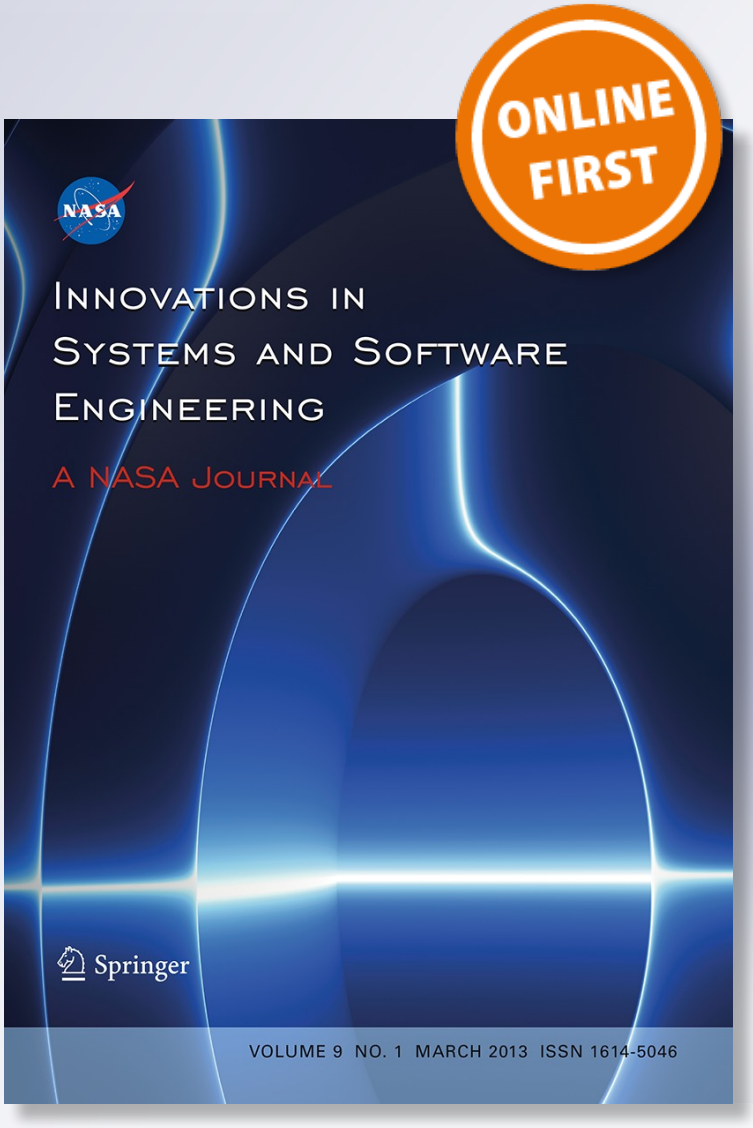
Software health management: a necessity for safety critical systems

Ashok N. Srivastava & Johann Schumann

Innovations in Systems and Software Engineering
A NASA Journal

ISSN 1614-5046

Innovations Syst Softw Eng
DOI 10.1007/s11334-013-0212-0



Springer

Innovations Syst Softw Eng
DOI 10.1007/s11334-013-0212-0

SI: SwHM

Software health management: a necessity for safety critical systems

Ashok N. Srivastava · Johann Schumann

Received: 7 February 2012 / Accepted: 29 April 2013

Abstract As software and software intensive systems are becoming increasingly ubiquitous, the impact of failures can be tremendous. In some industries such as aerospace, medical devices, or automotive, such failures can cost lives or endanger mission success. Software faults can arise due to the interaction between the software, the hardware, and the operating environment. Unanticipated environmental changes lead to software anomalies that may have significant impact on the overall success of the mission. Latent coding errors can at any time during system operation trigger faults despite the fact that usually a significant effort has been expended in verification and validation (V&V) of the software system. Nevertheless, it is becoming increasingly more apparent that pre-deployment V&V is not enough to guarantee that a complex software system meets all safety, security, and reliability requirements. 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, *while* the system is in operation. The prognostic capability of the SWHM to detect and diagnose failures before they happen 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 and motivates the need for SWHM in safety critical applications.

A. N. Srivastava
NASA Ames Research Center, Moffett Field, CA, USA
e-mail: ashok.srivastava@nasa.gov

J. Schumann (✉)
SGT, Inc., NASA Ames, Moffett Field, CA, USA
e-mail: Johann.M.Schumann@nasa.gov

Published online: 22 May 2013

Keywords Software Health Management · IVHM · Verification and validation · Safety-critical software

1 Introduction

Modern society relies on hardware and software intensive systems, many of which are safety-critical systems like aircraft, automobiles, medical equipment, and nuclear facilities. In many of these systems, a sudden problem in the software can lead to catastrophic failures with potential loss of life. To ensure the safety of these complex systems they undergo significant verification and validation procedures throughout the design cycle for potential failure modes. These procedures have 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 h of operation. Although the engine is highly reliable, engineers continue to monitor sensor readings and other information from the engines and are developing prognostic techniques to estimate the remaining useful life of the engine components and subsystems. These prognostic systems rely on much crucial 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. These steps are taken because the engine is a safety-critical element of the aircraft and also because any degradation in engine performance can lead to increased maintenance costs.

Although software plays a systemic role in the operation, performance, and safety profile of an aircraft, in most implementations it is treated qualitatively differently than hardware components of similar importance in an aircraft. In the case of software, it undergoes significant and extensive

verification and validation before implementation, but few safeguards are in place to detect, diagnose, and predict the effects of an adverse event due to software on an aircraft. This mismatch between the two approaches is a primary motivator for this paper.

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" [26]. 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 continued 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.

