The Case for Software Health Management

Ashok N. Srivastava[†]
Johann Schumann[‡]
[†] NASA Ames Research Center
[‡] SGT Inc., NASA Ames

***Abstract*—Software Health Management (SWHM) is a new field that is concerned with the development of tools and technologies to enable automated detection, diagnosis, prediction, and mitigation of adverse events due to software anomalies. Significant effort has been expended in the last several decades in the development of verification and validation (V&V) methods for software intensive systems, but it is becoming increasingly more apparent that this is not enough to guarantee that a complex software system meets all safety and reliability requirements. Modern software systems can exhibit a variety of failure modes which can go undetected in a verification and validation process.**

While standard techniques for error handling, fault detection and isolation can have significant benefits for many systems, it is becoming increasingly evident that new technologies and methods are necessary for the development of techniques to detect, diagnose, predict, and then mitigate the adverse events due to software that has *already undergone* significant verification and validation procedures. These software faults often arise due to the interaction between the software and the operating environment. Unanticipated environmental changes lead to software anomalies that may have significant impact on the overall success of the mission. Because software is ubiquitous, it is not sufficient that errors are detected only after they occur. Rather, software must be instrumented and monitored for failures before they happen. This prognostic capability will yield safer and more dependable systems for the future. This paper addresses the motivation, needs, and requirements of software health management as a new discipline. [1]

I. Introduction

Modern society relies on advanced hardware and software intensive systems, many of which are safety-critical like aircraft, automobiles, medical equipment, and nuclear facilities. A sudden, previously undetected problem can lead to catastrophic failures with potential loss of life. Complex machinery is thoroughly tested and analyzed throughout the design cycle for potential failure modes, a process, which has given society unprecedented access to highly reliable and fault tolerant systems. For example, typical modern jet aircraft engines have very few faults even after $100,000$ hours of operation. Although these systems are highly reliable, engineers continue to monitor the health of the engines and are developing prognostic techniques to estimate the remaining useful life of the engine components and subsystems. These prognostic systems rely on several crucial pieces of information, including real-time sensor readings from different parts of the engine, fleet-wide performance comparisons with other engines of similar make and model, and advanced physics models that are representative of the evolution of the engine performance as a function of time.

Software is often treated differently in the sense that automatic detection, diagnosis, prognosis and mitigation of adverse events due to software is not a common practice. A recent book by the National Research Council on software dependability says that software must be treated as a system component, and that "dependability is not an intrinsic property of software. The committee strongly endorses the perspective of systems engineering, which views the software as one engineered artifact in a larger system of many components, some engineered and some given, and views the pursuit of dependability as a balancing of costs and benefits and a prioritization of risks. A software component that may be dependable in the context of one system might not be dependable in the context of another." [8] As part of this system engineering perspective, it is critical to develop techniques to monitor the health of the software in its operating environment.

Assuming that appropriate fault detection and isolation technologies are available anomalies occurring in the software such as the flight control system can be detected and isolated to continue safe operation. In some cases it is possible to detect faults as they are developing. The ultimate goal of prognostics, or the ability to estimate the remaining useful life of the software system is generally not part of these technologies.

Integrated Vehicle Health Management (IVHM) systems are being developed to detect adverse events during the operation of a complex system (e.g., an aircraft engine), diagnose the root-cause of the problem, and then estimate the severity of the event and its impact on the overall mission of system. In many cases, IVHM technologies are developed to improve the safety of the overall system. However, they can also be used to reduce maintenance costs by enabling condition-based maintenance, a maintenance paradigm where components or subsystems undergo repairs

[1]Published in the Proceedings of the IEEE Conference on Space Mission Challenges for Information Technology, Palo Alto, CA, August 2011.

only when those repairs are needed (e.g., [9]). This is in contrast to scheduled maintenance, where repairs are made regardless of the health of the system. Condition-based maintenance can be more cost effective without sacrificing safety by reducing unnecessary maintenance activities. A health management system consists typically of both hardware and software, working together to determine the current state of health of the host system[2]. An IVHM system monitors the health of the host system through the use of sensors, physics-based models, and data-driven methods to detect, diagnose, predict, and subsequently mitigate the adverse events due to a problem of the system. These steps are defined as follows:

