Innovations in Systems and Software Engineering manuscript No.
(will be inserted by the editor)

Software Health Management with Bayesian Networks

Johann Schumann · Timmy Mbaya · Ole Mengshoel · Knot Pipatsrisawat · Ashok Srivastava · Arthur Choi · Adnan Darwiche

Received: date / Accepted: date

Abstract Software Health Management (SWHM) is an emerging field which addresses the critical need to detect, diagnose, predict, and mitigate adverse events due to software faults and failures. These faults could arise for numerous reasons including coding errors, unanticipated faults or failures in hardware, or problematic interactions with the external environment. This paper demonstrates a novel approach to software health management based on a rigorous Bayesian formulation that monitors the behavior of software and operating system, performs probabilistic diagnosis, and provides information about the most likely root causes of a failure or software problem. Translation of the Bayesian network model into an efficient data structure, an arithmetic circuit, makes it possible to perform SWHM on resource-restricted embedded computing platforms as found in aircraft, unmanned aircraft, or satellites. SWHM is especially important for safety critical systems such as aircraft control systems. In this paper, we demonstrate our Bayesian SWHM system on three realistic scenarios from an aircraft control system: (1) aircraft file-system based faults, (2) signal handling faults, and (3) navigation faults due to IMU (inertial measurement unit) failure or compromised GPS (Global Positioning System) integrity. We show that the method successfully detects and diagnoses faults in these scenarios. We also discuss the importance of verification and validation of SWHM systems.

Johann Schumann
SGT, Inc., NASA Ames Research Center
Tel.: +1-650-604-0941
E-mail: Johann.M.Schumann@nasa.gov

Timmy Mbaya
RIACS/USRA
E-mail: Timmy.Mbaya@nasa.gov

Ole Mengshoel
Carnegie Mellon University, Silicon Valley Campus
E-mail: ole.mengshoel@sv.cmu.edu

Knot Pipatsrisawat
University of California, Los Angeles
E-mail: thammakn@cs.ucla.edu

Ashok Srivastava
NASA Ames Research Center
E-mail: Ashok.N.Srivastava@nasa.gov

Arthur Choi
University of California, Los Angeles
E-mail: aychoi@cs.ucla.edu

Adnan Darwiche
University of California, Los Angeles
E-mail: darwiche@cs.ucla.edu

1 Introduction: Software Health Management

Modern aircraft increasingly rely on highly safety critical functions (e.g., aircraft control, auto-throttle, autopilot, communications) implemented in software for digital control (fly-by-wire). Despite strict rules for certification (e.g., DO-178B [66] for civil aviation) and immense efforts to perform verification and validation (V&V) on the software during its development, software failures occur, threatening mission, safety and life of passengers and crew. Such failures are typically caused by latent bugs in the code or unexpected hardware-software interactions.

These potentially dangerous failures can occur not only in aircraft, but in other systems as well. Many safety critical applications contain embedded software and face the same type of problems (e.g., cars, medical devices, space applications, and industrial control).

In this paper, we present our approach to *Software Health Management* (SWHM). The SWHM system is powerful on-board software, which has the potential

to *detect* many possible software-related faults in real-time, as soon as they occur while the system is in operation; *diagnose* the root cause(s); and initiate *mitigation* measures. SWHM therefore substantially contributes to overall system safety and reliability. Such a SWHM system

- *monitors the behavior of the software, the operating system, and the attached sensors during system operation.* Information about operational status, signal quality, quality of computation, reported errors, etc., is collected and processed on-board. Much of this information is readily available without the need to further instrument or otherwise modify the software.
- *performs substantial diagnostic reasoning in order to identify the most likely root cause(s) for the fault and provides a quality measure for that result.* Safety critical systems with fast dynamics require a suitable reaction to an off-nominal condition within a very short time frame. Thus, diagnosis with the help of a ground station is in most cases infeasible.
- *provides prognostic information.* In many cases, software failures do not manifest themselves immediately, and prognosis of future problems can improve safety substantially as mitigation can be initiated before the actual fault occurs. For example, a program with an observed memory leak can continue to operate in the short term, but may crash in the long term. An estimate of when all available memory will be used up can be used by an SWHM system to prevent actual memory overflow.

Most aircraft, like other complex machinery, have fault detection and diagnosis systems for all larger hardware components (e.g., power plant, generators, power distribution). These systems (usually called FDDR (Fault Detection, Diagnosis, Recovery) or IVHM (Integrated Vehicle Health Management) continuously monitor the hardware components. If a malfunction occurs, a diagnostic message is displayed in the cockpit. However, such systems currently do not monitor critical software subsystems or software-hardware interactions.

In this paper, we present a Bayesian approach to SWHM. The system health model is developed using a Bayesian network. During monitoring, an efficient SWHM executive performs fault detection and reasoning using probabilistic and statistical methods founded on Bayes' rule.

Our Software Health Management System, which monitors and diagnoses software and interfacing hardware components, has to meet important criteria in order to be useful and safe. In our paper, we will need to address the following issues for SWHM.

The SWHM must provide reliable answers. The notorious "Check Engine" light in a car, which is actually the output of a simple fault detection system, is often not reliable. This lamp is coming on, but a subsequent check at the shop does not reveal any problems with the engine. Such a situation, called a *false alarm,* can reduce the usability of the system (e.g., down time for diagnosis). A *missed alarm*, where the diagnosis system does not recognize a real failure, is more serious. Reliance on the responses of the diagnosis system can lead to serious events, when the monitored component suddenly fails during flight in the case of a missed alarm.

On the other hand, repeated false alarms tend to be ignored by the operator; sometimes even leading to intentional disabling of the health management system. During the investigation of the accident of Northwest Flight 255 [53] it was detected that a circuit breaker for the central aural warning system had most likely been disabled intentionally, because [53] "... the same pilots had intentionally disconnected the alarm on another MD-80 two days before...".

Traditionally, each individual system component has its own fault detection and diagnosis system with its own annunciator. In complex systems, however, single faults can have substantial adverse effects on other subsystems. An integrated diagnosis system must take those effects into account in order to efficiently diagnose the failure.

The recent incident with the Qantas Airbus A-380[1] illustrates this issue. When one of the four engines exploded during flight, not only the engine, but also several other subsystems were affected. 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"[1] 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 to the loss of hydraulic power. An integrated 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 pilot had the opportunity to spend several hours on this diagnostics list before they landed.

The far-reaching impact of one fault can be particularly serious in distributed software systems, which

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

communicate with each other. In most aircraft (and also in cars) multiple software systems, running on individual computers or microcontrollers ("boxes"), exchange data over dedicated buses. Thus, a single software failure can have a dramatic effect elsewhere. For example, "... 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)" [38]. The obvious software error did not only cause a total loss of navigation, but also communication was heavily affected. Luckily, 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" [38].

This again is an indication that SWHM must be able to obtain information from multiple subsystems and it must perform *advanced* diagnostic reasoning. The SWHM and its reasoning must be powerful enough to be able to reliably detect and diagnose important failures. Bayesian networks and the associated advanced reasoning algorithms, discussed in this paper, provide such capabilities. Furthermore, the SWHM system (which is a piece of software in itself) must not fail. Thus, rigorous verification and validation of the SWHM and the health models must be performed. We will discuss approaches for V&V of SWHM systems in this paper.

The remainder of this paper is structured as follows: Section 2 discusses other approaches to monitoring software, performing runtime verification, or to dynamically initiate mitigation and recovery actions. We will discuss several approaches and highlight their advantages and shortcomings.

In Section 3 we introduce our Bayesian Software Health Management approach. We present a brief introduction to its underlying Bayesian networks and reasoning algorithms and discuss how software- and sensor-related data are preprocessed before being fed into the network.

We illustrate our approach using a case study: a small and simplified but typical aircraft control system, running on an emulated real-time operating system, performs GN&C (Guidance, Navigation, and Control) functions for the aircraft. An SWHM system monitors that software system, the operating system, as well the aircraft sensor data. In Section 4, we present an overview of the full system and discuss different failure scenarios that could be properly diagnosed using our Bayesian SWHM system. Results of simulation runs are shown.

Section 5 opens a discussion on specific requirements as well as techniques and tools for the verification and validation (V&V) for SWHM as well as the V&V tasks that have to be performed on the model (i.e., Bayesian network) and on the actual implementation level. Finally, Section 6 discusses future work and concludes.

2 Related Work

When taking a broad view, there is much related research in the area of dependable and secure computing. These techniques can be classified according to the life-cycle phase in which they are applied [5], see Table 1. The life-cycle phase we are primarily focused on in software health management is the deployment phase. Even within this phase there is a number of related techniques and approaches (see bottom of Table 1); we base our discussion in this section on a well-established classification of deployment-phase techniques [5].

2.1 Redundancy-Based Fault Tolerance

Fault tolerance is the ability of a system to preserve, without external assistance, its correct functionalities in the presence of faults [60,3,35,45]. Redundancy is one requirement for achieving fault tolerance [27], and there are two main techniques for achieving redundancy:

1. *Recovery block schemes* serialize redundant software components which have the same function [62,2]. At any given time, one component is activated, such that if a fault is detected, an alternative component is activated to complete the task.
2. *N-version schemes* utilize redundancy by maintaining N versions of the same algorithm, ideally developed independently by N programmers [4,13]. At runtime, all N versions are executed and their results are compared. The underlying assumption of this approach is that the probability of two versions of the program having the same bug is low. Therefore, the chance of any two versions producing the same incorrect behavior is very small, and in aggregate the accuracy is improved.

