Identifying Contingency Requirements Using Obstacle Analysis

Robyn Lutz, JPL/Caltech and Iowa State University, rlutz@cs.iastate.edu
Stacy Nelson, NelsonConsulting/QSS, NelsonConsult@aol.com
Ann Patterson-Hine, Ames Research Center, apatterson-hine@mail.arc.nasa.gov
Chad R. Frost, Ames Research Center, cfrost@mail.arc.nasa.gov
Doron Tal, USRA/RIACS at NASA Ames Research Center, dtal@email.arc.nasa.gov

Abstract

This paper describes the use of Obstacle Analysis to identify anomaly-handling requirements for a safety-critical, autonomous system. The software requirements for the system evolved during operations due to an on-going effort to increase the autonomous system's robustness. The resulting increase in autonomy also increased system complexity. This investigation used Obstacle Analysis to identify and to reason incrementally about new requirements for handling failures and other anomalous events. Results reported in the paper show that Obstacle Analysis complemented standard safety-analysis techniques in identifying undesirable behaviors and ways to resolve them. The step-by-step use of Obstacle Analysis identified potential side effects and missing monitoring and control requirements. Adding an Availability Indicator and feature-interaction patterns proved useful for the analysis of obstacle resolutions. The paper discusses the consequences of these results in terms of the adoption of Obstacle Analysis to analyze anomaly-handling requirements in evolving systems.
Keywords: contingency requirements, obstacle analysis, safety-critical software, requirements evolution, autonomy, anomaly-handling.

1. Introduction

This paper describes the use of Obstacle Analysis to identify anomaly-handling requirements for a safety-critical, autonomous system. The software requirements for the system continued to evolve during operations due to an on-going effort to increase system robustness. The objective of the requirements evolution was for the software to handle more faults and anomalous situations autonomously. The resulting increase in autonomy also increased complexity. One risk was that the added software complexity and interactions would result in missing, inconsistent, or poorly understood requirements for handling anomalies. Obstacle Analysis was undertaken to mitigate that risk.

The software for the system was developed in incremental builds. Requirements evolved rapidly with autonomous features being added regularly. Autonomy is the capability of a software system to make control decisions on its own. An advantage of autonomy in this system was that it allowed the system to react faster than with human-in-the-loop controllers to failures, anomalous situations, and changes in environment. With autonomy, safe operation of the system could continue even when the human controller was unavailable.

Autonomy can also contribute to system robustness by detecting and responding to a broader class of anomalies than just failures. The requirements for anomaly handling in many safety-critical, autonomous systems encompass not only traditional fault protection but also unexpected environmental or operational scenarios that could contribute to hazards.

These broader classes of anomalies that must be anticipated and handled are called "contingencies" [7]. Contingencies include, but go beyond, traditional fault protection. They are obstacles to the fulfillment of the system's high-level requirements that can arise during real-time operations. Contingency handling involves requirements for detecting, identifying, and responding to contingencies.

The application of obstacle analysis in this paper was to an unpiloted aerial vehicle (UAV) called a rotorcraft (described in Section 2). The system was safety-critical in that it was required to take off and land autonomously. Initially flight and landing occurred in constrained airspace, but eventually will occur in populated areas. Unpiloted aerial vehicles are also being

developed for safety-critical applications such as imaging high-risk environments (e.g., volcanoes and forest fires); detecting accidents in highway traffic, and assisting search and rescue operations for lost hikers or downed pilots [13].

The fact that the vehicle was already operational (flying) during most of the process made an early understanding of contingencies and of their potential effect on safe flight important to the project. To achieve the required levels of system robustness, a structured approach was needed for investigation of contingency requirements.

Accordingly, we drew on the KAOS framework for goal-oriented obstacle analysis developed by van Lamsweerde and Letier. They defined a goal as a set of desired behaviors and an obstacle as a set of undesirable behaviors that impeded a goal [23]. Their goal-oriented techniques for obstacle analysis have been previously demonstrated on critical systems such as ambulance dispatch and the robotic production cell with good results.

The underlying rationale for the choice of goal-oriented obstacle analysis for the requirements analysis of the rotorcraft was that contingencies are either obstacles to achieving goals or are indications that the goals are unrealizable with the available agents. The requirements evolution in this system was toward more autonomy. Incremental autonomy, which involves replacing human agents with software agents, produces new sub-goals to achieve autonomy, new obstacles associated with those sub-goals, and additional sub-goals to resolve those obstacles, as well as new alternatives to resolving previous obstacles. All of these must be investigated if the system is to remain robust.