- Detection: The task of detection is to determine whether or not the current state of the host system is operating in an off-nominal condition. This task is difficult because if the host system undergoes complex mode changes during its operation, the characterization of nominal and off-nominal operation requires either a data-driven or physics-based model that accounts for all nominal operational modes.
- Diagnosis: Because most system faults manifest themselves in multiple ways, it is critical to determine the root-cause of the problem. Thus, the diagnosis system must be able to distinguish between potentially hundreds of competing root-causes of the detected problem. For example, low oil pressure and vibration could point to many different problems if looked at separately. Only when considered in combination a worn-out engine bearing can be diagnosed. Depending on the application, diagnosis must be done rapidly in order to enable the estimation of remaining useful life and subsequent mitigation of the adverse event. A critical issue in diagnosis is differentiating between a sensor fault and a fault in the system being sensed. In some real-world scenarios, it has become evident that sensor redundancy is not sufficient to enable this differentiation and that a model of the system may be essential for disambiguation of adverse events.
- Prognosis: An actual fault in the system can, even if correctly detected and diagnosed, lead to a safety-hazard. For example, a broken cog in a rotocraft engine can lead to a fatal crash. Prognostic technology uses the available data and models to estimate the remaining useful life of the system. Thus, the actual occurrence of the fault can be avoided because the part can be replaced before it reaches the end of its useful life.
- Mitigation: Once a fault has been detected and diagnosed, depending on the amount of remaining useful life, an IVHM system could attempt to mitigate this failure to ensure uninterrupted and safe operation. Depending on the severity of the fault and the estimated remaining useful life, this may involve partially automated procedures.

[2]In this article, we refer to the host system as the system, which is undergoing health management. The host system may be comprised of hardware, software, or a combination thereof.

Health management for electrical and mechanical systems is state-of-the-art and is under active research and development in many aerospace and military applications. NASA, the Air Force, and numerous companies such as Boeing, Lockheed Martin, GM, and others from many industrial sectors invest in health management technologies. Even in most modern automobiles, some degree of health management systems can be found. For example, the notorious *check engine* light is the output of a relatively simple engine health management system.

An obvious question arises: if these technologies are being researched, developed, and implemented for hardware systems, why are there no health management systems for software? Software is ubiquitous and will become even more prevalent in coming decades. Should we not have a warning indicating *Please save, your favorite OS will be crashing within 2 minutes with a probability of 95%*? Although such functionality would be convenient and would avoid much nuisance, the situation is much more severe in safety-critical areas. In practically all safety-critical systems (aircraft, nuclear and medical devices, business applications), software plays a prominent role, and this role will become even more important in the future. Because errors in such software can lead to catastrophic failures which can cost human life, developers often expend an extreme amount of effort in developing and certifying highly reliable software. Nevertheless, such software can still have bugs and errors as demonstrated by many examples. So, why can't we simply "hook up" the software to an IVHM system and use that to detect, diagnose, predict, and mitigate the software problems? Unfortunately, this problem cannot be solved so easily. In this paper we will discuss the issues with building such a system and present requirements and and approaches toward developing a *Software* Health Management (SWHM) System.

Before we begin discussing the details of software health management and the issues surrounding this new subject, one may question whether technologies similar to Software Health Management already exist. For example, the familiar Remote Software Upgrade, Fault-Tolerant Computing, and Runtime Verification all appear to address some of the issues in software health management. We briefly touch on these three areas and contrast them with the concept of software health management.

The Remote Software Upgrade process is fairly routinely used, there are some key differences in the schemes. The Internet-based patching and monitoring system does not detect, isolate, and predict the impending consequences of an error, nor does it automatically generate a patch. Computer usage data are analyzed in a semi-automated fashion and then humans primarily generate patches for dissemination back to the machines.

Fault tolerant computing provides techniques and infrastructure (in hardware and software) enable the software system to reliably operate in the presence of faults. The field of fault tolerant computing is very mature; still it does not provide all the necessary properties which are required for SWHM. Perhaps the most important difference between fault tolerant computing and SWHM lies in the missing prognostic capabilities of fault tolerant systems.

Most properties for a software system that need to be shown must be demonstrated during actual software development. Runtime Verification groups together a number of approaches, which provide tools and techniques to detect violations of properties during runtime of the system. Most of these techniques are preventive techniques, some have diagnostics aspects, and most describe automatic recovery mechanisms.