2.2 Checkpointing and Rolling Back

Another approach to software health management is based on periodically storing the state of a software system, typically in stable storage. Upon failure, computation can be restarted (rolled back) from the latest saved state [19,24], which is called a *checkpoint* [62].

In software systems with multiple components, a failure in one component may require rolling back multiple components to ensure consistency [41]. This situation may give rise to what is known as the *domino effect* [62], in which multiple components are forced to roll back because of a failure of a root component.

2.3 Runtime Monitoring

Runtime monitoring [69] and runtime verification (e.g., [6,7]) are approaches for observing a system's behavior and detecting errors. There is a range of runtime monitoring techniques with varying properties [20]. These techniques include safety property monitoring using linear temporal logic (LTL) [34] and runtime certification [67]. Multiple approaches to performing verification tasks on a software system while it is in operation have been investigated [6,7].

2.4 Trace Analysis

Trace analysis, or log-file analysis, is a technique for analyzing the behavior of software systems from their traces [10]. In contrast to runtime monitoring, trace analysis puts emphasis on analyzing the output generated by a target system, which could be performed during or, most often, after the execution of a program (post-mortem analysis). A variety of methods have been proposed to deal with the challenge of efficiently extracting relevant information from potentially large log files [79,31,30]. Another key challenge in log file analysis is the automatic construction of a test oracle from formal specifications of systems, where a test oracle is a program which determines whether a given system's behavior is correct or not [11,22,54,59,64].

2.5 Software Rejuvenation

Long-running software can suffer from the deterioration of system resources (e.g., memory leaks) and accumulated calculation errors, which may cause a system to fail. Note that these errors often occur in extremely rare cases and may be difficult to reproduce, making them hard to catch during software development. This phenomenon of decreasing software quality/reliability, due to long use/running times, is called *software/process aging* [36].

A well-known example of software aging is a failure in the Patriot missile system's software systems [8]. A Patriot missile battery had been running for a long period of time and some imperfect rounding mechanism in the software (a fault) had caused the internal clock to be off by approximately 0.34 seconds (an error). Such a discrepancy was enough to cause the system to fail to intercept a Scud missile, which was moving at approximately 1600 meters per second.[2] The idea of software rejuvenation is to gracefully restart the aging process at an appropriate time to prevent a fault or an error from turning into catastrophic failures and to maximize computing system availability.

2.6 Built-In Test (BIT)

Built-in test is a methodology in which testing is viewed as an integrated part of software [25,9,84]. This view is to be contrasted with the conventional approach which views testing as a separate process to be built and applied independently of software development. Built-in test requires developers to embed testing code into the software module or class for which the tests are intended. This allows the software to operate in two modes, normal mode and maintenance mode, and testing code is only activated in maintenance mode. One of the main advantages of built-in test, especially when coupled with the object-oriented programming paradigm, is that it increases test reusability [32,84–86]. For example, built-in tests can be inherited like other functionalities in object-oriented programming [86].

2.7 Computer Immunology

Computer immunology is an area inspired by immune systems found in nature (usually the human immune system) [80,26]. When applied to software health management, computer immunology is concerned with the construction of mechanisms that prevent or hinder the system from being compromised by external agents (e.g., viruses or hackers) [33].

Examples of such approaches include the use of patterns in detection of malicious attacks in Windows systems [51]. An intrusion (a fault) is then determined by the number of observed sequences not found in the database. While the detection approach presented in this example is not complete, it is relatively inexpensive and can be easily distributed. Moreover, it can potentially provide protection against unseen abnormal activities. An immune-system inspired approach for fault detection and diagnosis in automotive engines has also been developed [21].

[2] `http://www.ima.umn.edu/~arnold/disasters/patriot.html`

2.8 Self-Healing Software

Self-healing software [28,83,39,40,29] is a relatively new concept that is also inspired by biology. This area of research is concerned with the construction of mechanisms for automatically detecting and mitigating faults, and also automatically recovering from a fault state.

For example, a self-healing mechanism for containing the spread of Internet worms has been developed [15]. In such a self-healing system, each computer is equipped with the ability to detect an infection. Once a computer detects an infection, it broadcasts an alert to other computers in the network, allowing them to filter and block future attacks. This mechanism could effectively limit the spread of the attacks, thus mitigating the extent of system failure.

2.9 Discussion

The techniques discussed above have some commonalities with software health management, which emphasizes fault detection, diagnosis, recovery, and mitigation. For example, approaches such as runtime monitoring, trace analysis, built-in test and computer immunology are typically employed for fault detection and diagnosis. Approaches such as redundancy-based fault tolerance, checkpointing, and software roll-back are, in contrast, typically employed for fault tolerance and fault recovery. Approaches such as software rejuvenation and self-healing software are approaches that can assist in fault mitigation. Similar to many of the above techniques, software health management is also a process that occurs at run-time, and is particularly important as it serves to deal with unforeseen situations that were not envisioned or addressed during the design, development, and testing phases.

3 Bayesian SWHM

3.1 Bayesian Networks

Bayesian networks (BNs) represent multivariate probability distributions and are used for reasoning and learning under uncertainty [57,18]. They are often used to model systems of a (partly) probabilistic or uncertain nature. Roughly speaking, random variables are represented as nodes in a directed acyclic graph (DAG), while conditional dependencies between variables are represented as graph edges (see Figure 1 for an example). A key point is that a BN, whose graph structure often reflects a domain's causal structure, is a compact representation of a joint probability table if the DAG is relatively sparse. In a discrete BN (as we are using for SWHM in this work), each random variable (or node) has a finite number of states and is parameterized by a relatively compact conditional probability table (CPT).

During system operation, observations about the software and system (e.g., monitoring signals and commands) clamp the states of some of the nodes, so-called evidence nodes, in an SWHM BN. Various probabilistic queries can be formulated based on the assertion of these observations to yield predictions or diagnoses for the system. Common BN queries of interest include computing posterior probabilities and finding the most probable explanation (MPE). For example, an observation about the abnormal behavior of a software component could, by computing the MPE, be used to identify one or more components that are most likely to be in faulty states ("root causes").

Different BN inference algorithms can be used to answer these queries. These algorithms include join tree propagation [42,37,77], conditioning [16], variable elimination [43,88], stochastic local search [55,49,50], and arithmetic circuit evaluation [17,12]. In resource-bounded systems, including real-time avionics systems, there is a strong need to align the resource consumption of diagnostic computation with resource bounds [52,46] while also providing predictable real-time performance. The compilation approach—which includes join tree propagation and arithmetic circuit evaluation—is attractive in such resource-bounded systems, since it typically meets those needs.

We note that traditional diagnostics ("flight rules") can easily be modeled as a Bayesian network. Here, many relationships between variables can be described by functional dependencies, which lead to CPT values that are only 0 or 1. With our approach, we are thus able to incorporate flight rules into our models and reason about them. For example, different flight rules (or even parts thereof) can be "weighted" differently and their probability of occurring merged in a principled way. In addition, with an appropriate SWHM model, the Bayesian network can be exploited for sensor validation as well as for diagnosis [48].

3.2 Bayesian SWHM Example

The goal of a Bayesian SWHM technique is to detect and identify active software bugs, which may be software-only or involve both software and hardware. A distinguishing feature of our approach is that we use the well-established technology of Bayesian networks.

A very simple example of a Bayesian network (Figure 1) as it could be used in diagnostics is as follows.

Technique	Purposes	Automation	Resources	Completeness
Design and programming methodologies (design/development phase)				
Model-based design	fault prevention	man	N/A	No
Goal-based operations	fault prevention	man	N/A	No
Aspect-oriented programming	fault prevention	semi-auto	N/A	No
Recovery-based computing	fault prevention, fault tolerance	man	N/A	No
Verification and Validation (V&V) (testing phase)				
Testing	fault removal	man,semi-auto	adjustable	No
Simulation	fault removal	auto	moderate-high	No
Debugging	fault removal	semi-auto	varied	No
Numerical analysis	fault removal	man	low	No
Model checking	fault removal	auto	high	In some cases
Theorem proving	fault removal	auto	high	In some cases
Runtime techniques (post-deployment phase)				
Redundancy-based fault tol. (Sec 2.1)	fault tolerance	auto	varied	No
Checkpointing/roll-back (Sec 2.2)	fault tolerance	auto	varied	No
Runtime monitoring (Sec 2.3)	fault detection	auto	minimal	No
Trace analysis (Sec 2.4)	fault detection, diagnosis	auto	varied	No
Built-in tests (Sec 2.6)	fault detection	auto	minimal	No
Software rejuvenation (Sec 2.5)	detection, rollback	auto	minimal	No
Computer immunology (Sec 2.7)	fault detection, isolation	auto	usually minimal	No
Self-healing software (Sec 2.8)	fault detection, compensation	auto	varied	No

Table 1 Classifications of software health management techniques.