For hardware platforms, Integrated Vehicle Health Management (IVHM) systems are being developed to detect adverse events during the operation of a system, diagnose the root-cause of the problem, and then estimate the severity of the event 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 which is a maintenance paradigm where components or subsystems undergo repairs only when those repairs are needed (e.g., [27,40]). This is in contrast to scheduled maintenance, where repairs are made regardless of the health of the component. 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. [1] 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 fault 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.

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 [55]. 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

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

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 min with a probability of 95 %*? Although such functionality would be convenient and would avoid much nuisance, the need is much more apparent 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 for several reasons including the fact that the nature of the sensors, underlying dynamics of the system, and other properties are significantly different. In this paper we will discuss the issues with building such a system and present requirements 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 the ability to provide patches and monitor systems over the Internet constitutes Software Health Management. While such technologies exist and are fairly routinely used, there are some key differences in the schemes. For one, not every software application is connected to a network. Additionally, 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. Instead, data are collected about machine performance from the computers under service. These data are analyzed in a semi-automated fashion and then humans primarily generate patches for dissemination back to the machines. The concept of software health management is distinctly different: the SWHM 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.

The remainder of the paper is structured as follows: in Sect. 2, we discuss software and software related problems in aeronautical, automotive, and medical domains to help provide motivation and case material for discussion in the paper. Section 3 describes software health management as a new discipline and lays out the requirements of the field. We define the notions of detection, diagnosis, prognosis, and mitigation in the context of SWHM. In Sect. 4, we discuss similarities and differences between IVHM and SWHM requirements and functionality. We compare and contrast these two areas and discuss ways in which lessons learned from IVHM research can benefit SWHM. A large number of computer science approaches for the detection, removal, and handling of software problems during operation exist. Such techniques range from simple exception handling to fault-tolerant computing to runtime verification and self-healing software, just to mention a few. Although these techniques are useful and important, they do not comprise SWHM, as we demonstrate in Sect. 5. In Sect. 6 we discuss key issues like self-reference (does a SWHM system monitor itself?), as well as software development, V&V, and certification questions. Finally, Sect. 7 concludes and discusses the potential impact of SWHM on complex systems in safety-critical areas and everyday applications.

2 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" [43]. 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 detailed analysis of the specific incident.

2.1 Software problems in aircraft

Aerospace systems are certainly software intensive—over half of the the cost of a modern aircraft is due to software development [62]. However, the use of software far exceeds just the code running on a single aircraft. The proposed new airspace operations system, known as NextGen [26], 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.

2.1.1 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 [23] 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 unit's 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 such a situation, because it would be able to check consistency of the incoming sensor data (the altitude information in this case) and can determine if the current signal values might cause the software system to take actions, which are inconsistent with the overall safety profile of the vehicle. However, low quality and reliability of these checks could cause additional problems as discussed in the following example.

2.1.2 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." [43]. Quoting from the NTSB report [8], 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 HMS 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.

2.1.3 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" [18]. 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." [21].

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 h and the F-22s continued their journey to Kadena." [29].

2.1.4 A380: exploded engine

When one of the four engines exploded during flight of a Qantas Airbus A-380 [2] not only the engine, but also several other subsystems were affected. Among others, several wing tanks had been pierced by debris and hydraulic power was lost. The pilots had to manually sort through "literally hundreds of diagnostic messages"[2] in order to find out what happened. In addition, several diagnostic messages contradicted each other or did not make sense, given the overall state of the aircraft. For example, one message suggested to pump fuel from one galley to another to better balance the aircraft. However, the fuel pumps did not work due the loss of hydraulic power. A health management system, which can perform reasoning *across* subsystems would not have displayed such a diagnostic message, as it was known that there was no hydraulic power. Luckily, the aircraft was flying stable and the pilots had the opportunity to spend several hours on this diagnostics list before they landed.

2.2 Software problems in satellites and spacecraft

2.2.1 Mars Polar Lander: mission lost due to spurious sensor signals

On December 3, 1999, a robotic spacecraft known as the Mars Polar Lander (MPL) was beginning descent into the Martian atmosphere when mission control lost all contact with the craft. An assessment was performed by the Mars Program Independent Assessment Team which concluded that "the most probable cause of the failure was the generation of spurious signals when the lander legs were deployed during descent. The spurious signals gave a false indication that the spacecraft had landed, resulting in a premature shutdown of the engines and the destruction of the lander when it crashed on Mars." [61] The interpretation of the signals from the lander legs was likely performed in software that had been rigorously tested. Although the software had been tested, it behaved unexpectedly due to unanticipated spurious signals. This incident shows that complex software can react in unanticipated ways even after passing verification and validation.

2.2.2 Mars rover Spirit

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 operations of the rover impossible. According to reports [37,59], 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" [2]. 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 [48]. Even during flight (before landing), a 10-day test was successfully performed. However, the problem only materialized at Martian day (Sol) 18 [2]. 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.

2.2.3 LCROSS: excessive fuel burn

In 2009, the Lunar Crater Observation and Sensing Satellite (LCROSS) experienced a significant fuel drain because of a fault in which LCROSS' attitude control system switched to a navigation system that expended extra fuel [3]. The fault was due to a problem in the Inertial Measurement Unit (IMU) and an incorrect persistence counter that caused the the navigation to switch to a star-tracker mode. This mode had a sub-optimal dead-band setting which led to an excessive fuel burn. In all, LCROSS was sent with about 306 kg of fuel but lost about 140 kg of fuel due to the error. Had the fuel margins been slightly smaller this mission could have been significantly compromised due to this error. A SWHM system may have been able to detect this fault and mitigate it to avoid the excessive fuel burn.

2.3 Automotive industry

In recent years, the amount of software used in cars has increased tremendously [11]. Modern cars have dozens of interacting processors, which control many highly safety-critical components like brakes, suspension, and engine control. Software problems can endanger lives and can cause costly recalls, like the recent recall of defective brakes on Toyota Prius Hybrids [58]. Another software problem, described in [15] 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 [1,57] and it is to be expected that with the growing complexity (in particular for electric and hybrid cars), the number of software-related problems will increase.