The concept of software health management is distinctly different from the methods listed above: a viable software health management system automatically detects, diagnoses, predicts, and mitigates adverse events due to software errors or errors due to correct operation of software with incorrect environmental information.

II. Software and Software-related Problems

Software and software-related problems are pervasive in modern computer systems. Peter Neumann has assembled a relatively comprehensive list of "Risks to the Public in the Use of Computer Systems and Related Technology." [12] Each item from the list is demarcated with numerous symbols indicating whether the issue can lead to loss of life, loss of resources, whether the issue resulted due to intentional or unintentional misuse, and a number of other factors. We summarize a few key examples from the fields of aeronautics, the automotive industry, medicine, and military systems to discuss some of the key issues that have arisen related to software. The list is long and pervasive thereby motivating many of the technologies discussed in this paper. However, we do not claim that all of these issues could have been resolved with high certainty with an appropriate Software Health Management system, since that would require further analysis of the details specific incident.

A. Software Problems in Aeronautical Systems

Aerospace systems are certainly software intensive—over half of the the cost of a modern aircraft is due to software development. However, the use of software far exceeds just the code running on a single aircraft. The proposed new airspace operations system, known as NextGen [8], will be an extremely software intensive system, and has been called the most complex dynamical system to ever be developed. In this section we highlight some key issues that have arisen either on a single aircraft or in the management of the aircraft in the Global Airspace to show the type and severity of the issues that have arisen in the recent past due to software problems. The list below is certainly not exhaustive but represents a few issues that particularly motivate the development of new software health management technologies.

1) British Airways Flight 027: Error in Terrain Collision and Avoidance System: In June 1999, due to an error in the Terrain Collision and Avoidance System (TCAS) on an aircraft, two Boeing 747 jets came within 600 feet of collision over a remote region of China. Fortunately, the error did not result in fatalities, but the source of the incident points to a significant issue that may be possible to address through the use of appropriate Software Health Management techniques. It is important to note that the software in the system has obviously undergone extensive verification and validation. However, due to an unexpected change in environmental variables, the TCAS system was forced into a mode that could have lead to catastrophic loss of life.

In this incident, a British Airways Boeing 747 and another Korean Air Cargo Boeing 747 were flying in opposite directions in the same airspace with the British Airways flight 2000 feet above the Korean Air Cargo flight. Greenwell and Knight [7] provide an excellent description of the incident as follows, "The TCAS unit installed on the Korean Air jet indicated traffic 400 feet below and approaching head on and shortly thereafter instructed the pilot to climb to avoid the oncoming traffic. In reality, there were no other aircraft in the vicinity of the Korean Air jet except for the British Airways flight 2,000 feet above, and the TCAS units indication and climb instruction were erroneous. The pilot had no way of knowing this, however, as he was operating in a region of airspace without air traffic control service and the cloud layer severely limited his visibility, and thus he followed the climb instruction issued by TCAS. The Korean Air pilot reported that the vertical separation between his aircraft and the phantom aircraft indicated by TCAS decreased to zero before increasing, and before reaching zero TCAS instructed him to increase his rate of climb. The pilot complied and pitched his aircraft further, unknowingly placing it on a collision course with British Airways flight 027, which was now closing in rapidly from above...". The primary source of the problem was determined to be due to damaged circuitry in the TCAS system on the Korean jetliner, which lead to multiple problems in the estimation of the altitude of the British Airways aircraft. The TCAS system, in essence, made correct decisions based on incorrect information coming from part of its circuitry. A software health management system would be valuable in determining whether the incoming environmental variables (the altitude information in this case) are causing it to take actions which are inconsistent with the overall safety profile of the vehicle.

2) Northwest Flight 255: Monitoring System Disabled: A critical aspect of any health management system (HMS), whether it be for software or hardware components, is the fact that the HMS must be engaged in order for it to provide safety value. In the accident described here, a warning system was disabled for unknown reasons. Neumann reports that "…the same pilots had intentionally disconnected the alarm on another MD-80 two days before raises suspicions." [12]. Quoting from the NTSB report[3], the "evidence indicated that the flaps and slats were in the up/retract position and had not been deployed for takeoff. Neither pilot recited the items of the taxi checklist. Stall warnings were annunciated but an aural takeoff warning was not annunciated by the central aural warning system (CAWS). It was confirmed that 28 Volt DC power was not provided to the CAWS power supply #2. The reason for the loss of electrical power was traced to a circuit breaker but no malfunction of the circuit breaker was found." This unfortunate loss points us to an important aspect of a SWHM system: what would check the status of the HM system itself, i.e., *who checks the checker?* The flight ended tragically in the loss of all crew and passengers except for a four year old girl.