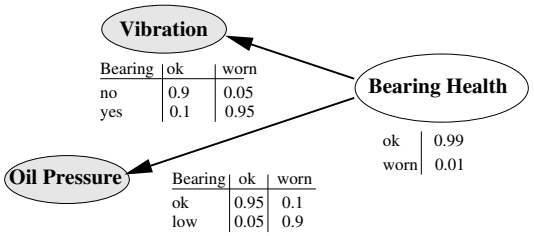

Fig. 1 Simple Bayesian network. CPT tables are shown near each node. The discrete states of each node comprise the rows of the CPT; columns correspond to states of nodes from incoming edges.

We have a node `Bearing Health` (BH) representing the health of a ball bearing in a diesel engine, a sensor node `Vibration` (V) representing whether vibration is measured or not, and a node `Oil Pressure` (OP) representing measured oil pressure. Clearly, the sensor readings depend on the health status of the ball bearing, and this is reflected by the directed edges. The degrees of influence are defined in the two Conditional Probability Tables (CPTs) depicted next to the sensor nodes. For example, if there is onset of vibration, the belief $b(BH \neq \text{ok})$ increases, more formally $p(BH \neq \text{ok} \mid V = \text{yes}) > p(BH \neq \text{ok})$. To obtain the health of the ball bearing, we input (or clamp) the states of the BN sensor nodes and compute the posterior distribution (or belief) over BH. The prior distribution of failure, as reflected in the CPT shown next to BH, is also taken into account in this calculation.

3.3 Bayesian System Health Models

At a first glance, an SWHM system may look very similar to a traditional integrated vehicle health management (IVHM) system: sensor signals are interpreted to detect and identity any faults, which are then reported. Such FDIR systems are nowadays commonplace in aircraft and other complex machinery. It seems like it would be straight-forward to attach software to be monitored (application software) to such an FDIR system. However, there are several critical differences between FDIR for hardware and software health management. Most prominently, many software faults do not develop gradually over time (e.g., like an oil leak); rather they occur instantaneously. Whereas some of the software faults directly impact the current software module (e.g., when a division-by-zero is detected), there are situations where the effects of a software fault manifest themselves in an entirely different subsystem, as discussed in the F-22 example above. For this reason, and the fact that many software problems occur due to problematic SW/HW interactions, both software and hardware must be monitored in an integrated fashion.

Based upon requirements as laid out in Section 1, we are using Bayesian networks to develop SWHM models. On a top-level, data from software and hardware sensors are presented to the nodes of the Bayesian network, which in turn performs its reasoning (i.e., updating the internal health and status nodes) and returns information about the health of the software (or specific components thereof). The information about the

health of the software is extracted from the posterior distribution, specifically from the Bayesian network health nodes.

3.4 Hardware and Software Sensors

Information that is needed to reason about application software health must be extracted from the software itself as well as from all components that interact with it. Examples of this include hardware sensors, actuators, the operating system, middleware, and the computer hardware. Different software sensors provide information about the software on different levels of granularity and abstraction. Table 2 gives an impression of the various layers of information extraction.

The SWHM gets a reasonably complete picture of the current situation only if information from different levels is made available. Such multi-level information therefore becomes an enabling factor for fault detection and identification. Information directly extracted from the application software (Table 2, top) provides very detailed and timely information. However, this information might not be sufficient to identify a failure. For example, the aircraft control task might be working properly (i.e., no faults show up from the software sensors). However, some other task might consume too many resources (e.g., CPU time, memory, etc.), which in turn can lead to failures related to the control task. We therefore extract a multitude of different, usually readily available, information about the software. This information can be of continuous or discrete nature and usually is provided at different rates. Therefore, significant preprocessing of the sensor signals is important.

GN&C Software	
errors	flagged errors and exceptions
memsize	available memory
quality	signal quality
reset	filter reset (for navigation)
Software Intent	
fs_write	intent to write to FS
fork	intent to create new process(es)
malloc	intent to allocate memory
use_msg	intent to use message queues
use_sem	using semaphores
use_recursion	using recursion
Sensors and Actuators	
sensor_signals	signals from hardware sensors
actuator_signals	command signals to actuators
Operating System	
cpu	CPU load
n_proc	number of active processes
m_free	available system memory
d_free	percentage of free disk space
shm	size of available shared memory
l_msgqueue	length of message queues
sema	information about semaphores
realtime	missed deadlines
n_intr	number of interrupts

Table 2 SWHM information sources that are representative of inputs (i.e., evidence) to our Bayesian software health management approach.

3.5 Processing of Sensor Data

In our modeling approach, we chose to use so-called static Bayesian networks, which do not reason about temporal sequences (i.e., dynamic Bayesian networks). Therefore, all sensor data, which are usually time series, must undergo a preprocessing step, where certain (scalar) *features* are extracted (Figure 2). We use standard techniques for extracting features from time series, e.g., sliding window algorithms for calculation of mean values, maximal values, rates, or integrals. A Fast Fourier Transform is used to detect recurrent behavior (e.g., oscillation). Model-specific feature extraction using state estimation with Kalman filters or particle filters is possible, but has not been exploited in this paper. The extracted feature values are then, as we are using discrete BNs, discretized with given fixed thresholds into symbolic states (e.g., `low` and `high`) before being presented to the BN. We do not use continuous values and probability density functions as inputs to our Bayesian networks.

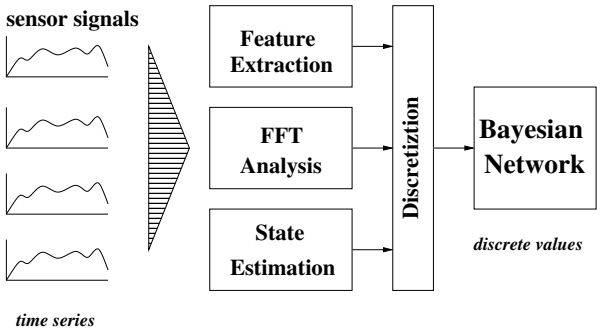

Fig. 2 From continuous sensor readings to input for Bayesian SWHM model: Feature extraction from time-series sensor data, discretization, and input to Bayesian network (or Arithmetic Circuit) SWHM model as evidence. For simplicity, discrete sensor readings and commands that are not discretized are not shown here, even though they are also input to our Bayesian SWHM model.

3.6 Constructing the Model

Nodes. Our Bayesian SWHM models are set up using several kinds of nodes. All nodes are discrete, i.e., each node has a finite number of mutually exclusive and exhaustive states. Figure 3 shows the different node types and their connectivity.

CMD node C: Signals sent to these nodes are handled as ground truth and are used to indicate commands, actions, modes, or other (known) states. For example, a node `Write_File_System` represents that an action, which eventually will write some data into the file system, has been commanded. In Figure 7, `Write_File_System` is an example CMD node. It is used to input, to the BN, a file system operation.[3] The CMD nodes are root nodes (have no incoming edges); see Pearl's discussion of interventions [58]. During the execution of the SWHM model, these nodes are always directly connected (clamped) to the appropriate command signals.

SENSOR node S: A sensor node S is an input node similar to the CMD node. The data fed into this node is sensor data, i.e., measurements that have been obtained from monitoring the software or the hardware. Thus, this signal is not necessarily correct. It can be noisy or wrong altogether. Therefore, a sensor node is typically connected to a health node, which describes the health status of the sensor node (see Figure 3). In another example in Figure 7, `S_File_System` and `S_Queue_length` are example SENSOR nodes, used to input (to the BN) the readings of sensors that reflect the status of the file system and the length of the message queue, respectively.

HEALTH node H: A health node reflects the health status of a sensor or component. The posterior probabilities of the health nodes comprise the output of an SWHM model. A health node can be binary (with states, say, `ok` or `bad`), or can have more states that reflect health status at a more fine-grained level. Health nodes are usually connected to sensor and status nodes. In Figure 7, `H_File_System` and `H_Msg_queue` are example HEALTH nodes, used to compute and output the health status of the file system and the message queue, respectively.

STATUS node U: A status node reflects the (unobservable) status of a software or hardware component or subsystem. Examples of STATUS nodes in Figure 7 are `U_File_System` and `U_Msg_queue`. They reflect the internal status of the file system and the message queue, respectively.

BEHAVIOR node B: Behavior nodes connect sensor, command, and status nodes and are used to recognize certain behavioral patterns. The status of these nodes is also unobservable, similar to the status nodes. However, usually no health node is attached to the behavioral nodes. In Figure 7, `Delay` is an example BEHAVIOR node.

CMD and SENSOR nodes are observable (in other words, input or evidence) nodes; all other nodes are unobservable. The posterior distribution over the HEALTH nodes makes up the essential output for our model.

Our approach inherently handles the identification of multiple faults. Specifically, this is enabled by the inclusion of multiple HEALTH nodes in a model. For example, the model in Figure 7 has two health nodes, `H_File_System` and `H_Msg_queue`. Each health node contains multiple states, and the states can be partitioned into healthy (or normal) states and faulty (or abnormal) states. A fault state constitutes a root cause for observed abnormal behavior. In the simplest case, a health node has two states representing healthy and faulty behavior respectively. For example, in Figure 1 the health node has two states `ok` and `worn`.

Strictly speaking, only input (CMD and SENSOR) nodes and output (HEALTH) nodes are needed in a Bayesian SWHM model. The other node types have been found to be useful but are not required. Employing only CMD, SENSOR, and HEALTH nodes, one would typically create a bipartite BN in which HEALTH nodes are root nodes, SENSOR nodes are leaf nodes, and CMD nodes are either root nodes or leaf nodes. The benefit of going beyond CMD, SENSOR, and HEALTH nodes is that it often leads to SWHM models that are more natural and perform better, both in terms of accuracy and computational speed.