[2] http://www.aerosocietychannel.com/aerospace-insight/2010/12/exclusive-qantas-qf32-flight-from-the-cockpit/.

2.4 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 [35]. 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 [34].

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 [34] clearly indicates that this system had not undergone an appropriate verification and validation procedure for a safety critical system.

2.5 Security-related software problems

Software security assurance is critical to economic, national, and homeland security. Although a vast number of security vulnerability and hacks are reported and cause substantial economical damage (e.g., due to identity theft, credit card fraud), the safety and security-related areas have been kept quite separate. With software controlling more and more safety-critical systems and those controllers are increasingly accessible through the net, incidents, where breaches in software security actually created problems with a (physical) system have increased. Probably the most relevant example is Stuxnet [12,49]. This highly sophisticated worm, which targets a specific Siemens industrial control software. It is suspected that Stuxnet targeted the enrichment centrifuges in Iranian uranium enrichment facilities and caused physical damage to those devices. With the advent of networked engine controllers and on-board entertainment systems in modern cars, malicious software attacks ("Car-hacking", [38]) can gain remote access to cars, disable cars, and might even tamper with safety-related components like the engine or brakes. Here again, software, which can be remotely accessed, is controlling safety-critical hardware systems.

Malicious intrusion or providing of fake data can get a complex software system into severe problems. For example, it is strongly suspected that the U.S. RQ-170 Sentinel UAS (Unmanned Air System) that was lost over Iran, was captured using GPS spoofing [47]. In this case, the UAS was provided with faked GPS signals, causing the on-board software to guide the plane to an Iranian airport.

Although the safety-critical software components have been designed and validated properly for safety requirements, malicious attacks can disable or change the software in such a way that safety and performance of the overall system can be compromised. Typically such a situation can be detected as a deviation from nominal behavior [39]. Promising results on test examples from the Top 10 malware list [53] reported in [39] makes SWHM a candidate for the detection of security events.

3 Software health management as a new discipline

In the cases described in the previous section, preventive 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 during the design and implementation process. The errors were due to changes in environmental factors that were unanticipated by the designer. There could have been a number of reasons for that. Design and implementation of the software might have been built upon assumptions that are have been violated in this situation. Also hardware failures, to which the software does not respond correctly can be seen as an environmental factor. Last but not least, the system might have been operated in an unanticipated physical environment (e.g., outside the usual temperature ranges, in a dusty atmosphere, or in unexpected high winds). 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 arise. Of course, SWHM is not meant to replace V&V. We assume that the software has already gone through an extensive and rigorous V&V procedure and due diligence has been done to remove as many software errors as possible. Even then, faults can occur due to undetected logical or implementation errors, due to unexpected hardware-software interactions, or due to unexpected situations in the environment. If we want the SWHM system to properly detect and mitigate such problems (and possibly predict them), the SWHM system must be able to dynamically monitor the software system and the related hardware (e.g., sensors). This has to happen, while the software is in operation, for example during the flight, and without disturbing the normal operation of the software under scrutiny. In Fig. 1 we show a schematic architecture of a software health management system, which monitors the health of an aircraft control system. The SWHM has to monitor the input and output signals of the software controller, the

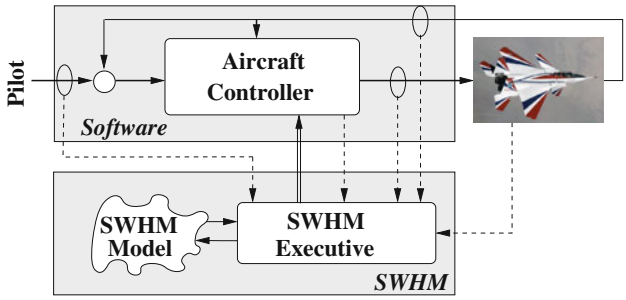


Fig. 1 Example for a high-level architecture of a software health management system: the *top row* shows a typical (feedback) aircraft control architecture. Pilot inputs are mixed with the aircraft sensor feedback signals and fed into the aircraft controller, which is implemented as a piece of software. The software health management system 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 (*double line*) to the controller

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 them with the output of a model which is a high level abstraction of the system.

The SWHM model shown in Fig. 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 to identify of faults 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 Sect. 2.2.2.

3.1 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. 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.

For a SWHM system, the sensors could record data as it passes between software methods, the inputs from the host system and the outputs to the host system, and data regarding the state of health of the host system itself. The SWHM sensors must therefore capture enough information to assess the state of health of the host system because an error or unanticipated event in the host system can potentially lead to a software anomaly. In this case, differentiation between a sensor fault and a fault in the software that is being sensed can be extremely difficult to address, because we cannot directly apply the notion of sensor redundancy.

Recently, researchers at Vanderbilt University [17] have developed a model [based on the CORBA Component Model (CCM)] for software, 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 [7,25,54].

Many prominent system failures have been attributed to sensor failures and the interaction between failed sensors and the software. The accident of the Mars Polar Lander (Sect. 2.2.1) is thought to be due to spurious signals, which occurred during the deployment of the landing gear at high altitude. An appropriate SWHM system should detect that although one set of sensors from the landing gear may indicate that the gear are deployed and the landing feet have touched ground, additional information about the short duration of this signal and cross checks with other sensors of the spacecraft, like the radar altimeter, would have ruled out that MPL has already touched ground, and thus would have avoided premature engine shutdown.

3.2 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 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 is its disambiguation algorithm. Such 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. Diagnosis systems like HyDE [41] use such an approach. Instead of simulating the full physical system, simplified, abstracted models are used for fault detection and diagnosis. Such simulation-based approaches are powerful for hybrid systems with continuous (e.g., physics-based) and discrete components [42], but they require substantial computational resources, which are often not available for embedded control applications.

