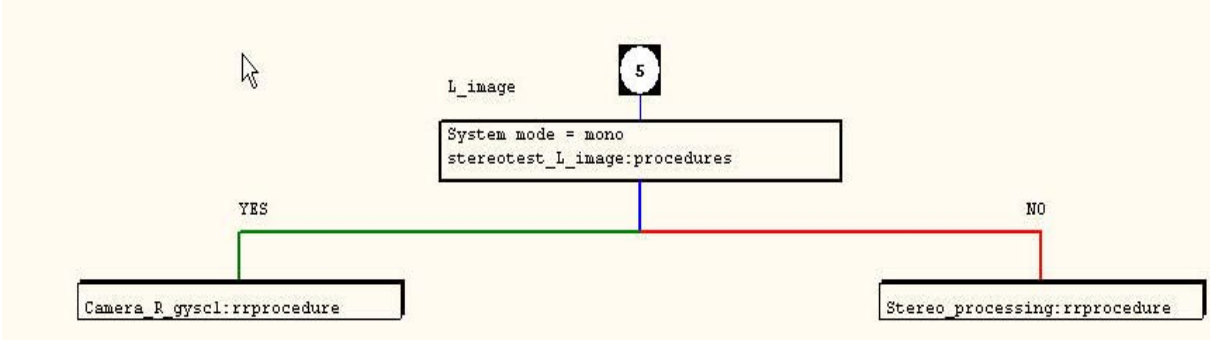

Figure 2. Excerpt from diagnostic tree

determine which faults (or other contingencies) can be detected at each test point [14]. To achieve this, TEAMS builds a dependency matrix in which each row represents a fault source (e.g., a component that can fail) and each column represents a test (e.g., a software diagnostic check). Thus, if fault i can be detected by test j, the cell (i,j) is non-empty. If row i has no entries in it, it means that the fault represented by row i cannot be detected by any available test.

For the purposes of this study, that means that we have modeled a contingency which is undetectable in the current system as modeled. Informally, something bad is happening, but there is no visibility into it on-board. Obviously, detectability is a prerequisite to autonomous detection—if the failure data is not available, the software cannot monitor it.

Most of the gaps found during the contingency analysis involved data that the software needed to perform its functions, such as messages to which it needed to subscribe or resource usage states that it had to track. This is consistent with earlier results from the TEAMS-based analysis of a previous helicopter which resulted in a recommendation to add accelerometer data to the available signals for diagnostics [13]. Identification of missing monitor data added visibility into the system state and defined the data prerequisites for future expansion of autonomy in those subsystems.

4.2 Identifying indistinguishable contingencies

If there is a set of two or more rows in the dependency matrix with identical entries, then currently no test can distinguish the fault in the first of those rows from the fault in the other row(s) in the set. That means that the software checks do not adequately isolate the source of the fault. The effect is that the contingency cannot be uniquely identified during operations in the as-modeled design. Contingencies that thus cannot be disambiguated are termed "ambiguity groups" in TEAMS. In such cases, the remedy is to add software diagnostic checks that can distinguish among the contingencies listed in the ambiguity group and re-run the automated analysis to verify the adequacy of the change.

For example, for the critical-pointing software, we investigated what can be known (i.e., on the ground via telemetry) when the software terminates and sends a message that its effort to point the spacecraft has failed. We found that, although the flown code was entirely correct, our model, which was based on the available design documentation, failed to distinguish between two anomalous situations. The automatically-generated TEAMS diagnostic tree provided a first-cut at a trouble-shooting guide to help isolate the problem that caused the critical-pointing software to terminate in a failed state.

One implication of a failure-to-isolate for autonomous contingency-handling is that the autonomous software response may, as a consequence, be needlessly coarse-grained. Since the precise source of the problem cannot be isolated, the software may do more than is needed (reconfigure more than is needed) in order to cover multiple possible failure sources. This action may address the problematic behavior, but can also have unwanted side-effects (since components not at fault may also be reconfigured) and unnecessary expenditure of resources (e.g., power, memory, cycles, etc.). It is thus important to, as much as possible, isolate the problem to its source.