The following informal way to think about *edges* in Bayesian networks are useful for knowledge engineering purposes: an edge (arrow) from node X to node Y indicates that the state of X has a (causal) influence on the state of Y. More generally, the types of influences typically seen in SWHM BNs are as indicated by the following patterns (see Figure 3):

$\{H, C\} \rightarrow U$ represents how status U may be commanded through command C, which may not always work as indicated. This is reflected by the health H of the command mechanism's influence on the status.

$\{C\} \rightarrow U$ represents how the status U may be changed through command C; the health of the command mechanism is not explicitly represented. Instead, imperfections in the command mechanism can be represented in the CPT of U.

[3] If there is a reason that this command signal is not reliable, the command node C is used in combination with a H node to impact state U (Figure 3) as further discussed below.

$\{H, U\} \rightarrow S$ represents the influence of system status U on a sensor S, which may also fail as reflected in H. We use a sensor to better understand what is happening in a system. However, the sensor might give noisy readings; the level of noise is reflected in the CPT of S.

$\{H\} \rightarrow S$ represents a direct influence of system health H on a sensor S, without modeling of status (as is done in the $\{H, U\} \rightarrow S$ pattern). An example of this approach is given in Figure 1.

$\{U\} \rightarrow S$ represents how system status U influences a sensor S. Sensor noise and failure can both be rolled into the CPT of S.

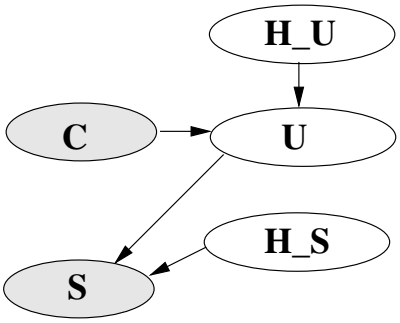

Fig. 3 Example pattern of nodes and edges in an SWHM Bayesian network. Here, the observable nodes are a COMMAND node C and a SENSOR node S. HEALTH nodes are H_S (for the sensor) and H_U (for the STATUS node U). Note how the HEALTH nodes have no parents and are independent given no evidence, but are conditionally dependent given evidence $S = s$. Evidence in the form of a COMMAND $C = c$ does not, on the other hand, create conditional dependency between the HEALTH nodes.

Once the nodes and edges are in place, the conditional probability tables (CPTs) need to be considered. The *CPT entries* are set based on a priori and empirical knowledge of a system's components and their interactions [65, 47]. This knowledge may come from different sources, including (but not restricted to) system schematics, source code, analysis of prior software failures, and system testing. As far as a system's individual components, mean-time-between-failure (MTBF) statistics are known for certain hardware components, however similar statistics are not well-established for software. Consequently, further research is needed to determine the prior distribution for health states, including bugs, for a broad range of software components. As far as the interaction between a system's components, CPT entries can also be obtained from understanding component interactions, a priori, or testing how different components impact each other.

As an example, consider a testbed like NASA's advanced diagnostic and prognostic testbed (ADAPT) [61], which provides both schematics and testing opportunities. Using a testing approach, one may inject specific states into the navigation system and record the impact on states of the guidance system, and perform statistical analysis, in order to guide the development of CPT entries for the guidance system. Setting of software component CPTs to reflect their interactions with hardware can be conducted in a similar way. Clearly, the well-known limitations of brute-force testing apply, and when this occurs one needs to utilize design artifacts, system schematics, source code, and other sources of knowledge about component interactions to develop CPTs. Typical frequencies of operations or internal states should be modeled in detail for the SWHM model to improve reasoning accuracy. For example, in a system, where there are only very infrequent writes into the file system, a low probability $p(write_{FS})$ (compared to $p(read_{FS})$ which is almost 1) would enable the Bayesian network to disambiguate failures related to erroneous writes into the file system against, for example, other software problems or hardware errors.

The knowledge engineering process employed to construct the BNs involves different tasks, which are partly manual and partly automatic. The construction part typically starts from the sensors and commands available as well as the required health states. It is then a matter of creating a BN structure; the patterns discussed above provide helpful heuristics. If there is no or minimal structure, or it is difficult to come up with, a bipartite BN can be created as discussed above. Although it has been demonstrated that Bayesian health models for monitoring an electrical power distribution system can be generated automatically [47], the situation appears to be different for the monitoring of software. An automatically generated health model for the software (e.g., derived from a Simulink model) would likely be too low-level and large for reasoning purposes. Furthermore, the health of individual calculation steps would be of limited use.

A good starting point for the generation of a model could be system and software requirements as well as fault trees. Requirements in the form of flight rules, for example, can be transformed into a Bayesian network as discussed above. In a similar manner, fault trees can be translated into Bayesian networks (see, e.g., [14] for a possible approach). As structural and architectural information has to go into model construction, a fine line between details and abstraction has to be walked; this requires manual modeling work.

3.7 Compilation to Arithmetic Circuits

In our approach, we compile Bayesian networks into arithmetic circuits. An arithmetic circuit is a compact representation of a network polynomial. More specifically, an arithmetic circuit (AC) is a DAG in which leaf nodes represent variables (parameters and indicators) while other nodes represent addition and multiplication operators. Size, in terms of the number of AC edges, is a measure of the complexity of inference. Unlike tree-width, a coarser complexity measure, AC can have significantly reduced complexities, as an AC can take advantage of BN determinism and local structure during compilation, reducing the complexity of inference [12].

Our example network in Figure 1 represents the joint probability $p(BH, V, OP)$ and is shown in Table 3. For simplicity, we replace all CPT entries with θ_x (i.e., $\theta_{ok} \leftrightarrow$ "BH is ok," and $\theta_{\sim ok} \leftrightarrow$ "BH is worn"). Here, we also use indicators λ_x to incorporate evidence when the state of a variable is observed [18], where $\lambda_x = 1$ if x is consistent with our observations and $\lambda_x = 0$ if x is inconsistent with our observations.[4]

BH	V	OP	$p(BH, V, OP)$
ok	v	op	$\lambda_{ok}\lambda_v\lambda_{op}\theta_{v\|ok}\theta_{ok}\theta_{op\|ok}$
ok	v	∼op	$\lambda_{ok}\lambda_v\lambda_{\sim op}\theta_{v\|ok}\theta_{ok}\theta_{\sim op\|ok}$
ok	∼v	op	$\lambda_{ok}\lambda_{\sim v}\lambda_{op}\theta_{\sim v\|ok}\theta_{ok}\theta_{op\|ok}$
ok	∼v	∼op	$\lambda_{ok}\lambda_{\sim v}\lambda_{\sim op}\theta_{\sim v\|ok}\theta_{ok}\theta_{\sim op\|ok}$
∼ok	v	op	$\lambda_{\sim ok}\lambda_v\lambda_{op}\theta_{v\|\sim ok}\theta_{\sim ok}\theta_{op\|\sim ok}$
∼ok	v	∼op	$\lambda_{\sim ok}\lambda_v\lambda_{\sim op}\theta_{v\|\sim ok}\theta_{\sim ok}\theta_{\sim op\|\sim ok}$
∼ok	∼v	op	$\lambda_{\sim ok}\lambda_{\sim v}\lambda_{op}\theta_{\sim v\|\sim ok}\theta_{\sim ok}\theta_{op\|\sim ok}$
∼ok	∼v	∼op	$\lambda_{\sim ok}\lambda_{\sim v}\lambda_{\sim op}\theta_{\sim v\|\sim ok}\theta_{\sim ok}\theta_{\sim op\|\sim ok}$

Table 3 Probability distribution for $p(BH, V, OP)$

According to this joint probability distribution table, the first row ($\lambda_{ok}\lambda_v\lambda_{op}\theta_{v|ok}\theta_{ok}\theta_{op|ok}$) is representing the probability that the health of the ball bearing is okay (BH = ok and $\lambda_{ok} = 1$) and that vibrations and good oil pressure are observed ($V = \text{v}, OP = \text{op}$ and $\lambda_v = 1, \lambda_{op} = 1$). Given the corresponding numerical CPT entries, this probability is calculated as $\theta_{v|ok}\theta_{ok}\theta_{op|ok} = 0.1 \times 0.99 \times 0.95 = 0.09405$, indicating a very low degree of prior belief in such a state. On the other hand, the third row ($\lambda_{ok}\lambda_{\sim v}\lambda_{op}\theta_{\sim v|ok}\theta_{ok}\theta_{op|ok}$) representing the probability that the ball bearing is okay (BH = ok and $\lambda_{ok} = 1$), and there are no vibrations and good oil pressure ($V = \sim \text{v}, OP = \text{op}$ and $\lambda_{\sim v} = 1, \lambda_{op} = 1$) is much higher (85%) and is computed as follows: $\theta_{\sim v|ok}\theta_{ok}\theta_{op|ok} = 0.9 \times 0.99 \times 0.95 = 0.84645$.