3) F-22 Raptors Experience Multiple Computer Crashes: The first test flight of the F-22 Raptor in 1992 ended in a crash at Edwards Air Force base, fortunately without loss of life. The cause of the crash was determined to be due to an error in the flight control software that failed to prevent a pilot-induced oscillation. The first crash of an F-22 in production also points to an issue in the flight control system, which is due to unanticipated environmental conditions: "A problem with a flight-control system caused an F/A-22 Raptor to crash on the runway at Nellis AFB, NV, on Dec. 20, according to a US Air Force report released 08 June 2005. The malfunction of the flight-control system was caused by a brief power interruption to the aircraft's three rate-sensor assemblies, which caused them to fail. The assemblies measure angular acceleration in all three axes: pitch, roll, and yaw. With three failed assemblies, the F/A-22 is not able to fly, investigators said." [17].

Later, "while attempting its first overseas deployment to the Kadena Air Base in Okinawa, Japan, on 11 February 2007, a group of six F-22 Raptors flying from Hickam AFB, Hawaii experienced multiple computer crashes coincident with their crossing of the 180th meridian of longitude (the International Date Line). The computer failures included at least navigation (completely lost) and communication. The fighters were able to return to Hawaii by following their tankers in good weather. The error was fixed within 48 hours and the F-22s continued their journey to Kadena." [16].

B. Mars Spirit Rover

A short time after landing on Mars, the rover Spirit encountered a "reboot loop", where a fault during the booting process caused the system to reboot again. More than 60 reboots per day made any operation of the rover impossible. According to reports a problem in the EEPROM, which is used on board as a file system for intermediate data storage over time was at fault. When this memory storage was filled up, "the boot process failed while trying to read the file system" [1]. A software patch solved the problem and the mission continued. The software for Spirit (and Opportunity) had been developed according to highest reliability standards and rigorous V&V had been performed [13]. Even during flight (before landing), a 10-day (10-sol) test was successfully performed. However, the problem only materialized at Sol 18 [1].

This example shows how, despite careful testing, hard to detect errors can still remain in the software. Furthermore, this example shows that certain kinds of software related failures could be detected by monitoring before the actual fault occurs.

C. Automotive Industry

In recent years, the amount of software used in cars has increased tremendously [3]. Modern cars have dozens of interacting processors, which control many highly safety-critical components like brakes, suspension, engine, and transmission.

Software problems can endanger lives and can cause costly recalls, like the recent recall of defective brakes on Toyota Prius Hybrids. Another software problem, described in [4] concerned automatic cruise control, where under specific circumstances the full throttle was applied suddenly due to a sudden internal mode change in the software. In fact, about 40% of all factory recalls and stalls are due to electrical and electronic problems, which include software and it is to be expected that with the increasing complexity (in particular for electric and hybrid cars), the number of software-related problems will increase.

D. Medical Industry

The medical industry has also experienced significant issues due to software related problems. For example, a software problem in the Therac-25 led to five deaths when the machine erroneously gave radiation levels nearly 100 times the appropriate amount [11]. The machine injured six patients between 1985 and 1987 and lead to the death of two individuals. The safety issues regarding this machine are related to numerous software design, coding, testing, and verification and validation issues. The causal factors of the accidents include, "overconfidence in software, confusing reliability with safety, lack of defensive design, failure to eliminate root causes, complacency, unrealistic risk assessments, inadequate investigations of accident reports, inadequate software engineering practices, software reuse," and other problems [15].

[3]See www.ntsb.gov for more information

At first glance, this appears to be an excellent candidate for a SWHM system. However, it violates a key assumption that the software has been designed and built and passed rigorous verification and validation procedures. The article by Leveson [15] clearly indicates that this system had not undergone an appropriate verification and validation procedure for a safety critical system.