The challenge for correct fault disambiguation can be seen, for example, in the Terrain Collision Avoidance System anomaly in Sect. 2.1.1. 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.

3.3 Prediction algorithms

A major motivation for the development of IVHM systems lie in the fact that they could predict the remaining useful life of a component or subsystem. This allows the operator to be much more flexible in maintenance schedules: for example, a jet engine would only need to be replaced the estimated remaining useful life is less than some predefined threshold. 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 queuing theory) and general software risk models (e.g., [9]). 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 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 memory usage, or an overflow in the hard drive.

It is imaginable that failures such as the ones on the Spirit Mars rover (Sect. 2.2.2), where the on-board memory file system was filled 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.

It is useful to compare and contrast our view of SWHM with the area of software reliability engineering in which software is made more reliable while having an emphasis on reducing cost and complexity of design and implementation. The software reliability engineering approach puts sets development objectives with related reliability metrics that the team must meet during the design and production cycle to ensure implementation efficiency and reliability. While these are worthy goals for development of a complex software system, the concept of SWHM differs from this because it assumes that the software has already undergone extensive testing but for unanticipated reasons develops a fault. This situation emphasizes real-time fault identification and disambiguation and decision making using in-situ software sensors and advanced diagnostic and prognostic techniques. Even software developed using rigorous software reliability engineering methods can develop faults. The SWHM system should identify and mitigate such faults.

3.4 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. However, 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.

4 SWHM is not IVHM

When considered from a system perspective, a software system is quite different from a hardware system. The mathematical and theoretical background upon which a physical (hardware) system is developed is very mature and has been developed over the course of several hundred years. Together with the theoretical foundations, a century old wealth of knowledge and experience is available for the development of such a system. On the other hand, Software Engineering as a discipline is very young and so has been studied far less. Thus, the design of a reliable, complex mechanical system, e.g., a combustion engine, seems to be much more accessible than for a software system.

One of the most striking differences between software and hardware is the discrete nature of software systems. Software is usually based upon logic and theory of (discrete) computation; everything is represented and handled in form of discrete (e.g., binary) elements. Even if a continuous operation is to be performed by software, all values have to be discretized (e.g., to 32 or 64 bit floating-point numbers) and the calculations itself are sequences of discrete operations on bits.

On the other hand, physical systems work in a continuous domain, unless we go down to quantum mechanics: materials can bend, wheels can slip. This difference to software systems manifests itself in many aspects: for the description and modeling of physical systems, usually differential or difference equations are used. Sets of differential equations describe the possible behaviors of the system over time. This enables the developer to analyze the system and its properties using well-established mathematical theories.

Stability and sensitivity analysis for linear and non-linear systems is a mature field. In many applications, e.g., aircraft design, linear differential equations are sufficient to describe the system behavior. Drawing from a wealth of theory, important safety guarantees (e.g., phase and gain margin for stability) can be formally derived. Even in the non-linear case, formal theorems about stability can be proven, although the underlying mathematical theory is more demanding.

Another, but not less important and powerful approach to describing physical systems is based upon statistical models. A variety of different approaches, e.g., Bayesian or frequentist, enable a data-driven and physics-based analysis.

The theoretical background for software systems is based upon mathematical methods, in particular discrete mathematics (like graph theory or Petri Nets), automata theory, formal systems and languages, mathematical logic, and statistics (e.g., Markov processes). With this wealth of mathematical foundations, strong properties about the (discrete) behavior of software systems can be proven in a formal way.

5 SWHM and other techniques

After discussing the main functionality of a SWHM system, one could argue that SWHM is just “dynamic bug fixing”

and as such, a part of traditional V&V. Indeed, finding an error or fault in a software and fixing it is a major part of all V&V activities. However, the aim of V&V, namely to ensure that a piece of software is free of errors and behaves as required, is substantially different from the goals of SWHM: here, not only the software and its failures are under consideration but also the surrounding hardware and environment as well as the interaction between software and hardware. Also, traditional V&V is supposed to be performed before the system is deployed rather than during its operation.

In this section, we will look at several software engineering approaches, which can be considered to be in the vicinity of SWHM. We will see that each of these approaches can perform certain functionality within an SWHM system, but in isolation are not a software health management system.

5.1 Remote software upgrade

A very common and effective technique to fix software problems after the initial deployment of the system are system upgrades or patches, which are provided by the software vendor and are loaded and installed on the system via some (remote) installation procedure. Pretty much all common operating systems and non-critical software applications nowadays feature remote update capabilities. However, this technique requires human effort for detecting and identifying the failure, and for providing the actual changes in the software.

Additionally, software upgrade can produce its own set of problems, because it can introduce inconsistencies into the software, could be prone to adverse attacks, or open up new software failures. For example, a software upgrade for a Boeing 777 flight computer "could result in anomalies in the fly-by-wire primary flight control, autopilot, auto-throttle, pilot display, and auto-brake systems, which could result in high pilot workload, deviation from the intended flight path, and possible loss of control of the air plane" [19]. This FAA emergency advisory requested that the previous software version to be installed again.

5.2 Automatic software reconfiguration

Automatic software reconfiguration techniques try to automatically change the configuration of a software system in order to mitigate a problem. A wealth of different approaches have been developed to make sure that a consistent state after the reconfiguration is retained and that no problematic transient effects occur. These techniques, however, only cover a part of the requirements set up for SWHM, as they focus on fault removal.