Posterior marginals can be computed from the joint distribution:

$$p(BH, V, OP) = \prod_{\lambda_x} \lambda_x \prod_{\theta_{x|\mathbf{u}}} \theta_{x|\mathbf{u}}$$

where $\theta_{x|\mathbf{u}}$ are the parameters of the Bayesian network, i.e., the conditional probabilities that a variable X is in state x given that its parents $\mathbf{U}$ are in the joint state $\mathbf{u}$, i.e., $p(X = x \mid \mathbf{U} = \mathbf{u})$. Further, λ_s are indicators that indicate whether or not state s is consistent with the observations.

Taking the sum of all individual joint distribution entries yields a multi-linear function, referred to as the *network polynomial* f [17]:

$$\begin{aligned} f = {} & \lambda_{ok}\lambda_v\lambda_{op}\theta_{v|ok}\theta_{ok}\theta_{op|ok} + \\ & \lambda_{ok}\lambda_v\lambda_{\sim op}\theta_{v|ok}\theta_{ok}\theta_{\sim op|ok} + \\ & \lambda_{ok}\lambda_{\sim v}\lambda_{\sim op}\theta_{\sim v|ok}\theta_{ok}\theta_{\sim op|ok} + \\ & \lambda_{ok}\lambda_{\sim v}\lambda_{op}\theta_{\sim v|ok}\theta_{ok}\theta_{op|ok} + \\ & \lambda_{\sim ok}\lambda_v\lambda_{op}\theta_{v|\sim ok}\theta_{\sim ok}\theta_{op|\sim ok} + \\ & \lambda_{\sim ok}\lambda_{\sim v}\lambda_{op}\theta_{\sim v|\sim ok}\theta_{\sim ok}\theta_{op|\sim ok} + \\ & \lambda_{\sim ok}\lambda_v\lambda_{\sim op}\theta_{v|\sim ok}\theta_{\sim ok}\theta_{\sim op|\sim ok} + \\ & \lambda_{\sim ok}\lambda_{\sim v}\lambda_{\sim op}\theta_{\sim v|\sim ok}\theta_{\sim ok}\theta_{\sim op|\sim ok}, \end{aligned}$$

or in other words

$$f = \sum_{\mathbf{x}} \prod_{\lambda_x} \lambda_x \prod_{\theta_{x|\mathbf{u}}} \theta_{x|\mathbf{u}}$$

where the summation is over assignments $\mathbf{x}$ of all variables $\mathbf{X}$, i.e., each row of the joint probability table. When there are no observations, this summation will evaluate to one.

An arithmetic circuit is a compact representation of a network polynomial [18], which clearly are exponential in size and thus unrealistic in the general case. Answers to probabilistic queries, including marginals and MPE, are computed using algorithms that operate directly on the arithmetic circuit. A bottom-up pass over the circuit, from input to output, evaluates the probability of a particular evidence setting (or clamping of λ parameters). And a subsequent top-down pass over the circuit, from output to input, computes partial derivatives. From these partial derivatives one can compute many marginal probabilities, provide information about how change in a specific node affects the whole network (sensitivity analysis), and perform MPE computation [17,18].

[4] For example, if we observe a vibration on sensor V, then $\lambda_v = 1$ and $\lambda_{\sim v} = 0$; if we observe no vibration on sensor V, then $\lambda_{\sim v} = 1$ and $\lambda_v = 0$; if we have not yet observed the sensor V, we leave $\lambda_v = 1$ and $\lambda_{\sim v} = 1$

4 Case Study: Aircraft Control System

4.1 System Architecture

In order to conduct realistic experiments, we used a simulation of a real-time architecture platform reflective of timing and memory constraints of real-time execution in embedded systems. All flight software simulations, including AC computation, have been implemented in C, and a real-time operating system emulator for OSEK,[5] Trampoline,[6] was used. While the OSEK Real-Time Operating System (RTOS) is mostly used in the automotive industry, we decided on this RTOS platform rather than other RTOSes well established in the aerospace industry such as Wind River's VxWorks[7] or GreenHills' INTEGRITY[8] because its basic functionalities and availability were sufficient for the purpose of our experiments [76].

System Processes. The experiments are run on an RTOS emulator, which schedules the flight software and SWHM processes to run at fixed rates. For simplicity, the simulation model of the plant was integrated into a single task running as one of the scheduled OSEK tasks (see Figure 4). Hardware actuators and sensors are not modeled in detail in order to avoid addition of drivers and interrupts routines. This experimental architecture, notwithstanding its simplicity, is sufficient to run simple simulations of aircraft dynamics and GN&C software within real-time requirements (fixed time slots, fixed memory, inter-process communication, shared resources).

Our architecture consists of the following three relevant tasks: the controller or GN&C (Guidance, Navigation and Control) task; the plant (aircraft or spacecraft) task; and the inference engine SWHM task. The aircraft plant task simulates a simplified aircraft model whose state space is characterized by its Euler angles, velocity, acceleration, fuel mass, and altitude estimation. The simulated fighter jet incorporates guidance, navigation, and control software into a single controller task implemented in C [76]. The plant task is run by the RTOS every 20ms; the GN&C task and the SWHM inference engine are run every 500ms. Listing 1 shows how our SWHM is integrated as an OSEK task. This task, which is scheduled to run every 500ms first reads sensor values, preprocesses them (not shown), and inputs them as evidence (observe()) to the Arithmetic Circuit. Then proper reasoning is carried out in several steps by evaluating and differentiating the Arithmetic Circuit, updating the network nodes, and extract the posteriors of the health nodes, which are placed back into the system memory. All accesses to the global memory have to be protected by calls to the OSEK functions GetResource and ReleaseResource.

Listing 1 Code for core functionality of the SWHM executive as an OSEK task, which includes operations on the arithmetic circuit model (compiled from the Bayesian network).

```
TASK(T_iswhm_500ms) {
    t_iswhm_data mySWHM_data;
    t_ISHWM_posteriors mySWHM_posteriors;

    GetResource(global_data);
      mySWHM_data = SWHM_data;
    ReleaseResource(global_data);
    preProcess(mySWHM_data);
    observe(varIndex("Sensor_FileSystem"),
            mySWHM_data.sensor.sensor_FS);
    ...
    evaluate(); differentiate(false);
    evaluationResults();
    mySWHM_posteriors = getPosteriors();
    getResource(global_data);
      SWHM_posteriors = mySWHM_posteriors;
    ReleaseResource(global_data);
    TerminateTask();
 }
```

System Architecture. The complete SWHM development process and system architecture implemented to run experiments is depicted in Figure 4. The SWHM model of the whole aircraft system was developed as a Bayesian network using the tool SamIam[9] and compiled into an arithmetic circuit, using UCLA's ACE[10] Arithmetic Circuit compiler package. The resulting data structure is integrated with the rest of the system (tasks including controller, plant, and the inference engine) running on the OSEK operating system. The Bayesian network model is compiled offline into an Arithmetic Circuit serving as the knowledge base—amortizing compilation overheads with real-time execution in the running system.

4.2 Experimental Scenarios

Based on well-known actual software-related incidents in the area of aerospace systems [81,75], a number of relevant scenarios were designed for the purpose of conducting experiments with our implementation of the SWHM approach. The most relevant scenarios include:

– file system related faults,

[5] Open Systems and their Interfaces for the Electronics in Motor Vehicles; http://www.osek-vdx.org/

[6] http://trampoline.rts-software.org/

[7] http://www.windriver.com

[8] http://www.ghs.com

[9] http://reasoning.cs.ucla.edu/samiam/

[10] http://reasoning.cs.ucla.edu/ace/

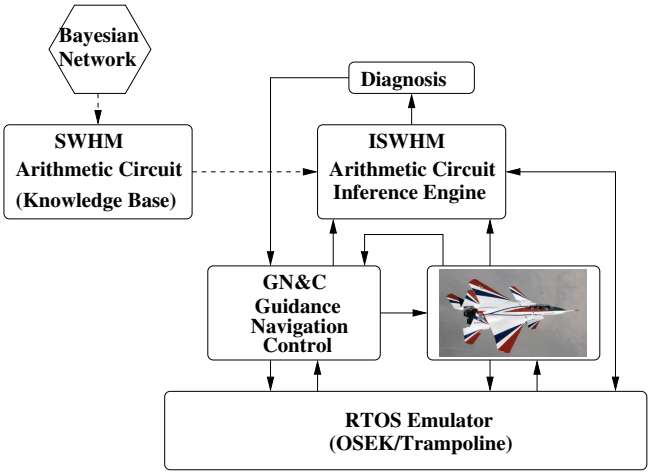

Fig. 4 System Architecture and process for aircraft/spacecraft SWHM: the Bayesian Network model is compiled into an arithmetic circuit representing the knowledge base. The OSEK real-time operating system schedules three main tasks: the controller, the plant, and the SWHM inference engine.

- digital signal processing and handling faults,
- IMU or compromised GPS integrity navigation faults,
- software signal inversions/crossing,
- dead-band overlapping faults,
- signal bias/quality faults,
- resource allocation faults,
- non-matching signal ranges,
- priority inversion faults,
- transient signals faults,
- Euclidean geometry navigation faults,
- dateline crossing faults, and
- Euler angle computation faults.

In the following sections we will, in detail, discuss three fault scenarios: aircraft file system-based faults, signal handling faults, and navigation faults due to IMU failure or compromised GPS integrity.