In the system described here, incremental autonomy meant that something previously not done automatically now was done by the software. In some cases the function had previously been done manually by operator control, while in other cases something that previously could not be done at all was now done automatically.

For example, collision avoidance (e.g., of the rotorcraft with simulated buildings) initially was done via remote control by a pilot steering the airborne rotorcraft remotely from the ground. Later, collision avoidance was done by calculating a path around the buildings before each flight began and then autonomously executing the planned path during flight. Still later, the path-calculation requirements incorporated new objects that appeared during flight (such as another, hovering rotorcraft). Future software will autonomously plan a path during flight based on real-time processing of images taken by the rotorcraft's cameras and on software that autonomously detects changes in the imaged scene. As this example demonstrates, any approach to contingency analysis needed to be readily extensible and maintainable. We describe in Section 4 the benefits and limitations of obstacle analysis in this regard.

One advantage of autonomy is that it adds flexibility to a mission, e.g., allowing it to take advantage of unanticipated science opportunities that would be missed if observations had to wait for human interaction. The work described here is not concerned with this sort of adaptive re-planning for gain, but solely with negative contingencies that can place the system in a hazardous state. Some of the results appear to be applicable to intelligent systems that re-plan for optimization, but that investigation is beyond the scope of the work described here.

The paper describes our approach to using obstacle analysis as a structured way to reason incrementally about the new requirements and their alternatives. Results reported in the paper show that obstacle analysis complemented standard safety-analysis techniques in identifying undesirable behaviors and how to resolve them. The lightweight, step-by-step use of Obstacle Analysis identified potential side effects and missing monitoring and control requirements. Adding an Availability Indicator and feature-interaction patterns proved useful for the analysis of obstacle resolutions. The results suggest that Obstacle Analysis is an effective and readily adoptable technique for analyzing anomaly-handling requirements in evolving systems.

The rest of the paper is organized as follows. Section 2 briefly describes the application domain. Section 3 discusses our approach. Section 4 presents and evaluates the results of our use of obstacle analysis to identify contingency requirements. It then discusses the implications of these findings in terms of requirements evolution and incremental autonomy. Section 5 puts the results in the context of recent, related work. Section 6 provides concluding remarks.

2. Rotorcraft Application

This paper describes experience using obstacle analysis to identify contingency requirements on an unpiloted aerial vehicle (UAV) called a rotorcraft. The Autonomous Rotorcraft Project (ARP) is a NASA UAV project researching autonomous, low-altitude, flight systems.

Two small helicopters, originally developed for remote-piloted crop-dusting in farming

regions, serve as the platforms. The autonomous system has an onboard attitude sensor (accelerometers and gyros), a GPS sensor, and a communication modem. The rotorcraft has four cameras—a stereo pair of cameras to provide input to a passive range estimation algorithm, a camcorder to provide video recording, and a color camera to provide situational awareness. Ground support is housed in an instrumentation trailer with a GPS ground station, radio modems for communication with the aircraft, and video displays of the camera images. Additional description appears in [24].

We performed contingency analysis on two subsystems with safety-critical functionality (communications and perception) recommended by the project.. Our involvement began about the time that the rotorcraft began regular autonomous flights within a constrained airspace with autonomous collision-avoidance path-planning around stationary objects such as simulated buildings. The project involved a tight-knit team of experts working together with strong technical leadership. Most of our knowledge of the system came from participation in weekly team meetings and from discussions with the experts on the team rather than from limited project documentation.

The project considered contingency analysis of the communications subsystem to be important because many UAVs currently rely upon flight termination (including hard landings commanded over the radio channel) to stop the vehicle in case of failure of the wireless communication channel. Improved software contingency handling for communications failure would make it possible for the UAV to fly to a safe rally point (a pre-designated location) and land normally. The project also considered contingency analysis of the camera subsystem (the cameras and range finders) to be important. This was because the cameras have mission-critical responsibilities during some operations and because the cameras provide backup position or ranging information when other components fail.

3. Obstacle Analysis Approach

Our approach in the work described here was to use a simplified version of the KAOS framework developed by van Lamsweerde and Letier for goal and obstacle analysis to guide the requirements analysis. The reader is referred to [15,16,23] for a detailed account of the goal-oriented approach. Our application of obstacle analysis to the problem was informal and manual. The use of obstacle analysis met the criteria for lightweight applications of formal methods to requirements modeling described by Easterbrook et al. [10] in that the obstacle analysis was applied in response to an existing development problem, applied selectively to only some critical portions of the requirements, offered only a partial solution without guarantees of completeness or correctness, and fed back into the development process to improve the product.