III. Software Health Management as a New Discipline

In the cases described in the previous section, preventative measures have been taken to avoid the problems mentioned. However, there are several critical commonalities among these examples. Each system had gone through rigorous verification and validation testing at multiple steps in the design and implementation process. The errors were due to changes in environmental factors that were unanticipated by the designer. Our point is, while traditionally these errors can be caught and corrected after an incident occurs (sometimes after long manual analysis), we need to develop technologies that can diagnose, predict, and mitigate the effects which occur due to faulty software and hardware interactions as soon as they arise, or even before they they manifest themselves.

In Figure 1 we show a schematic architecture of a software health management system, which monitors the health of an aircraft control system. The SWHM monitors the input and output signals of the software controller, the behavior of the aircraft and the pilot action, and then compares these signals to an internal model of the system. The architecture shown here is very similar to the standard state estimation model used in control theory. A key difference, however, is that this system must monitor mixed (continuous, discrete, and categorical) signals and also compare it with the output of a model which is a high-level abstraction of the system.

The SWHM model shown in Figure 1 is an abstraction of the host system. Thus, it does not contain details of the system dynamics or state equations. However, it may contain information about the host software itself, about the software environment, and about the operating system. Such information will be on different levels of abstraction and can be discrete or continuous. A typical low level piece of information might be the validity or quality of a signal or the occurrence of a "division-by-zero" error. A higher abstracted view could trace the software's timing, stack, or memory usage. On another level, information like CPU load, the length of message queues, or free space in the file system might be used for fault identification and potentially for prognostics. An almost full file system and active processes that write to the file system could serve as a indicator that there might be problems in the near future, a scenario that is somewhat similar to the example presented in Section II-B.

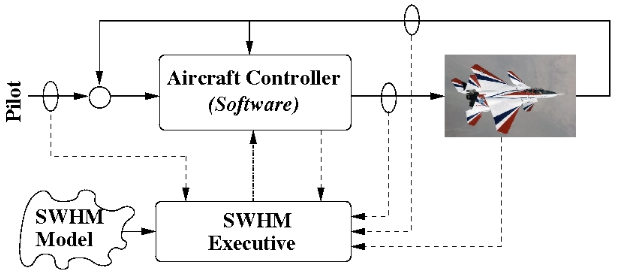

Figure 1. Principal architecture of a software health management system: the top row shows a typical (feed-back) aircraft control architecture. Pilot inputs are mixed with the aircraft sensor feedback signals and fed into the aircraft controller, which is implemented as a piece of software. The software health management system (bottom) will obtain information from the software system itself, the hardware (aircraft), and by monitoring the inputs and outputs of the software (dashed lines). Using its SWHM model, the software health management executive is continuously trying to detect and isolate faults in the monitored system and, if necessary, will issue mitigation or recovery actions (dot-dashed line) to the controller.

A. Detecting Problems with Software Sensors

Sensors produce data about a specific component of the host system and enable anomaly detection. The sensor readings can be discrete, categorical, or continuous measurements. Continuous sensor readings are often evaluated based on a predefined envelope of safe operation. If the readings fall outside the envelope of safe operation (a so-called red-line condition) the system may be in a fault state. In some cases, particularly for aerospace applications and other safety critical systems, redundant sensors are deployed and a voting scheme is used to enable differentiation from sensor faults and faults in the host system.

The SWHM system must detect and identify anomalies in the software execution [10]. These could be triggered errors (e.g., division-by-zero), unexpectedly high memory requirements, unexpected bad numerical accuracy, or the software system making a logical choice, which may be correct, but not the best one for the overall system. The SWHM system must have the capability to *monitor* the software during its execution. We thus speak of *software sensors*, which continuously watch the execution of the software and report data to the SWHM system. The SWHM must also receive data from a multitude of environmental and hardware sensors, as software failures often occur due to the interaction between SW and hardware and unanticipated environmental variables (e.g., turbulence, icing). The concept of an SWHM system can be extended to also deal with problems in the computer hardware (e.g., radiation hit) and anomalies caused by malicious code.

Recently, researchers at Vanderbilt University [5] have developed a model (based on the CORBA Component Model (CCM)), which expresses small software modules as components with inputs, outputs, measured (or sensed) parameters, and the system state. This component model allows complex software systems to be visualized to assess the best locations to sense messages for sensing failures in the software.