4.2.1 Scenario I: Aircraft File System-based Faults

Many aircraft systems rely on subsystem communication through MIL-STD-1553 bus controllers, which can also hold messages in a buffer before transmission to the target terminal. Some real-life incidents involving aerospace systems, such as the failure of the Mars rover Spirit due to continuous reboot, have been caused by software failure of on-board data storage or file systems. The incident involving the Mars rover Spirit was attributed to an overflow of the embedded storage file that resulted in a reboot cycle [1]. Such incidents motivate the simulation of file system related fault scenarios for SWHM demonstration purposes. These scenarios are based on a simulated file system and involve delay, overflow, and similar problems. For example, writing to a full file system or buffer might have ripple effects throughout an avionics system by causing delays in subsystem communication, which again may result in induced aircraft oscillations, shutdowns, or other faults. For our experimental purposes, a flawed software architecture was designed with a global message queue that buffers all sensor, controller, and science camera signals, and logs them in the on-board file system before transmitting (Figure 5).

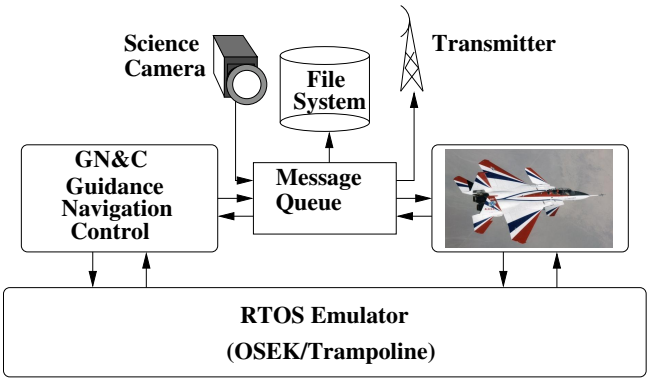

Fig. 5 Software architecture for file system related fault scenarios.

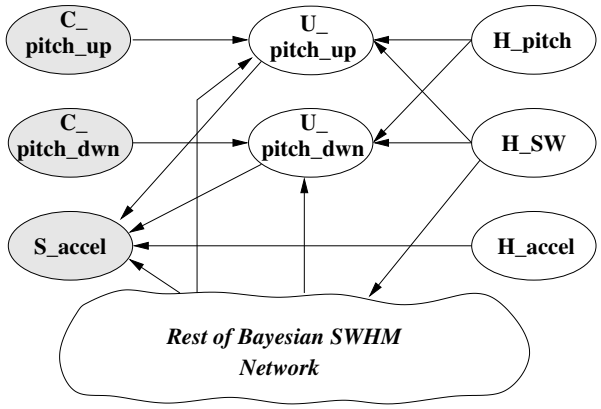

Fig. 6 Bayesian SWHM model (excerpt) for basic pitch dynamics.

The SWHM model. Figures 6 and 7 show the relevant excerpts from our SWHM model for this scenario. Whereas the network in Figure 6 focuses on the basic mechanisms of pitching up and down, together with the IMU accelerometer sensor, Figure 7 deals with the part of the system consisting of the file system and the message queue.

The SWHM model for the monitoring of pitching (Figure 6) receives inputs from the flight control system (commands `C_pitch_up` and `C_pitch_dwn`). The accelerometer is the only sensor input in this network. Since it can fail, it has an associated health node `H_accel`. The given commands influence the (unobservable) pitch-up and pitch-down states of the aircraft, which, in turn have an influence on the acceleration (if the aircraft pitches up, an increased acceleration can be measured). Health and consistency of the pitch states is controlled by the health nodes for the pitch mechanism and software health `H_SW`.

The software sensor nodes for the Bayesian SWHM model for the file-system relevant parts (Figure 7) are again located on the left-hand side of the network: a sensor to detect writes to the file system, a sensor providing information on how full the file system is (with states: `empty`, `medium`, `almost-full`, and `full`). A similar sensor (`S_Queue_length`) provides information for the message queue. Finally, `S_Delta_queue` receives information if the length of the message queue is increasing or decreasing. The oscillation sensor in this SWHM uses accelerometer or IMU data as raw inputs. During preprocessing, a Fast Fourier Transform is used to obtain the current frequency spectrum of vibrations and oscillations. Oscillations are detected if the amplitude of the low-frequency content is above a given threshold.

Our SWHM system has to detect and disambiguate faults through real-time automated reasoning in different failure scenarios such as the ones reported in [76]:

- A sudden induced oscillation of the aircraft occurs, but there are no pilot inputs. The underlying causal scenario, which our SWHM system has to detect is the following: the on-board file system or data storage is almost full. Writing the messages in the message queue thus takes substantially more time and causes delays in the control loop, which eventually start dangerous oscillations of the entire aircraft. In this scenario, the reasoning capability of our SWHM system is critical, as oscillations occur in the aircraft, though the root cause originated in the file system. Furthermore, no single component raised an error flag.
- The source of oscillation might also be pilot inputs. The SWHM reasoner is to disambiguate by evaluating whether the fault is due to Pilot Induced Oscillations (PIOs) or rather one or more flight software failures. For our experiments, the SWHM monitors pilot's control stick movements (not shown in Figure 7) because PIO might result from faulty flight software where gain parameter values are incorrectly preprocessed, for instance.
- In a science Unmanned Aerial System (UAS), signals may be dropped or system delays may result from the science camera taking a large number high-resolution images that need to be sent by a transmitter with inappropriate bandwidth. Our SWHM model needs to detect such failures involving resource competition and blocking. Non-trivial reasoning is important, because such failures can manifest themselves in seemingly non-related manners (aircraft oscillation, for example). The SWHM model needs to disambiguate between a message queue overflow resulting from non-matching transmit receive rates during high activity, bugs in the flight software itself, or failure of the unit's hardware.

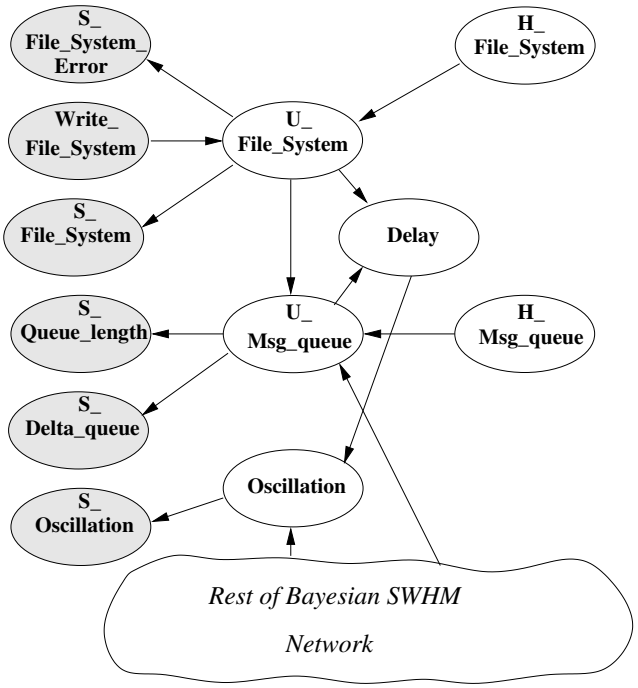

Fig. 7 Bayesian SWHM model (excerpt) for file-system and message-queue related software components.

Experiments and Results. Experimental runs show that the system being monitored runs fine in the nominal case as shown in Figure 8. Several pitch commands are given by the flight management system (top panel), which result in stretches of positive and negative vertical speed and which increase or decrease the aircraft's altitude (middle panel). Pitch commands and accelerometer signals, which are input to the health model as shown in Figure 6, result in the posterior probabilities shown in the bottom panel. A value close to 1 indicates that the accelerometer, the pitch component, and the control software are healthy.

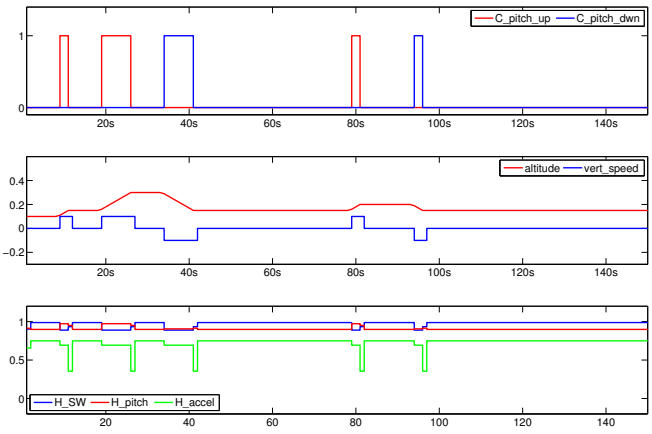

Fig. 8 Temporal trace for the nominal case of file system based scenarios. The degree of belief in the health of the system software, in blue graph in the bottom panel, remains high.