4.3 Identifying redundant diagnostic checks

The TEAMS tool also detects cases in which redundant tests exist for a single contingency. This may indicate inefficiency in the contingency detection software or it may indicate that the intent differs from the actual system. (Of course, the model could also be wrong, so iterative review by project experts is a key step in the process.) We found several instances in which the consequences of a single failure (e.g., the lens cap accidentally being left on the camera) were detectable in several tests. In this case, the initial failure led to a cascading series of problems (all-black image, loss of stereo imaging) each of which was caught at a different testpoint. However, the checks were not redundant in that they operated at

different levels of refinement with some capturing more detail than others.

4.4 Simulation

To dynamically verify the adequacy of the contingency diagnostic software, we performed hardware-in-the-loop simulation. The project chose to use the rotorcraft itself for verification because there was no camera model in the software simulator which had been built to test flight control laws. Since the vision algorithms developed by the project had been verified previously with the hardware-in-the-loop simulation, it was a natural extension to also verify the contingency detection and handling software for the camera subsystem on the rotorcraft.

To accomplish the dynamic verification, the Autonomous Rotorcraft Project developed an automated translator from the XML version of the diagnostic tree produced by TEAMS to the language used by the rotorcraft's reactive planner, APEX [1]. This was an important step in supporting the on-going evolution of the autonomous rotorcraft software since it reduced the level of effort needed to dynamically verify the diagnostic trees produced by TEAMS.

The TEAMS-to-APEX translator enables updates of the diagnostic trees produced by the TEAMS model to be automatically translated from TEAMS' XML output to input readable by the rotorcraft's planner. By automating the translation between the model's diagnostic tree and the planner's execution, the consequences of design changes on contingency-handling can be cost-effectively explored on the rotorcraft prior to flight.

Model-based tools such as TEAMS have been extensively used for modeling the behavior of physical hardware, wherein it is relatively straightforward to create models of components and their dependencies or connections. In this effort we have investigated a model of flight software diagnostics which guide the vehicle when things do not work as expected. Modeling software components has its own advantages and limitations. For instance, it is easier to include more "sensors" for software components, but it is harder to completely predict dependencies and information flow between components.

4.5 Supporting evolution and reuse

A contribution of the work reported here is providing a tool-supported technique that can help verify that the onboard diagnostic software and the consequent contingency-handling software behavior are correct and appropriate when future design decisions are made.

For our first application, the initial TEAMS modeling of the Autonomous Rotrocraft Project's camera subsystem evolved as the system evolved. Some of the updates reflected changes to the hardware components (such as the addition of a new range finder and the removal of the video camera) while other updates reflected increased autonomous or imaging capabilities (e.g., image rectification and laser conversion to world frame).

As the software evolved we updated the model and re-ran the automated analysis to ensure that contingencies of concern (primarily the ever-present risk of losing stereo imaging) could be detected and tracked to root causes. Similarly, once the automated translator from the XML output of the TEAMS diagnostic tree to the rotorcraft's planning language was completed, dynamic verification was possible. Executing the camera contingency diagnostics onboard the grounded rotorcraft and watching its suggested actions to recover from some induced sub-optimal image conditions provided additional assurance that the software remained robust.

For the second application, our initial modeling of the Mars Exploration Rover's critical pointing software was updated in anticipation of the software's reuse on the Mars Smart Lander. Maintenance of the model was feasible because the contingencies that prevent successful spacecraft pointing are common to both Mars missions.

Examples of changes to the software requirements were a different sequencing of actions (e.g., at what point to open valves) and a different allocation of functions among the components (e.g., where to put a required delay). The biggest change is that on MSL the descent phase (i.e., approaching Mars) may do its own pointing of the vehicle, while the critical-pointing software modeled in this work will be used primarily for the cruise phase of the mission.

As such design changes are explored, the automated analysis of the updated TEAMS model can then be readily re-run to help verify that updates to the design maintain the capability to detect and isolate the identified contingencies.

5. Related Work

The work reported here builds on previous work to identify requirements for diagnosis and recovery at early stages of development. Following de Kleer and Kurien, we distinguish two traditions within fault diagnosis [6]. The first, fault-detection and isolation, has been primarily concerned with analyzing the diagnosability of the system, whether failures can be discriminated, or disambiguated, and what instrumentation is needed to accomplish that diagnosis [15]. More specifically, Azam et al. have previously used TEAMS, the tool we use here, to relate a hierarchical, graphical model of the data-driven system architecture to the failure modes and how they are detected [2]. Our work extends this tradition by considering not only failures but also contingencies, i.e., other anomalous operational scenarios that can interfere with the accomplishment of the system's high-level requirements.