In the rest of this section we give a brief, step-by-step account of our adaptation of the obstacle analysis framework to the project. In the following section we will describe the results of applying the obstacle analysis and evaluate our use of the goal-obstacle framework.

Step 1. Identify the Goals. A goal defines a set of desired behaviors. Goal-oriented requirements engineering organizes goals in an AND/OR structure (a directed acyclic graph). The AND nodes refine the goals into sub-goals both of which must be satisfied, and the OR nodes provide alternative ways to meet the goals. Refinement continues until a sub-goal (a terminal goal) can be achieved by an agent. Software requirements are terminal goals assigned to software agents.

Step 2. Identify the Agents. Agents (e.g., human operators, hardware devices, or software components) are active objects to whom the implementations of the sub-goals are assigned. (Note that this definition of an agent differs from the notion of autonomous software agents. In the goal-oriented approach an agent may or may not be software, and may or may not be autonomous.)

Step 3. Identify the Obstacles. An obstacle describes a set of undesirable behaviors. For example, an obstacle to the goal "Store Camera Images in Memory" is "Images Exceed Available Memory." This type of obstacle is called a "non-satisfaction" obstacle since it obstructs the satisfaction of a goal [23]. Obstacles cover a broad space of possible barriers to achieving required functionality. Contingencies are obstacles that can occur during real-time operations, including failures and other anomalous events or scenarios.

Like goals, obstacles are organized and refined in AND/OR structures, and are associated with the goals they impede. The example in [23] uses a table to specify the goals, assigned agents, and obstacles for the system being analyzed (the London Ambulance System). Obstacles usually are associated with terminal goals, i.e., goals assigned to individual agents. Obstacle refinement patterns, described both formally and as heuristics of the form "if the specification has such or such characteristics then consider such or

such type of obstacle to it," are described in [23]. We report experience with these refinement patterns below.

Step 4. Identify alternative resolutions to the obstacle. Once obstacles have been identified, they need to be resolved. There are some standard strategies to resolve the obstacles [23]. For example, to eliminate the obstacle, we can consider getting rid of the goal that it obstructs, assigning a different agent so that the obstacle does not occur, adding a new goal to require that the obstacle be avoided, changing (or "de-idealizing") the goal, or changing the domain such that the obstacle can no longer occur. Another strategy is to just tolerate the obstacle. This might be done by adding a new goal (e.g., a new requirement for contingency handling) to mitigate the consequences of the obstacle, or just by deciding to accept the occasional occurrence of the obstacle.

Step 5. Select a resolution among the alternatives. Obstacle resolution involves evaluating and selecting from among the available alternatives. As we will see, this often involved the generation of new sub-goals to eliminate, reduce, or tolerate the identified contingency. These resolutions yielded new software requirements when assigned to software agents.

4. Results: Contingency Requirements from Obstacle Analysis

In this section we describe the results from the use of obstacle analysis to identify anomalies that could occur during operations and how they could be handled by software in the existing system. The results show that a key benefit came in the improved analysis of alternative solutions to contingencies.

4.1 Goal and Agent Identification

The first steps toward obstacle analysis were to identify the goals and agents. In this system the goals and agents were usually well-understood and already refined to the necessary level. As input to the obstacle analysis, we thus produced only those portions of the goal-graph about which there was some confusion or controversy. This was done in order to facilitate review of accuracy by experts. For example, Fig. 1 shows a portion of the goal graph for three-dimensional collision avoidance that was developed for such a discussion. For most goal specifications, however, we used architectural diagrams, state diagrams, and functional

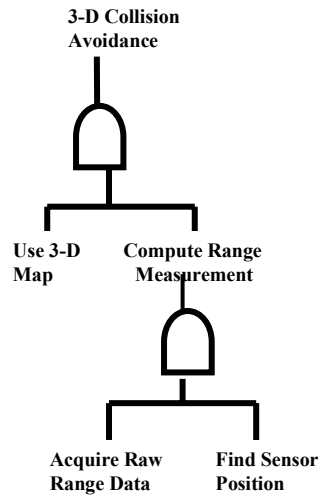

Figure 1. Goal graph excerpt

dependency graphs that we had earlier produced rather than producing a separate goal-graph.