5.3 Autonomic computing

Autonomic computing has some interesting parallels with the proposed SWHM system described in this paper. This paradigm features self-management for the purpose of improving the efficiency and reliability of a complex computer system. The approach, detailed in [60] includes capabilities for self configuration, self healing, self optimization, and self-protection. These elements require similar characteristics to those proposed for the SWHM. The SWHM system is proposed for highly-complex, safety critical, real-time systems. These systems require a very high level of consistency in their operation and must adapt to highly variable environmental conditions. In contrast a typical parallel and distributed computing system may not have the same constraints and operational conditions as those developed for safety-critical systems. However, it is critical for a SWHM implementation to take advantage of the advances made in the Autonomic Computing area.

5.4 Fault-tolerant computing

Fault tolerant computing provides techniques and infrastructure (in hardware and software) to enable the software system to reliably operate in the presence of faults. Typically fault tolerant applications can be found in highly safety critical components, like a control computer for an aircraft or a rocket. Here, multiple computer systems, often even with software developed in different languages and by different groups, are performing the same calculations continuously. An additional piece of highly reliable hardware then analyzes the results, and, if not all computers produce the same result, tries to find the most reliable result by, e.g., using a voting scheme.

Although field of fault tolerant computing is very mature, still it does not provide all the necessary properties, which we require for SWHM. Typically, the check if the redundant system is working correctly is restricted to the result of the computation (e.g., the calculated attitude of the spacecraft). In contrast, SWHM needs to take into account numerous different properties to yield a conclusion on the health of the software. In case a discrepancy is detected in such a fault tolerant system, most often a binary decision (fault, no-fault) is made and rather drastic mitigation methods (e.g., turn off the faulty computer) are taken.

Perhaps the most important difference between fault tolerant computing and SWHM lies in the missing prognostic capabilities of fault tolerant systems. Such systems only react to occurring faults and take appropriate measures; SWHM tries to use its prognostic capabilities to predict certain classes of potential failures even before they occur.

Although there are differences between SWHM as an emerging discipline and fault tolerant computing, an

eventual SWHM system must incorporate the signals from fault tolerant systems and use them in order to improve the overall health state assessment of the system. For example, in a situation where there is software and hardware sensor redundancy, it is critical that the SWHM system reason appropriately about the health state of the system given the inconsistent information. The ability to actively query the underlying system in order to disambiguate true faults from candidate faults can improve the overall effectiveness of SWHM systems while taking advantage of the fault tolerant technology already included on many operational systems.

Fault tolerant systems that use voting schemes to overcome conflicting information can be very beneficial to and can benefit from software health management systems that have an active querying capability. In this case, true faults may be disambiguated because the HM system may be able to issue several different queries to help reduce the ambiguity set and increase the likelihood of identifying the correct course of action.

5.5 Runtime verification

Traditional software development processes require that all verification and validation activities are carried out before software deployment. Some properties, however, cannot be verified at this time, due to the complexity of an a priori validation approach, or since the property depends on information not available during development time. Runtime Verification (RV) addresses this problem by monitoring properties of the software system while it is running and by reporting property violations. Whereas early approaches entirely focused on the discrete portion of the software and were based on the aspect-oriented programming paradigm [20], or code instrumentation (e.g., [24]) for the dynamic check of properties, the field has widened and matured considerably. Techniques described in the proceedings of recent conferences on Runtime Verification [5,32,46] deal—among others—with hybrid systems (e.g., [10,52]) and the detection of security-related events [39] using model-checking, stochastic, and machine-learning based approaches for a multitude of sequential, concurrent, and highly parallel applications.

Thus, RV is providing technology very close to SWHM. In fact, many approaches lend themselves to be used synergistically in the realm of SWHM, where runtime verification methods for monitoring and detection of off-nominal events can be combined with diagnostic reasoning, root-cause analysis, and mitigation techniques. In particular, for the detection of violations in complex temporal properties, RV techniques has a big advantage over traditional FDIR approaches, which often use a highly simplified notion of time.

6 Key issues of SWHM

The development of a SWHM system will have an impact on the entire software design, implementation, and verification and validation process. The SWHM system will assume that the validation steps have been completed and that there are no design or coding errors.

However, there are a number of more subtle issues, which, when addressed properly can substantially help to increase safety and reliability while reducing effort and cost of the software development.

First of all, SWHM should seamlessly work together with traditional V&V tasks. If traditional V&V can reliably show or prove that certain properties of the software cannot be violated, then the SWHM does not have to monitor these specific properties. Furthermore, this knowledge substantially helps the SWHM system with the task of fault identification. If, for example, the implementation of a navigation subroutine could be proven not to produce numerically invalid results ("not-a-number", NaN), then any occurrence of a NaN can be immediately attributed to the incoming sensor signals and no further analysis of this piece of code has to be performed by the SWHM system.

Since SWHM is being proposed as a separate discipline it can be seen to *augment* verification and validation. Due to the high complexity of SWHM, it is still vital to find and remove as many errors as possible during the pre-deployment V&V period, but SWHM can produce an additional layer of safety and reliability for the overall system.

6.1 Verification and validation

Although it is possible that the gain in safety and reliability due to a SWHM system may be significant, it comes at a price. The overall system will be more complex and its safety properties can only be safe and reliable to the extent of the SWHM system. A poorly designed or malfunctioning SWHM system can produce false alarms ("false positives") or can miss important failures occurring in the software ("false negatives"). A false alarm occurs if the SWHM system reports a failure, whereas the monitored system is working flawlessly. A continuously lit "Check Engine" light in the car can be such a nuisance signal. Although false alarms are not primarily a safety concern they can severely impede the system operation by performing unnecessary fixes and reconfigurations. Moreover, they can lead to a situation where operators ignore the output of the system, thereby leading to other potentially significant problems. If the SWHM (or a monitoring system in general) is turned off because of repeated nuisance alarms, real hazards can occur, as discussed in Sect. 2.1.2, where the pilots supposedly disabled the monitoring system.