The second fault-diagnosis tradition described in [6] is model-based diagnosis. Model-based diagnosis seeks to automate the comparison of observed behavior with a predictive model of the behavior to reason about diagnosis [16]. The goal is usually run-time verification and repair. Other promising approaches to model-based diagnosis combine off-line and on-line analyses, see, e.g., [11]. Delgado, Gates and Roach present a taxonomy and catalog of run-time, software fault-monitoring tools as of 2004 in [4].

Our approach, while also based on the automated analysis of a model of the behavior of the system, instead runs at design time to verify the robustness of the software design. It checks that the design provides adequate *monitorability* and *controllability* to provide robustness against the identified contingencies of concern [17]. Monitorability ensures that contingencies can be detected and diagnosed (i.e., uniquely identified) with existing data signals and software monitors. Controllability ensures that planned responses to contingencies are appropriate and feasible with the data signals available in the design.

The primary limitation of our approach is that, because it does not reason about faults, it cannot reliably detect unforeseen contingencies. A thorough hazards analysis (as was done on both the rotorcraft and the spacecraft described here) can reduce the possibility that high-risk contingencies escape detection to some extent. However, it remains the case that the quality of the contingency model determines the thoroughness of the verification.

The work described here also builds on previous results in requirements evolution. However, our effort differed from most of the earlier work in that we focused on the evolution of a safety-critical system during post-deployment operations. Requirements evolution post-deployment has been studied primarily from the viewpoint of how it can be managed. The focus has been on establishing processes to scope or evaluate proposed changes. Lutz and Mikulski, however, have shown how requirements evolved during operations to compensate for safety-critical anomalies [7].

The work described here also builds on previous work in vehicle health management. Patterson-Hine et al. investigated potential failures or malfunctions of engine and thruster components on a helicopter, together with the effects on the system and the visibility into faults obtainable on the ground [13]; see also [5]. Whalley et al. subsequently described the challenges involved in using vehicle health modeling of a UAV to assist in automated transition from remote control of the vehicle to computer control [18]. The approach to verification of contingency design described in this paper is part of an effort to find a process that supports that type of incremental autonomy.

6. Implications for Critical Autonomous Systems

The work described here has several implications for verification of anomaly-handling designs in other critical autonomous systems:

- **Contingency-handling adds robustness to autonomous systems.** Autonomous systems expand the traditional notion of fault protection by offering mechanisms to detect and handle a much broader range of anomalous circumstances that can occur during operations. We introduced the notion of contingencies in Section 1 to describe the expanded robustness requirements assigned to software in autonomous systems.
- **Joint modeling of software components and contingencies finds gaps in diagnostic capabilities.** Results from the TEAMS modeling efforts described here show how specifying in a single model both the planned software monitors and the contingencies of concern enabled tool-based analysis of missing diagnostic signals, missing diagnostic tests (i.e., monitors), and ambiguous diagnoses.
- **Simulation of the diagnostic tree helps verify the correctness of the contingency responses.** The capability to translate the output from the diagnostic analysis into an executable format allowed the diagnosis of contingencies and the triggering of the appropriate responses also to be checked on the executing system. The finding that executable specifications assist verification is not novel; what is new is the finding that execution of auto-generated planner commands from the TEAMS diagnostic tree provided a cost-effective way to dynamically verify key robustness capabilities in the design.
- **Automated tool support helps maintain a verified system as autonomous software evolves.** As in the UAV and the spacecraft, autonomous detection and response capabilities are often added incrementally or evolve after operational use has commenced. Verifying that the enhanced software design works as intended, as well as that it continues to correctly diagnose and respond to previously modeled contingencies, benefits from tool support. Enabling this tool-supported, iterative analysis reduced the risk that change introduced gaps in the contingency software.

7. Conclusion

The tool-based contingency modeling and analysis process applied here supports verification by providing additional information about which anomalous behavior can be detected and to what level of detail its causes can be isolated. The diagnostic tree automatically produced

by the TEAMS toolset provides answers to questions such as:

- Can the design detect each contingency that we modeled?
- Can the design isolate each contingency that we modeled?
- Is there a recovery action specified in the model for each contingency?

Such understanding of the causes and consequences of potential contingencies is a prerequisite to increased autonomy in critical systems such as UAVs and spacecraft. The automatic generation of diagnostic trees both saves time and helps manage the complexity of evolving interfaces between software and the system.