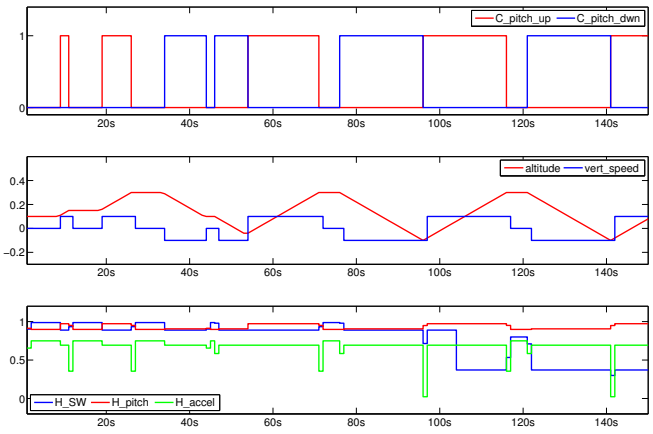

Fig. 9 Temporal trace for a file system related fault scenario resulting in oscillations. The SWHM inference engine's evaluation indicate that the degree of belief in the health of the system's software (blue in bottom panel) substantially drops when oscillations are detected at about $t = 100$, after overflow of the file system resulted in commands stalling in the message queue and delayed pitch up and pitch down command signals from the controller. Readings from the altitude sensor(blue in middle panel) show oscillating altitude starting at about $t = 30s$.

In an illustrative file system-induced fault scenario [76], the flight controller pitch-up and pitch-down commands to the aircraft plant are affected by faults originating in the file system, causing the aircraft to oscillate up and down rather than maintaining the desired altitude. For the purpose of our experiments, we set the file system to almost full at the start of the run (which will cause commands to accumulate in the message queue), no additional pilot inputs were assumed, and the pitch and accelerometer systems were assumed to work with no or negligible faults.

As the system runs and control input commands are issued and logged, delays in executions start taking place after 30 time units (Figure 9). Eventually, altitude oscillations are detected by a Fast Fourier Transform and reflected by the altitude sensor as shown in the middle panel of Figure 9. The SWHM then infers that the posterior probability of nominal health of the software is low, as the probability substantially drops while the health of pitch and accelerometer systems are mostly high despite some transient lows. This indicates a low degree of belief in the health of the software and that the most likely cause of oscillations would be a software fault. Note that in this scenario the health of the file system `H_File_System` and of the message queue `H_Msg_queue`, when considered individually, do not drop significantly.

4.2.2 Scenario II: aircraft/UAS Signal handling faults

Inappropriate software signal handling, such as handling erroneous data due to values outside the valid range, can have catastrophic impacts as the Ariane-V incident dramatically illustrates [87]. Defective software has also led to incidents as experienced by Qantas Flight 72 Airbus A330-300's "sudden [uncommanded] nose down" caused by defective control software in its Air-Data Inertial-Reference Unit (ADIRU) [68]. These incidents motivate experiments with a fault scenario involving the control software's inappropriate handling of signals from a radar altimeter; we use simulated digital signal processing (DSP) and Inertial Measurement Unit (IMU) software for the purpose of SWHM demonstration. In this scenario (Figure 10) the Radar Altimeter input signals are received and processed by signal handling software within the digital signal processing task. The signal handling software feeds the processed signals to the Auto-Lander software system's IMU, which will use a Kalman filter altitude estimation in order to plan a smooth descent of the aircraft. In addition, the estimated output state of the IMU is corrected over time with input from Global Positioning System (GPS) navigation, to correct integrated errors inherent to the IMU state estimation. In our simulation, the divergence between IMU and GPS is also tracked through additional software (`U_IMU_vs_GPS_conv` node in the Bayesian network model). A separate software module—similar to a Watchdog timer—monitors whether the output state vector from the IMU reaches some short term goals within a time limit in its waypoint trajectory after the

Auto-Lander system is engaged and the plant's descent glideslope is initiated.

Depending on the environment, the terrain, and hardware health, the Radar Altimeter will send good or bad (noisy) input signals to the DSP's signal handling software. However, the software handling the radar altimeter signal might have some undetected engineering defects. For the purpose of our demonstration, that software can fail in the presence of very noisy input signals and resort to using the last good altimeter reading. This injected fault is modeled in spirit after an actual mishap with an AV-8B Harrier fighter jet's AutoLander at NASA Ames Research Center (ARC) (personal communication). That software problem almost caused a crash during a flight test when the radar altimeter failed at a low altitude ($\approx$ 20 meters) while on an an Auto-Lander descent. The DSP signal handling software kept returning the last good reading of 20 meters, while the aircraft actually kept descending to lower altitudes as the Auto-Lander assumed it was still 20 meters off ground. A crash was averted by the pilot taking over control a few feet from the ground based upon visual localization and cues.

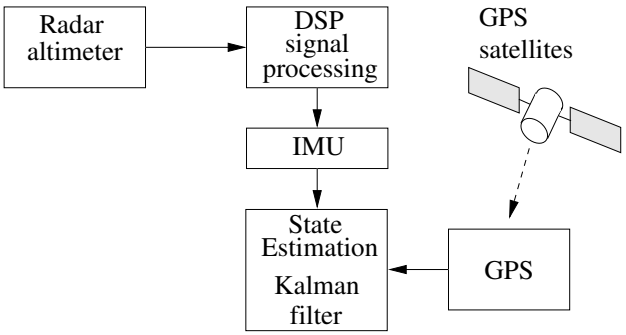

Fig. 10 Signal Handling fault scenario.

The SWHM model. For this case study, the system architecture to be monitored was modeled with a Bayesian network as shown in Figure 11 (excerpts). The relevant nodes are as follows:

- Auto-pilot/Take-off/Landing Control command sensor node (for simplicity not shown in Figure 10)
- Radar altimeter status, sensor, and health nodes
- Signal handling control software health node
- IMU navigation confidence (Kalman filter altitude estimation) sensor node
- IMU navigation health node
- Radar noise status and sensor nodes
- GPS navigation software health node (and relevant GPS nodes)
- IMU and GPS divergence node
- Terrain irregularity grade

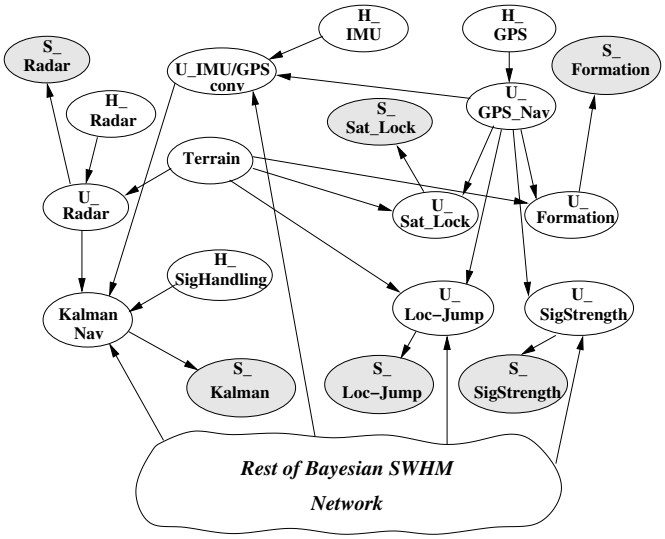

Fig. 11 Bayesian SWHM model for UAS signal handling (Section 4.2.2 and GPS navigation (Section 4.2.3) (excerpts).

The SWHM inference engine uses evidence from hardware and software sensor nodes, along with the SWHM model, to compute in real-time beliefs that reflect:

- whether the altitude estimated from the IMU (with Kalman filtering) is likely to be correct or not,
- whether to initiate altitude estimation from backup control software fed with GPS information rather than the IMU (and Kalman filtering) control software,
- whether the DSP signal handling software health affects altitude estimation,
- whether altitude estimation is affected by hardware health, and
- whether the GPS navigation software or the IMU navigation software is the likely cause of the excessive divergence or convergence of both measurements.

In cases such as the specific incident involving Qantas Airbus A330-300 mentioned above, SWHM would be an approach to evaluate the likelihood that control software such as the Air-Data Inertial-Reference Unit (ADIRU) is failing and provide real-time diagnosis and mitigating solutions. The following three experimental scenarios can be run with this architecture:

- The aircraft cruises at low altitude over very irregular terrain and the Auto-Lander is engaged for vertical landing. The radar altimeter signals have much noise, which might cause signal processing and handling to fail and feed erroneous input to the IMU for state estimation.

– The Auto-Lander is engaged and the craft's descent glideslope is initiated, but the IMU output state's altitude is inconsistent with expected values and GPS navigation.
– The GPS navigation state experiences a relative location jump and excessive lock satellite signal strengt (possibly from spoofing, see Section 4.2.3), which results in inconsistent values or divergence IMU and GPS state estimation.

Experiments and Results. Experimental runs of the fault signal handling scenario previously discussed—which includes the first two scenarios above—show that the system being monitored runs fine in the nominal case as shown in Figure 12. In the nominal case, when the radar signals have low noise, assuming hardware, IMU and GPS navigation have no faults, the SWHM inference engine reports—given reasoning from sensory evidence—a high degree of belief in the health of the signal processing and handling software and relatively good health of the IMU and GPS navigation. The degrees of belief in the health of the IMU and GPS (Figure 11 nodes `H_IMU` and `H_GPS` respectively) are not as high as the belief in the health of the signal handling software (node `H_SigHandling`). This is due to the conditional relationship of IMU and GPS, see the BN structure in Figure 11.