Thus, a SWHM system must be designed carefully to avoid false alarms and the implementation of the SWHM

itself has to undergo rigorous V&V as errors in the SWHM will not only deliver unreliable results but can cause severe problems in the software to be monitored. Therefore, the SWHM system must be verified and validated to a level of safety and reliability that is at least as high as that for the monitored system.

The literature contains only few papers on V&V of health management systems. For example, Lindsey and Pecheur [36] describe the use of a model checker for the analysis of Livingston IVHM models. The model checker automatically explores the entire state space of the IVHM model and checks properties (e.g., diagnosability), which are formulated in a temporal logic language. Darwiche [14] describes approaches for sensitivity and completeness analysis for IVHM models specified as Bayesian Networks. In general, V&V (and much more so certification) has to address the following two areas: verification/validation on the SWHM model level and code V&V of the actual implementation.

6.1.1 Model V&V

In model-based verification and validation, the model contains all information about the software (its structure, behavior, requirements), the hardware, and their interaction. The SWHM system performs reasoning based on information from hardware and software sensors and other available information to determine the health state of the software. The SWHM system may identify the most likely cause of a failure or make a prediction on the remaining useful life of the system. Thus, V&V has to make sure that the model is *adequate* for the given domain and SWHM requirements and that it is as *complete* and *consistent* as possible. Incomplete models can result in undetected failures; inconsistent models can lead to false alarms.

Besides testing and exhaustive approaches (e.g., [36]), techniques will have to be developed for model analysis and V&V. In particular, as the SWHM models span software and hardware models, and will, in most cases, describe systems on different levels of abstraction, this task will be challenging.

The adequacy of the modeling approach and the reasoning method must be demonstrated. Depending on the SWHM algorithms, this task can be easy or challenging; theoretical results about the reasoning method (e.g., on completeness, soundness, decidability, complexity) must be taken into account.

6.1.2 Code-level V&V

Even in such cases where the model and the reasoning algorithm has been fully verified, no guarantees can be provided yet, because the model and the SWHM system must be implemented as a piece of software. Thus, "code-level V&V", the V&V of the actual implementation has to be performed. Here, the SWHM is treated like a regular piece of software, which has to be tested and validated. In most cases, this will include testing according to tight code coverage criteria, e.g., the MCDC (Modified Condition Decision Coverage) as required by the DO-178B standard [50,51], worst case execution time analysis, stack and memory analysis, etc.

Software Health Management (SWHM) algorithms can contain algorithmic elements, for which current standards do not provide any guidance, for example search during reasoning, non-determinate algorithms, or traversal of large and complicated data structures. V&V guidelines and processes must be developed for such algorithmic elements.

Any SWHM has to take inputs from a multitude of different sources (hardware, sensors, software sensors, operating system, etc.) and potentially interact with the host system on multiple levels. The resulting architecture of a SWHM can therefore be rather complicated, requiring careful V&V. Specifically tailored architectures like the one described in Sect. 3.1 [17] can substantially reduce the effort for SWHM integration.

Certification of safety-critical software is an important and usually a very costly and time consuming process, because it has to be demonstrated to certification authorities (e.g., the FAA for US civil aviation) that the software has been developed according to a given software standard (e.g., DO-178C for civil aviation [51]) and that the SW obeys the required safety requirements.

Software Health Management (SWHM) can not (and should not) lessen the burden of software certification. As mentioned above, SWHM requires that traditional V&V has been performed and as SWHM does not intend to replace V&V. On the other hand, the SWHM system, as a piece of safety-critical software itself, will have to undergo the scrutiny of certification. Unless suitable V&V techniques for SWHM are available, certification might not be possible.

6.2 Applied SWHM

In its primary application, SWHM will provide an additional layer of safety and reliability for safety-critical software systems. The SWHM detects software-related faults, performs diagnosis for root cause analysis, and triggers, if applicable, appropriate mitigation actions. Everything is done in close to real-time while the aircraft is flying. In principle, SWHM can have a much broader applicability beyond a single software system. Like system IVHM does not only perform detection, diagnosis, and prognosis but is increasingly used for condition-based maintenance [56]. This maintenance process takes advantage of the fact that the IVHM system provides concise knowledge about the current state and material

condition of each subsystem. Therefore, repairs or replacements need to be carried out, when a component is defective or has reached its safe end of life, rather than in fixed intervals. Condition-based maintenance has demonstrated its capability to dramatically reduce maintenance costs and to lower environmental impacts.

The use of SWHM across multiple individual software systems on one aircraft or an entire fleet of aircraft can have its benefits. SWHM data assembled across multiple systems can be used to improve quality and reliability of the health management model by, for example, reducing the number of false alarms. Additionally, such data can be used to optimize the overall software and hardware system. In contrast to most log-file based approaches, a SWHM system would be drastically the amount of data to be collected and distributed among the individual systems.

6.3 Current research in SWHM

Whereas IVHM is a mature field, research on the specific topic of software health management is still in its infancy. In addition to topics related to runtime verification (discussed in Sect. 5.5), the two workshops on Software Health Management held during the Conference on Space Mission Challenges for Information Technology (SMC-IT) in 2009 [30] and 2011 [31] give an overview of some of recent approaches toward this topic. Obviously, monitoring of the software, while it is in operation is an important topic of research. Zhao et al. [63] extends the notion of runtime monitors for runtime verification to explore possible fault states of the software in advance (see also [16]). This technique can support prognostics in SWHM. The dynamic monitoring of highly reliable and redundant software poses its own challenges. Goodlow and Pike [22] analyzes a software problem in the Space Shuttle, which was caused by a Byzantine problem and discusses techniques for monitoring ultra-reliable systems [44]. Their tool for the monitoring of embedded systems (Copilot) is presented in this issue.