Traditional sensors suffer from additive sensor noise which could be due to underlying physical noise sources and can have known distributions. In the case of software, however, the sensors will not have an additive source of noise. However, if the output of a software sensor is a function of data that is processed from traditional sensors, the software sensor will also contain the same noise signal. The procedure for anomaly detection with software sensors can directly follow methods used for detecting anomalies in traditional sensor data. These methods include envelope detection for single sensors or more complex anomaly detection methods for multivariate heterogeneous (i.e., both discrete and continuous measurements) signals [2], [14].

B. Diagnosis and Disambiguation Algorithms

Detection of the anomaly is usually not sufficient to make an accurate prediction of the consequences of a particular anomaly and to fix the problem. Here, the SWHM must be able to diagnose the symptoms and find the cause of the problem, or the most likely cause(s) of the failure. Since the diagnosis component is necessarily model-based and needs to contain knowledge about the underlying combined hardware/software system, all requirements that need to be fulfilled for a traditional system diagnosis system are valid here as well. In particular, the identification should return a rank-ordered list of potential causes of the anomaly, and also should provide a measure of confidence for each diagnosis. In more complex situations, the health management system would need to identify faults in both hardware and software.

A critical element of a health management system are the disambiguation algorithms. These algorithms take sensor data from potentially multiple sources in the host system along with the output of anomaly detection algorithms and produce a list of potential sources of the fault. In almost all cases, these algorithms are passive, meaning that they do not have the ability to actively query the host system to help disambiguate faults. These algorithms often have an underlying abstracted physical model of the host system to help perform diagnosis and disambiguation. For a SWHM system, the challenge is to develop a rigorous model of the host system to enable model-based diagnosis methods. Techniques based on both model-driven and data-driven techniques are also possible and would resemble the methods used for traditional hardware based diagnosis systems. The diagnosis system may benefit from a simulation that is running in parallel to the real software system, which takes the same data as the input but simulates the behavior of the software system to help in fault identification and isolation.

The challenge for correct fault disambiguation can be seen, for example, in the Terrain Collision Avoidance System anomaly in Section II-A1. Here, the SWHM would need to identify the fact that an error had occurred in the damaged circuitry and disambiguate that fault from the event that an actual collision was imminent.

C. Prediction Algorithms

A major benefit of IVHM systems lies in the fact that the system can produce a prognosis about the health of the monitored component. This allows the operator to be much more flexible in maintenance schedules: for example, a jet engine only needs to be replaced when the prognosis system predicts that the engine is still safe to use but it is reaching its end of life and that major component failure might occur soon. In a traditional maintenance regime, the engine is replaced according to a schedule based on operational factors regardless of its actual state. For software, no notion of prognosis in general exists.

The only exceptions are performance prognosis for computer systems or networks (e.g., based upon queueing theory) and general software risk models. All software fault handling technologies are backwards oriented, i.e., they react on faults that already have happened or are imminent.

A critical issue is the development of methods to assess the severity and criticality of an error due to software. Many ideas of reliability theory, such as mean-time-to-failure, need to be translated into this new domain. These include real-time estimation of time-before-failure and the severity of the failure and its impact on the software and hardware systems. For example, in the case of a stack-overflow, one could imagine an estimation of the time to failure as being a function of the number of pushes that occur on the stack. For any finite stack, the number of operations before failure can easily be calculated. Of course, the difficult part of the problem is the estimation of the number of times the push will occur due to the external environment. Other examples of simple software errors that could be modeled for prognostics include memory leaks which can lead to slow but an unbounded increase in processes, or an overflow in the hard drive.

It is imaginable that failures such as the ones on the Spirit Mars rover (Section II-B), where the on-board memory file system became overfull within 18 days and causing a reboot cycle, could be predicted (and mitigated) by a SWHM system.

As with any prediction, the estimation of the certainty in the prediction is key. Note that this estimation must not solely be a function of the known environmental variables. If we were to base our predictions and certainty estimates just on the known environmental variables, we would not anticipate any faults, because we would assume that the environmental variables all stay within their nominal operations.