This exploitation of existing project artifacts worked well. The decision not to produce a complete goal-graph meant that the obstacle analysis was less rigorous, with no formal proofs of completeness possible. However, obstacle analysis explicitly supports the possibility of such lightweight applications as a way to guide inquiry into potential obstacles. In our case, iterative project review of our specifications gave some assurance that we had adequately captured the goals and sub-goals.

With regard to agent identification, the rapid evolution of the system meant that both new software agents (autonomous features), as well as new hardware agents (sensors, cameras, range-finders, etc.) were regularly being integrated into the operational flight system. We discuss the effect of this on the obstacle analysis in Section 5.

.

4.2 Obstacle Identification

To help identify obstacles to the high-level requirements for communication and perception, we used Bi-Directional Safety Analysis [19]. We selected BDSA because it combines a forward analysis (from potential failure modes to their system effects) with a backward analysis (from the failures to their contributing causes). The forward analysis is similar to a Software Failure Modes, Effects, and Criticality Analysis (SFMECA).

Item	*Generic Failure Mode*	*Failure Mode Description*	*Effects*	*Criticality*	*Mitigation*
Request for Image Processing	Absent	No processing command received	Raw image not compressed; buffer limit could be exceeded; downlink stressed	Minor	Use default processing; set default to "compressed" to limit memory usage and bandwidth
	Incorrect	Processing settings may be inappropriate for conditions	Poor quality; surveillance mission or autonomous landing may require usable image	Minor to Major	Restrict processing choices based on available info about mission, resource constraints
	Timing	Requested processing applied to earlier/later image	Delay in getting usable image	Minor	Time-tag images to detect discrepancy

Table 1. SFMECA excerpt for the image-processing request

The backward analysis is similar to a Software Fault Tree Analysis (SFTA). The combination of the forward and backward analyses has proven to be a powerful way to identify and understand the causes of software-associated failures in systems. Previous uses of BDSA [19] include validating fault protection software on two spacecraft and analyzing thruster failure modes on another.

We first performed a forward analysis of the communications and camera subsystems. The SFMECA structured the investigation of possible failures. For each input it considered the effect of its absence, corruption, or untimely arrival. The SFMECA also investigated what would happen if the software hung or failed in each state, or if a transition took it to a wrong state. Details of the technique appear in [19].

The SFMECA identified several contingencies. For example, one failure mode that could contribute to several hazardous scenarios was attaching incorrect identification data (vehicle pose, attitude, GPS position, etc.) to an image. The effect was that the identification data did not reflect the image content, so could mislead the software into an erroneous control decision. Another failure mode involved a failure of image compression that could result in running out of memory to store the images needed for safe, autonomous landing.

Table 1 shows an excerpt from the SFMECA for the data input "Request for Image Processing" received by the onboard camera software. The context is that a raw image is grabbed from the video camera stream and processed (e.g., compressed) onboard before being sent to the ground.

To investigate contingencies that did not involve the failure of single components, we also performed a backward analysis. Software Fault Tree Analysis (SFTA) is a graphical decomposition of a root node into its logical, component preconditions [19]. The fault trees took as root nodes the negations of communication and perception-related goals. In some cases the root nodes were hazards (e.g., collision) that negated high-level goals (e.g., collision avoidance). SFTA considered combinations of circumstances that could together cause a problem. By identifying alternate ways to get to an undesirable state, the SFTA helped guide analysis of fault detectability and propagation.

As an example, we consider a portion of the SFTA for the root node "Failure of Stereo Imaging." (The SFTA is not shown here due to space constraints.) Both a left and a right image are required for stereo imaging. The root-node failure occurs when either the Left_Camera_Fails OR the Right_Camera_Fails. However, the left camera is redundant in that there is both a grayscale and a color camera on the left side. The left camera failure thus occurs only when both the Gray-scale_Camera_Fails AND the Color Camera_Fails. The SFTAs anchored the investigation into faults that might prevent the functional requirements, such as stereo imaging, from being satisfied.

The SFTAs also identified environmental and operational obstacles to fulfilling requirements. For example, a strong crosswind is an environmental obstacle that interferes with the camera's ability to take images, and thus with its ability to find a safe landing site. Similarly, accidentally leaving a lens cap on a camera is an operational obstacle that results in an all-black, useless image. Among other consequences, this also interferes with the rotorcraft's ability to land autonomously. Leaving the lens cap on is thus a contingency that must be considered. It has, in fact, occurred on other UAVs.

4.3. Obstacle Resolution

Identification of Alternative Resolutions. Three sources were found to be useful in identifying alternative ways to resolve obstacles. The first two of these sources were the safety-analysis artifacts, the SFMECA and the SFTA. The third source was the generic obstacle resolution patterns in [23].