Other SWHM research focuses on specific software architectures that are particularly amenable for SWHM (e.g., [17]). Some research describes SWHM in architectures that conform to the ARINC 653 standard [6] while others discuss the automated generation of fault trees [33]. Fault trees can also serve as the basis to construct a Bayesian Health Management model as discussed in [13]. Pizka and Panas [45] describes a process-oriented approach to regularly check on the health of a (large) software system. Here, the goal is that regular (non-automated) health checks improve the technical condition of the software and has a positive economic effectiveness. Research on SWHM has also found its way in the development of safety-critical medical systems (e.g., pacemakers) [28] and the military [4].

7 Conclusions

Software health management 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.

Acknowledgments The authors would like to thank Eric Cooper, Paul Miner, Robert Mah, Claudia Meyer, Serdar Uckun, Gabor Karsai, and the NASA partners working on software health management. The authors would also like to thank the reviewers for valuable comments. 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". This paper is a substantially revised and extended version of a paper presented at SMC-IT 2011.

References

1. ADAC: Pannenstatistik (Wikipedia Germany) (2008). http://de.wikipedia.org/wiki/Pannenstatistik
2. Adler M (2006) The planetary society blog: spirit Sol 18 Anomaly. http://www.planetary.org/blog/article/00000702/
3. Andrews D (2011) Managing the bad day. NASA Acad Shar Knowl 44:5–10
4. Associates B (2009) Run-time verification and validation for safety-critical flight control systems. Air Force SBIR/STTR, AF04-246 https://www.afsbirsttr.com/Publications/Documents/Innovation-121109-BarronAssociates-AF04-246.pdf
5. Barringer H, Falcone Y, Finkbeiner B, Havelund K, Lee I, Pace GJ, Rosu G, Sokolsky O, Tillmann N (eds) (2010) Runtime verification—first international conference, RV 2010, 2010. Proceedings, Lecture Notes in Computer Science, vol 6418. Springer, Berlin
6. Barry M, Horvath G (2009) Goal-based flight software health management services (extended abstract). In: Karsai [30]. http://www.isis.vanderbilt.edu/workshops/smc-it-2009-shm
7. Bay SD, Schwabacher M (2003) Mining distance-based outliers in near linear time with randomization and a simple pruning rule. In: Proceedings of the ninth ACM SIGKDD international conference on knowledge discovery and data mining. ACM, New york
8. Board NTS (1989) NTSB identification DCA97MA058, Korean Airlines LTD. http://www.ntsb.gov/ntsb/brief.asp?ev_id=20001213X31759&key=1
9. Boehm B (2007) Software risk management: principles and practices. In: Selby RW (ed) Software engineering: Barry W. Boehm's lifetime contributions to software. Wiley, London
10. Chakarov A, Sankaranarayanan S, Fainekos GE (2012) Combining time and frequency domain specifications for periodic signals. In: Khurshid and Sen [32], pp 294–309
11. Charette R (2009) This car runs on code. http://spectrum.ieee.org/green-tech/advanced-cars/this-car-runs-on-code
12. Cherry S (2012) How stuxnet is rewriting the cyberterrorism playbook. IEEE Spectrum. http://spectrum.ieee.org/podcast/telecom/security/how-stuxnet-is-rewriting-the-cyberterrorism-playbook
13. Codetta-Raiteri D, Portinale L, Guiotto A, Yushstein Y (2012) Evaluation of anomaly and failure scenarios involving an exploration rover: a Bayesian network approach. In: Proceedings of