Acknowledgments

The research described in this paper was carried out in part at the Jet Propulsion Laboratory, California Institute of Technology, under a contract with the National Aeronautics and Space Administration. At JPL, we thank Tracy Neilson, Sanford Krasner, Charles Barnes and Kenneth Myers. At ARC we thank Matthew Whalley, the manager of the Autonomous Rotorcraft Project, as well as Chad Frost, Doron Tal, Rob Harris and Michael Freed for their assistance and encouragement. Their efforts were funded by the Intelligent Systems Program. We thank Qualtech Systems, Inc. for their advice regarding TEAMS modeling and diagnostics. The first author's research is supported in part by NSF grants 0204139, 0205588 and 0541163.

References

[1] *APEX Overview*, http://human-factors.arc.nasa.gov/apex /content/about.html.

[2] M. Azam, D. Pham, F. Tu, K. Pattipati, A. Patterson-Hine, and L.Wang, "Fault detection and isolation in the Non-Toxic Orbital Maneuvering System and the Reaction Control System", *Proc. Aerospace Conference, 5,* 2004, IEEE, pp. 2892- 2902.

[3] R. Dearden, R., et al., "Contingency Planning for Planetary Rovers", *Proc 3rd Int'l NASA Workshop Planning & Scheduling for Space*, Houston, Texas, 2002.

[4] N. Delgado, A. Q. Gates, S. Roach, "A taxonomy and catalog of runtime software-fault monitoring tools", *IEEE Trans on Software Engineering*, 30, 2004, pp. 859-72.

[5] R. Dixon, R. Wayne, T. Hill, K. A. Williams, W. Kahle, A. Patterson-Hine, and S. Hayden, "Demonstration of an SLI Vehicle Health Management System with In-fight and Ground-based Subsystem Interfaces", *Proc. IEEE Aerospace Conf*, 2003.

[6] J. de Kleer and J. Kurien, "Fundamentals of model-based diagnosis," *Safe Process,* 2003.

[7] R. Lutz and I. Mikulski, "Empirical Analysis of Safety-Critical Anomalies during Operations", *IEEE Trans Software Eng*, 30, 3, March, 2004.

[8] R. Lutz, S. Nelson, A. Patterson-Hine, C. Frost and D. Tal, "Identifying Contingency Requirements Using Obstacle Analysis", *Proc. 13th IEEE Req Eng Conf (RE'05)*, 2005.

[9] R. Lutz and R. Woodhouse, "Requirements Analysis Using Forward and Backward Search," *Annals of Software Eng*, 1997.

[10] Mars Science Laboratory mission homepage, http://sse.jpl.nasa.gov/missions/profile.cfm?Sort=Target&Target=Mars&MCode=MarsSciLab

[11] T. Mikaelian, B. C. Williams and M. Sachenberger, "Model-based monitoring and diagnosis of systems with software extended behavior" *AAAI05*, 2005.

[12] T. Neilson, "Auto-TCM, Critical Point & Sunpoint Functional Design Document", JPL Internal D22437, 7/23/03.

[13] A. Patterson-Hine, W. Hinderson, D. Sanderfer, S. Deb, and C. Domagala, "A Model-based Health Monitoring and Diagnostic System for the UH-60 Helicopter", *Am'n Helicopter Society 57th Annual Forum*, AHS, Washington, 2001.

[14] Qualtech Systems, Inc., www.teamqsi.com.

[15] L. Trave-Massuyes, T. Escobet, and R. Milne, "Model-based diagnosability and sensor placement application to a frame 6 gas turbine subsystem." *Proc. IJCAI'2001, 2001.*

[16] L. Trave-Massuyes, I. Russell, J. Thomas, *Model-based Fault Detection and Diagnosis: Articles & Tutorials*, Mar, 2004.

[17] A. van Lamsweerde and E. Letier, "Handling Obstacles in Goal-Oriented Requirements Engineering", IEEE *Trans Software Eng*, Vol 26, 3, October 2000.

[18] M. Whalley, M. Freed, M. Takahashi, D. Christian, A. Patterson-Hine, G. Schulein, and R. Harris, "The NASA/Army Autonomous Rotorcraft Project," *Proc. Am'n Helicopter Society 59th Annual Forum*, 2003.