D. Mitigating the Effects of an Error

If a problem has been detected and it is determined by the SWHM system to be indicative of a substantial error, mitigation strategies need to be employed that depend on the application domain. For example, in the case of the medical equipment described above, an appropriate mitigation strategy may be to simply shutdown the machine if the radiation output level is too high. This decision could be made in real-time and would need to be validated to see if it produces any unwanted side-effects. It turns out that such a system had been designed. However, the warnings were ignored by operators who continued to administer dangerous levels of radiation to the patients. For other systems, such as those on an aircraft, it may not be appropriate to simply shut down the system. In those cases, one could consider automatically generating a patch to help avoid a catastrophic problem until the plane has landed. All state information from the sensors, as well as the history of the system state could be recorded to help analysts reproduce the results. The composition of the existing software and hardware architecture along with the addition of the new patch must undergo some verification and validation process. Technologies to perform such rapid verification and validation must be developed that would ensure the integrity of the resulting system. In some cases, the solution may be chosen from a predetermined set of validated solutions. However, care must be taken in this case because the system, by definition, is running in an off-nominal condition.

IV. Conclusions

Software health management (SWHM) is a discipline that must be developed to address inevitable problems in software intensive systems that have already undergone verification and validation. The field is new and the problems are significant. A mathematical theory of software failure needs to be developed which allows for the development of provably correct algorithms for detecting, diagnosing, predicting, and mitigating the adverse events of software issues.

V. Acknowledgements

The authors would like to thank Eric Cooper, Paul Miner, and the NASA partners working on software health management. This article was written under the support of the NASA Aviation Safety Program Integrated Vehicle Health Management project and NASA's OSMA SARP project "Advanced tools and techniques for V&V of IVHM systems".

References

[1] M. Adler. The Planetary Society Blog: Spirit Sol 18 Anomaly, 2006.
URL http://www.planetary.org/blog/article/00000702/.

[2] S. D. Bay and M. Schwabacher. Mining distance-based outliers in near linear time with randomization and a simple pruning rule. *Proceedings of The Ninth ACM SIGKDD International Conference on Knowledge Discovery and Data Mining*, 2003.

[3] R. N. Charette. This Car Runs on Code. IEEE Spectrum, February 2009.

[4] A. Degani. *Taming HAL: Designing Interfaces Beyond 2001*. Palgrave Macmillan, 2004.

[5] A. Dubey, G. Karsai, R. Kereskenyi, and M. Mahadevan. A Real-Time Component Framework: Experience with CCM and ARINC-653. *IEEE International Symposium on Object-Oriented Real-Time Distributed Computing*, 2010.

[6] R. Filman, T. Elrad, S. Clarke, and M. Aksit. *Aspect-Oriented Software Development*. Addison-Wesley, 2004.

[7] W. S. Greenwell and J. C. Knight. What should aviation safety incidents teach us? Technical Report, University of Virginia, 2003.

[8] D. Jackson, M. Thomas, and L. I. Millett. *Software for Dependable Systems: Sufficient Evidence?* National Academies Press, 2007.

[9] A. Jardine, D. Lin, and D. Banjevic. A review on machinery diagnostics and prognostics implementing condition-based maintenance. *Mechanical Systems and Signal Processing*, 20 (7):1483–1510, 2006.

[10] G. Karsai, editor. *1st International Workshop on Software Health Management (SMH 2009)*, 2009. ISIS, Vanderbilt University. URL http://www.isis.vanderbilt.edu/workshops/smc-it-2009-shm.

[11] N. Leveson and C. S. Turner. An Investigation of the Therac-25 Accidents. *IEEE Computer*, 26(1):18–41, 1993.

[12] P. Neumann. Illustrative risks to the public in the use of computer systems and related technology, 2009. URL http://www.csl.sri.com/users/neumann/illustrative.html.

[13] P. Regan and S. Hamilton. NASA's Mission Reliable. *IEEE Computer*, 37(1):59–68, 2004.

[14] A. N. Srivastava and S. Das. Detection and prognostics on low dimensional systems. *IEEE Transactions on Systems Man and Cybernetics, Part C*, 39(1), 2009.

[15] N. Leveson Software: System Safety and Computers. *Addison Wesley*, 1995.

[16] D. Johnson. Raptors Arrive at Kadena, 2007.
URL http://www.af.mil/news/story.asp?storyID=123041567.

[17] GlobalSecurity.org. F-22 Raptor, 2004.
URL http://www.globalsecurity.org/military/systems/aircraft/f-22-testfly.htm.