The SFMECA tables contained a "Mitigation" column that described ways to eliminate or mitigate the failure mode in each row. The SFTA also identified alternatives for resolving obstacles since finding a way to negate leaf nodes removed the occurrence of some obstacles. The SFTA thus encapsulated information about the goal being studied (the negation of the root node); potential obstacles to the goal (the leaf nodes of the fault tree); and insights into necessary detection mechanisms (how to determine the occurrence of the events or conditions in the nodes).

The obstacle resolution patterns in [23] also provided guidance in thinking about possible ways to handle obstacles. For example, one pattern refers to the situation where a condition persists for an interval prior to the obstacle's occurrence. In that case the obstacle can be anticipated and perhaps prevented.

The "Agent Substitution" pattern was most frequently applicable to the autonomous handling of contingencies. This pattern captured how timeliness obstacles (such as when ground control cannot react quickly enough) could be resolved by transferring responsibility to onboard software.

Selection of a Resolution among Alternatives. Consideration of tradeoffs weighed heavily in the selection of obstacle resolutions in this application. For example, there were two different, onboard sensors, either of which could be used for range finding. However, they varied significantly in power consumption, precision, number of data points, range, and position on the vehicle. Selecting one over the other thus involved a tradeoff analysis either on the part of a ground operator or on the part of the on-board, recovery software. Other selections among alternative resolutions involved tradeoffs between color and increased memory, image compression and CPU usage, image quality and downlink bandwidth, and number of stored images vs. the size of images.

4.4 Deriving Software Requirements

In an example given earlier, where the obstacle to storing images was "Images Exceed Available Memory," alternative resolutions included "Reduce Number of Images To Be Stored" and "Reduce Size of Images To Be Stored" (compression). Each of these resolutions itself involved refinement into several sub-goals. For example, reduction of the number of images could be determined by how old the image was or by some other assigned or calculated priority measure. The calculated priority could be based on vehicle location, orientation, and camera-pointing angle or on the actual content of the image. The first option (prioritizing the images based on location, orientation, and pointing angle) yielded an approximation to the likelihood that the target of interest was captured in the image; the second option discarded images that did not include the target.

However, the first option was only feasible if the camera software could access the GPS data, which it currently could not. This is an example of what van Lamsweerde and Letier call the *unrealizability* problem, meaning that not all stated goals are realizable by agents in the system. They give some pragmatic conditions for unrealizability in [15]. For our application, the two most important conditions were Lack of Monitorability and Lack of Controllability. We also added a third condition, Lack of Availability.

Missing monitorability requirements. One of the most useful results of the obstacle analysis approach was in identifying gaps between the capability of the assigned software agents to monitor for certain states or events and the need to have them do so. Most of the new software requirements found during the contingency analysis involved the dependency of certain detection, isolation, or recovery actions on specific capabilities that the software currently lacked. In terms of goal orientation, these goals were currently unrealizable. Most often these gaps 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. Identification of missing monitor data also added visibility into the system state and defined the data prerequisites for future expansion of autonomy in those subsystems.

Missing controllability requirements. Similarly, the obstacle analysis identified several instances in which the resolution involved the software being asked to set the values of variables that it did not control. For example, the software responsible for dynamically throttling the writing of images into memory based on their priority level must be able to control (i.e., change) the value of

the priority threshold for grabbing images. Another example was that adjusting the jpeg quality of images in response to a low-light contingency (e.g., landing at twilight) can only be realized by software that controls the jpeg parameters. Both monitorability and controllability are important for establishing the software requirements needed for consistent increments in software autonomy.

Missing availability requirements We also found it useful to add a third type of unrealizability condition, which we called an Availability Indicator. The Availability Indicator addresses those situations in which the preferred alternative was not yet operational but soon would be. In particular, it documented whether the component on which a requirement depended had been installed at that point in time. In this paper we draw examples both from past software contingencies and from future, anticipated contingencies. However, in the rapidly evolving system it was essential to maintain rigor in checking consistency between a component's current availability and use of the software that depended on it.

For example, one obstacle to the goal of wireless communication that was encountered during operations was interference from other wireless devices in the vicinity. Several alternative resolutions existed to this obstacle: a return to remote (human) piloting, ignoring the risk (what is called "live with it" in [23]), creating an operational policy to not use the wireless channel in crowded environments, or switching communications to an alternate medium. This last alternative was the best, but required additional hardware and software that were not yet available.