the 11th international symposium on artificial intelligence, robotics, and automation in space (iSAIRAS-2012)
14. Darwiche A (2009) Modeling and reasoning with Bayesian networks. Cambridge University Press, Cambridge
15. Degani A (2004) Taming HAL: designing interfaces beyond 2001. Palgrave Macmillan, New York
16. Dong W, Leucker M, Schallhart C (2008) Impartial anticipations in runtime verification. In: 6th International symposium on automated technology for verification and analysis (ATVA'08), no. 5311 in LNCS. Springer, Berlin
17. Dubey A, Karsai G, Kereskenyi R, Mahadevan M (2010) A real-time component framework: experience with CCM and ARINC-653. In: IEEE international symposium on object-oriented real-time, distributed computing
18. F-22: F-22 Raptor stealthfighter (1992). http://www.f-22raptor.com/index_airframe.php1992
19. FAA: Airworthiness directive 2005–18-51 (2005). http://rgl.faa.gov/Regulatory_and_Guidance_Library
20. Filman RE, Elrad T, Clarke S, Aksit M (2004) Aspect-oriented software development. Addison-Wesley, Reading
21. GlobalSecurity.org: F-22 Raptor (2004). http://www.globalsecurity.org/military/systems/aircraft/f-22-testfly.htm
22. Goodlow A, Pike L (2009) Toward monitoring fault-tolerant embedded systems (extended abstract). In: Karsai [30]. http://www.isis.vanderbilt.edu/workshops/smc-it-2009-shm
23. Greenwell WS, Knight JC (2003) What should aviation safety incidents teach us? Technical Report. University of Virginia
24. Havelund K, Roşu G (2001) Monitoring Java programs with Java PathExplorer. In: Proceeding of the first workshop on runtime verification. Electronic notes in theoretical computer science, vol. 55(2). Elsevier, Amsterdam
25. Iverson DL (2004) Inductive system health monitoring. In: Proceedings of the 2004 international conference on artificial intelligence (IC-AI'04), CSREA Press
26. Jackson D, Thomas M, Millett LI (2007) Software for dependable systems: sufficient evidence? National Academy Press, Washington
27. Jardine A, Lin D, Banjevic D (2006) A review on machinery diagnostics and prognostics implementing condition-based maintenance. Mech Syst Signal Process 20(7):1483–1510
28. Jee E, Wang S, Kim JK, Lee J, Sokolsky O, Lee I (2010) A safety-assured development approach for real-time software. In: RTCSA. IEEE Computer Society, pp 133–142
29. Johnson D (2007) Raptors arrive at Kadena. http://www.af.mil/news/story.asp?storyID=123041567
30. Karsai G (ed) (2009) 1st international workshop on software health management (SHM 2009). ISIS, Vanderbilt University. http://www.isis.vanderbilt.edu/workshops/smc-it-2009-shm
31. Karsai G (ed) (2011) 2nd international workshop on software health management (SHM 2011). ISIS, Vanderbilt University. http://www.isis.vanderbilt.edu/workshops/smc-it-2011-shm
32. Khurshid S, Sen K (eds) (2012) Runtime verification—second international conference, RV 2011, San Francisco, September 27–30, 2011. Revised selected papers, Lecture Notes in Computer Science, vol 7186. Springer, Berlin
33. Kurtoglu T, Lutz R, Patterson-Hine A (2009) Using auto-generated diagnostic trees for optimized fault handling (extended abstract). In: Karsai [30]. http://www.isis.vanderbilt.edu/workshops/smc-it-2009-shm
34. Leveson N (1995) Safeware system safety and computers. Addison-Wesley, Reading
35. Leveson N, Turner CS (1993) An investigation of the Therac-25 accidents. IEEE Comput 26(1):18–41
36. Lindsey AE, Pecheur C (2004) Simulation-based verification of autonomous controllers via Livingstone Pathfinder. In: Jensen K, Podelski A (eds) Proceedings TACAS 2004, Lecture Notes in Computer Science, vol 2988. Springer, Berlin, pp 357–371
37. Mars Spirit Wiki (2005) Mars spirit software problem. http://c2.com/cgi/wiki?MarsSpiritSoftwareProblem
38. Melone L (2012) Car-hacking: remote access and other security issues. Computer World. http://www.computerworld.com/s/article/9229919/Car_hacking_Remote_access_and_other_security_issues
39. Milea NA, Khoo SC, Lo D, Pop C (2011) Nort: runtime anomaly-based monitoring of malicious behavior for windows. In: Proceedings of runtime verification (RV 2011), LNCS, vol 7186. Springer, Berlin, pp 115–130
40. Mobley R (2004) Condition based maintenance. In: Davies A (ed) Handbook of condition monitoring: techniques and methodologies. Chapman & Hall, London, pp 35–54
41. Narasimhan S (2007) Automated diagnosis of physical systems. In: International conference on accelerator and large experimental physics control systems (ICALEPCS '07)
42. Narasimhan S, Brownston L (2007) HyDE—a general framework for Stochastic and Hybrid model-based diagnosis. In: 18th international workshop on principles of diagnosis (DX '07)
43. Neumann P (2009) Illustrative risks to the public in the use of computer systems and related technology. http://www.csl.sri.com/users/neumann/illustrative.html
44. Pike L, Niller S, Wegmann N (2012) Runtime verification for ultra-critical systems. In: Khurshid and Sen [32], pp 310–324
45. Pizka M, Panas T (2009) Establishing economic effectiveness through software health management (extended abstract). In: Karsai [30]. http://www.isis.vanderbilt.edu/workshops/smc-it-2009-shm
46. Qadeer S (ed) (2012) Runtime verification 2012 (RV'12). pre-proceedings, Springer LNCS, Berlin. http://rv2012.ku.edu.tr/accepted-papers/ (to be published)
47. Rawnsley A (2011) Iran's alleged drone hack: tough, but possible. Wired
48. Regan P, Hamilton S (2004) NASA's mission reliable. IEEE Comput 37(1):59–68
49. Richardson J (2011) Stuxnet as cyberwarfare: applying the law of war to the virtual battlefield. Soc Sci Res Netw. http://ssrn.com/abstract=1892888 or doi:10.2139/ssrn.1892888
50. RTCA: DO-178B: software considerations in airborne systems and equipment certification (1992). http://www.rtca.org
51. RTCA: DO-178C/ED-12C: software considerations in airborne systems and equipment certification (2012). http://www.rtca.org
52. Sistla AP, Zefran M, Feng Y (2012) Runtime monitoring of stochastic cyber-physical systems with hybrid state. In: Khurshid and Sen [32], pp 276–293
53. Sophos: top 10 malware (2008). http://www.sophos.com/security/top-10/
54. Srivastava AN, Das S (2009) Detection and prognostics on low dimensional systems. IEEE Trans Syst Man Cybern Part C 39(1)
55. Srivastava AN, Meyer C, Mah R (2009) Integrated vehicle health management technical plan. Technical report, NASA
56. Stephenson D (2006) The airplane doctors. Boeing Frontiers 5(1):36–41. http://www.boeing.com/news/frontiers/archive/2006/august/ts_sf09.pdf
57. Süddeutsche Zeitung S (2010) Bevor es zu spät ist: Rückrufe in der Automobilbranche. http://www.sueddeutsche.de/automobil/13/503237/text/
58. Toyota: Toyota Prius recall—update ABS software (2010). http://www.toyota.com/recall/abs.html?srchid=K610_p280864979
59. Wikipedia: Mars Rover spirit (2005) http://en.wikipedia.org/wiki/Spirit_rover
60. Wikipedia: autonomic computing (2012) http://en.wikipedia.org/wiki/Autonomic_computing

61. Wilhide P (2000) Mars program assessment report outlines route to success. http://mars.jpl.nasa.gov/msp98/news/news71.html
62. Winter D (2008) Statement of Mr. Don C. Winter, VP Eng & IT, boeing phantom works before a hearing on NITRD. Committee on Science and Technology, U.S. House of Representatives
63. Zhao C, Dong W, Wang J, Sui P, Qi Z (2009) Software active online monitoring under anticipatory semantics (extended abstract). In: Karsai [30] http://www.isis.vanderbilt.edu/workshops/smc-it-2009-shm