In some cases, the resolution itself evolved. Here, in the short-term, the required response to detection of wireless interference was for the system to return to human control. In the long-term, once the new hardware agent and software agent are in place, the required response will be to switch communication to an alternate medium.

A single obstacle also sometimes spawned multiple new software requirements to achieve detection and resolution. For example, mitigating the obstacle where images exceed available memory involved both image compression and a "cancel" command to reverse acquisition of requested images that were no longer needed. In other cases one resolution could remove several obstacles. This was the case, for example, with the addition of accelerometer data on a previous helicopter [20].

4.5 Implications for Requirements Evolution

Systems that experience requirements evolution after deployment challenge the continued adequacy of previous anomaly-handling strategies. When the evolution is toward additional autonomy, the challenges are multiplied. In this section we discuss some ways in which obstacle analysis can help and some areas in which more work is needed.

Validity of existing requirements must be maintained. When requirements changed, a key use of obstacle analysis was to confirm that previously existing software requirements to detect, isolate, and respond to contingencies were still valid. This involved checking that for every previously identified obstacle to a goal that could still occur, the previously identified resolution (usually a derived software requirement) could still handle the obstacle. For example, when the capability was added for autonomous pivoting of the vehicle around a target, we had to check that the current handling of the obstacle "communication lost" remained valid.

Evolution tends toward more autonomy. In general, the evolving requirements produced greater autonomy. Enhancements to the rotorcraft provided new ways to detect, identify, or respond to existing contingencies. Often these enhancements resulted in new sub-goals (added autonomous functionality) and in new "OR" branches for the obstacle resolution model (new alternatives for how to handle contingencies). Most typical was the replacement of a subgoal leaf node assigned to a human agent by a subgoal tree with the new leaf nodes assigned to software agents.

Simple hardware agents (e.g., a camera affixed in place) tended to be replaced by more sophisticated hardware agents (e.g., articulated stereo cameras that swivel up and down), that made feasible new, alternative monitoring sub-goals for contingency detection, or alternative control sub-goals for contingency recovery. New software agents (e.g., the capability to dynamically add collision-avoidance targets to the path-planning calculation) offered improved autonomous capabilities for responding to contingencies and resulted in more recovery options.

Autonomy adds complexity. As more autonomy was required, new obstacles and dependencies were also introduced. That is, with the addition of new agent capabilities came the possibility of new contingences. For example, the addition of a new laser brought with it the failure

modes of the laser to be considered, as well as whether these failure modes were detectable on the ground and by onboard software. Some new agents also resulted in new requirements for calibration before use.

More interestingly, the evolving requirements entailed the need to identify new feature dependencies (e.g., between the fidelity of color images and passive-range algorithms). While the focus of incremental autonomy tended to be on what was *gained* in terms of more robust handling of contingencies, contingency analysis also looked at the potential *risks introduced* by the new software complexity. Contingency analysis investigated the failure modes for new agents, new opportunities for feature interactions, and new possibilities for conflicts among goals (most commonly in terms of resource contention).

Additional issues such as mixed-initiative control (where the rotorcraft receives inconsistent commands from the onboard software, the ground software, and the remote, human pilot), sensor fusion problems (where the replacement of a single sensor by a suite of perhaps heterogeneous sensors requires the software to compose the data and handle data inconsistencies), and coordination problems (where a fleet of rotorcraft must coordinate their movement and resource usage), were not obstacles in the current system but are potential, future obstacles. Our experience to date leads us to anticipate that obstacle analysis can scale to address the additional complexity in these interactions.

An unexpected finding was that sometimes the requirements for autonomy were reduced. In these cases some existing autonomous features were disabled and some control was moved from the onboard software to human ground control. This reduction of autonomy usually occurred temporarily for a demonstration or test of a new component or features. Since these instances often did not involve a simple rollback to a previous version but instead used a pruned version of the current software, verification that the reduced requirements still maintained the required obstacle handling had to take place.

Additional Obstacle-Refinement Patterns Needed. Two areas in which existing obstacle analysis techniques provided only limited guidance for the system studied were: (1) in analyzing failure isolation and (2) in identifying feature interactions. With regard to failure isolation, it was often easier to determine that a failure had occurred (detection) than it was to figure out precisely what had happened and how to prevent it from propagating (isolation). For example, if downlink communication stopped, it was easy to detect that a failure had occurred but could be quite difficult to isolate the problem. We are thus working to extend the obstacle refinement patterns to more explicitly address failure isolation issues in this domain.

With regard to detecting interactions as new features are added, we found that guidelines described by Doerr for identification of feature interaction in product lines were readily extensible to our application [9]. An example of a guideline that was useful here is that, if feature B uses feature A, then all features that also use feature A must be identified. (To illustrate, Doerr gives an example from the mobile phone domain, where both sending a short message and placing a call use the network component. To avoid conflicts, it is important to identify that both features use the same component). Doerr's guidelines supplement the obstacle-analysis patterns (for obstacles to achieving goals) and divergence-analysis patterns (for conflicts among goals) [23] in identifying feature interactions.

5. Related Work

The investigation reported here built on previous results in requirements evolution. However, our effort differed from most earlier work in that we focused on the requirements evolution of a safety-critical system during post-deployment operations.

Much of the existing work in requirements evolution addresses the pre-implementation phases of a system. Anton and Potts, for example, have used goals and obstacle analysis to refine evolving requirements in a developing system [1]. There have also been studies describing agile approaches for handling rapidly evolving systems (e.g., the evolutionary prototyping in [4]). Such approaches are recommended for market-driven domains such as e-commerce rather than for critical systems.

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, e.g., in terms of traceability [6] or change-impact studies. Similarly, maintenance methods have tended to focus on classifying and managing requirements changes rather than on analyzing changes [2].

Feather and Fickas have instead proposed monitoring operational systems for mismatches that develop between the assumptions underlying

requirements and the current environment. They describe "remedial evolutions", somewhat similar to our evolving obstacle resolutions, when such mismatches occur [11].

The domain of concern in most requirements evolution work has been the business environment rather than critical systems, as here. DeLemos, however, modeled an operational system in which requirements evolution (automating the self-destruct feature of a rocket) was structured so that architectural components remained unchanged while their interactions adapted to the changed requirements [8]. Lutz and Mikulski also showed how requirements evolved during operations to compensate for safety-critical anomalies [17,18].

Several recent papers described autonomy requirements, including fault handling, for existing or planned space missions, [3,5]. Other researchers, motivated by problems with fault identification on rovers, have presented improved algorithms for fault detection or responses. For example, Dearden et al. generated contingency plans for rovers in the presence of uncertainty regarding timing and resources [7]. The algorithms in these studies pre-supposed that requirements for fault and contingency monitoring and handling had been defined separately (the problem addressed in this paper).

A few researchers have specifically addressed safety in autonomous systems. For example, Fox and Das described the deployment of software agents for intelligent decision-making in safety-critical medical applications [12]. The issue of adjustable autonomy was raised by Schreckenghost et al. in the context of space life support systems where a human may need to override autonomous control when an anomaly occurs [21]. A European Space Agency ESTEC project investigated how to ensure safety and dependability of autonomous space software, based on lessons learned from non-space autonomous domains [22].

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 [20]. 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 [24]. Such a transition will require improved contingency analyses and helped motivate the work described in this paper.

6. Conclusion

The results reported here indicate that the use of obstacle analysis to reason about software requirements for handling contingencies has several advantages. Obstacle analysis gives a structured way to reason incrementally about new alternatives for handling contingencies that can occur during operations. The capacity for incremental reasoning is especially important in systems that are both currently operational and evolving rapidly to add autonomous features.

Obstacle analysis also supports evaluation of the continued validity of existing software contingency requirements as the system and the requirements evolves. The cyclical nature of goal-oriented requirements engineering, in which new goals are introduced to resolve an obstacle, but must then also be included in an iterative analysis for new risks, is a good match with safety-critical systems whose requirements continue to evolve after deployment.

We made several minor adjustments to the standard form of obstacle analysis to better fit our focus on contingencies.. First, we found that existing safety-analysis artifacts (SFMECA and SFTA) were useful and efficient baselines for identification and refinement of obstacles. This suggests that it may be relatively easy for projects that already produce traditional safety-analysis documentation to adopt obstacle analysis to supplement their existing analysis techniques. Second, we added to the analysis of alternative obstacle resolutions an explicit indicator of whether the required agent was currently available. This was because some alternatives for contingency detection and handling were not currently feasible in the system, but would be so in the near future. Third, we found a need for additional attention to feature dependencies and to fault-isolation problems, and are working to extend the obstacle-refinement patterns in those directions.

The objective of the requirements evolution was for the software to handle more faults and anomalous situations autonomously. By supporting the identification of software contingency requirements, obstacle analysis contributed to the building of a more robust 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 Aeronautic and Space Administration. It was

funded by NASA's Office of Safety and Mission Assurance Software Assurance Research Program. The first author's research is supported in part by National Science Foundation Grants 0204139 and 0205588. The authors thank Matt Whalley and the other members of the Autonomous Rotorcraft Project team for sharing their expertise and enthusiasm. The first author also thanks Martin Feather for insightful comments on an early draft.

References

[1] A. Anton and C. Potts, "The Use of Goals to Surface Requirements for Evolving Systems," *20th ICSE*, Computer Society, 1998, pp. 157-166.

[2] K. Bennett and V. Rajlich, "Software Maintenance and Evolution: a Roadmap", in A. Finkelstein, ed. *The Future of Software Engineering*. ACM Press, New York, 2000, pp. 75-87.

[3] G. Brat et al., "Experimental Evaluation of Verification and Validation Tools on Martian Rover Software," *Formal Methods Sys Design Journal*, to appear.

[4] R. Carter, A. Anton, A. Dagnino, and L. Williams, "Evolving Beyond Requirements Creep: A Risk-Based Evolutionary Prototyping Model," *ISRE*, 2001, pp. 94-101.

[5] S.Chien, et al., "Onboard Autonomy on the Three Corner Sat Mission," *Int'l Symp AI, Robotics, and Automation for Space*, Montreal, Canada, IEEE, 2001.

[6] J. Cleland-Huang, C. Chang and M. Christensen, "Event-Based Traceability for Managing Evolutionary Change", *IEEE Trans on Software Eng*, Computer Society, Los Alamitos, Sept, 2003, pp. 796-810.

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

[8] R. deLemos, "Safety Analysis of an Evolving Software Architecture," *5th IEEE Int'l Symp High Assurance Systems,* Computer Society, 2000, pp. 159-167.

[9] J. Doerr, "Requirements Engineering for Product Lines," Diploma thesis, U of Kaiserslautern, 2002.

[10] S. Easterbrook, R. Lutz, R. Covington, J. Kelly, Y. Ampo, and D. Hamilton, "Experiences Using Lightweight Methods for Requirements Modeling," *IEEE TSE*, Jan., 1998, pp. 4-14.

[11] M. Feather, and S. Fickas, "Requirements Monitoring in Dynamic Environments," *ICRE,* 1995, pp. 140-147.

[12] J. Fox. and S. Das, Safe and Sound, *Artificial Intelligence in Hazardous Applications*, AAAI Press, Menlo Park, CA, 2000.

[13] T. Johnson, H. Sutherland, and S. Bush, "The TRAC Mission Manager Autonomous Control Executive", *IEEE Aerospace Conference*, 2001.

[14] M. M. Lehman and J. F. Ramil, "Rules and Tools for Software Evolution Planning and Management," *Annals of Software Engineering*, 11, 1, 2001, pp. 15-44.

[15] E. Letier and A. van Lamsweerde, "Agent-Based Tactics for Goal-Oriented Requirements Elaboration", *24th ICSE*, ACM Press, 2002, pp. 83-93.

[16] E. Letier and A. van Lamsweerde, "High Assurance Requires Goal Orientation", *Int'l Workshop Reqmts for High Assurance Sys*, Essen, September 2002.

[17] R. Lutz and I. Mikulski, "Operational Anomalies as a Cause of Safety-Critical Requirements Evolution", *J Systems and Software*, 2003, pp. 155-161.

[18] R. Lutz and I. Mikulski, "Empirical Analysis of Safety-Critical Anomalies during Operations", *IEEE TSE*, Computer Society, March, 2004, pp. 172-180.

[19] R. Lutz and R. Woodhouse, ``Requirements Analysis Using Forward and Backward Search," *Annals of Software Eng*, 1997, pp. 459-475.

[20] A. Patterson-Hine, et al., "A Model-based Health Monitoring and Diagnostic System for the UH-60 Helicopter," *Am'n Helicopter Society 57th Annual Forum*, AHS, Washington, 2001.

[21] D. Schreckenghost, et al., "Adjustable Control Autonomy for Anomaly Response in Space-based Life Support Systems," *IJCAI-01 Wkshp, 2001.*

[22] *Software Product Assurance for Autonomy On-board Spacecraft*, European Space Agency ESTEC, ftp.estec.esa.nl/pub/tos-qq/qqs/SPAAS/StudyOutputs.

[23] A. van Lamsweerde and E. Letier, "Handling Obstacles in Goal-Oriented Requirements Engineering," *IEEE TSE*, Oct. 2000, pp. 978-1005.

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