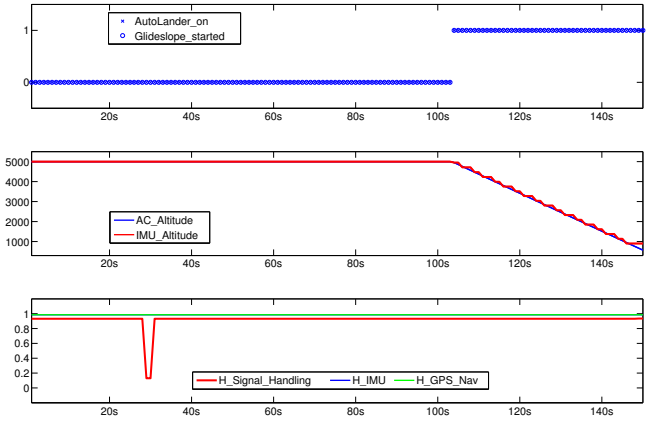

Fig. 12 Temporal trace for the nominal case of Signal Handling scenarios. When the auto-lander is engaged and the glideslope initiated (top panel) from an initial altitude of 5000ft around time $t = 105s$, the IMU returns an altitude estimation closely matching the aircraft's physical altitude. The degrees of belief in the signal handling software, IMU, and GPS navigation's health remain high (bottom panel). A short glitch in Radar signal handling ($t = 30s$) does not affect the health of the other navigation components.

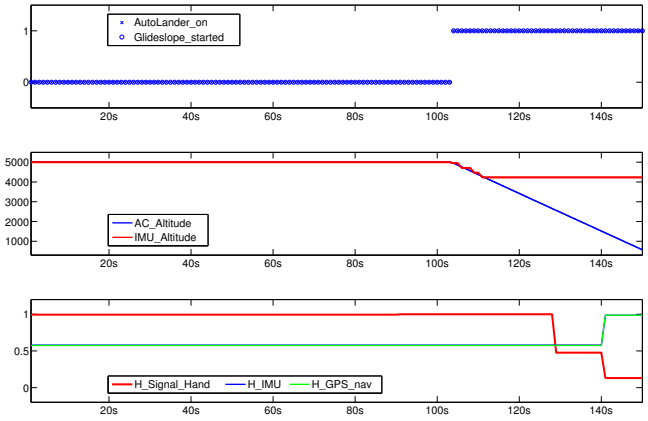

Fig. 13 Temporal trace for Signal Handling related fault causing erroneous IMU altitude estimation. Around time $t = 111s$ the IMU altitude estimate increasingly diverges from the plant actual altitude and erroneously indicates a constant altitude of about 4080ft. The belief in the health of the signal handling software remains high for approximately another 20 time units. Eventually, the SWHM inference engine reports a substantial drop in the belief in the health of the signal handling software after observation of further sensory evidence (bottom panel). CLose to $t = 140s$, further drop in this belief is indicated when excessive IMU and GPS navigations divergence is observed. But the beliefs in GPS and IMU health rise as the SWHM performs reasoning from all observed evidence so far.

It can be observed in the second panel of Figure 12 that in the nominal case when the Auto-Lander is engaged and the glideslope initiated from initial altitude, the IMU returns an altitude estimate that is mostly close to the actual plant altitude except for a some deviations accounted for by estimation error (due to the sensors' Gaussian error distribution, for instance). And the degrees of belief in the health of the signal handling software, the IMU, and GPS navigation remain relatively high.

We next study an off-nominal situation illustrated in Figure 13, when noise is injected into the radar altimeter's signals. For this experiment, the signal handling software is flawed and fails on the very noisy input at some point. The IMU altitude estimate starts to significantly diverge from the actual altitude as the radar altimeter reading is stuck at a previous altitude(4,080ft). The SWHM inference engine still reports a high degree of belief in the health of the signal handling software for about another 20 time units until further sensory evidence is observed, especially when the watchdog-timer raises a missed deadline flag. Given this additional sensory evidence, the SWHM inference engine reports a significant drop in the degree of belief in the health of

the signal handling software, as can be seen in the bottom panel of Figure 13. This belief drops even further upon observation of excessive divergence between actual IMU and GPS estimates and the expected values as provided by the IMU/GPS estimation. Meanwhile, the degrees of belief in GPS and IMU health rise. The SWHM system thus properly detects and diagnose this situation thus enabling the system to initiate mitigating or corrective actions.

4.2.3 Scenario III: GPS spoofing navigation fault

The use of Unmanned Aerial Systems (UASs) in critical operations has substantially increased across a broad range of applications. Recently, the 2012 National Defense Defense Authorization Act even provided for the establishment of pilot sites for integration of UASs into the national airspace.[11] UASs rely heavily on IMU and GPS navigation, and as their applications expand this reliance is likely to necessitate a higher degree of autonomous navigation and and decision making as well as intelligent real-time health monitoring and fault diagnostics. This includes monitoring for GPS spoofing-related navigation problems, a topic, which has recently gained much attention. Research on GPS spoofing by a Los Alamos National Laboratory team[12] and research by Tippenhauer et. al. pointed out specific spoofing techniques and related characteristics in affected navigation software [82]. A group at UT Austin has been able to demonstrate that the GPS receiver on a UAS can be deceived [78]. According to The Christian Science Monitor there is a strong likelihood that the U.S. RQ-170 Sentinel UAS was lost to Iran through GPS spoofing.[13]

Based on work of GPS navigation and spoofing, we have experimented with SWHM scenario whereby we monitor navigation software health, including the detection and diagnosis of failures or emergent behaviors in software caused by GPS navigation software faults or spoofing. Intermittent faults or events likely to occur in spoofing attacks, such as relative location jump, increased signal strength, GPS network formation change, and brief satellite lock loss, can be regarded as transient faults in navigation software leading to an erroneous state. Effective SWHM of a navigation system should therefore also address monitoring of GPS navigation in order to be able to detect and diagnose erroneous states caused by a spoofing attack.

[11] `http://defensesystems.com/articles/2011/12/22/ndaa-domestic-uas-test-sites.aspx`

[12] `http://www.homelandsecurity.org/bulletin/Dual\%20Benefit/warner_gps_spoofing.html`

[13] `http://www.csmonitor.com/layout/set/print/content/view/print/437272`

The SWHM model. The model for a navigation system software health affected by emergent behaviors or faults due to GPS spoofing is also depicted in Figure 11. The relevant software and hardware components and sensors conditional dependencies—including uncertainties—are effectively captured by the Bayesian network's nodes (on the right-hand side of Figure 11), edges, and conditional probability tables. Our SWHM model is designed to respond to situations including the following:

- Whether the estimated UAS position and altitude strays from the predicted values due to failure in the IMU or compromised GPS integrity.
- Whether transient altitude estimation behaviors (i.e. location jumps) are due to terrain irregularities or compromised GPS integrity.
- Whether the estimated position strays from the predicted position due to a failure in DSP signal handling software or compromised GPS navigation integrity.

The SWHM inference engine will provide real-time reasoning as to the integrity of the GPS and Inertial Measurement Unit navigation software for timely decision-making in critical situations. Integration of this SWHM reasoning capability with other monitoring systems can serve in real-time diagnosis disambiguation.

Experiments and Results. Temporal traces of an off-nominal experimental run is shown in Figure 14. Given sensory evidence the degrees of belief in the health of the GPS and IMU navigation software are mostly high, except for some terrain-related-transients before the simulated attack is injected. However, when the simulated attack is injected, some transient faults—such as excessive signal strength, location jump, formation change, and satellite lock loss—are eventually observed by the navigation software. A jump in the UAS location relative to desired position is briefly observed during that time interval, and the aircraft's estimated location is gradually and erroneously advanced toward the new destination—which will cause its belief state to be at waypoints close to the goal and initiate landing short of the actual goal. The middle panel of Figure 14 shows the perceived GPS position and the actual UAS position relative to the original goal. Upon receiving further subsystems' sensory evidence, the SWHM inference engine evaluates the belief in the integrity of the GPS and IMU navigation software as well as the signal handling software. At this point, SWHM system briefly reports a significant drop in the belief in GPS navigation integrity for about 5s, which slightly rises again while the degree of belief in the integrity of IMU navigation drops and subsequently plateaus.

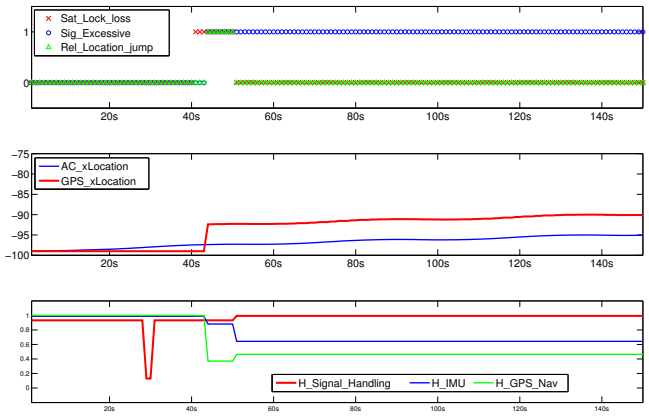

Fig. 14 Temporal trace for GPS spoofing scenario. The beliefs in the health of the GPS and IMU navigation software (green and blue respectively in the bottom panel) are high except for some terrain-related-transients before the simulated attack. When the attack is injected at $t = 40s$, transient excessive signal strength, location jump, and satellite lock loss are observed in the navigation software between $t = 40s$ and $t = 50s$ (top panel). Within that time interval, a jump in the plant location relative to satellites is observed(red in the middle panel). About 5 seconds later within this same time interval, a significant drop in the belief in GPS navigation integrity/health is reported between $t = 45s$ and $t = 50s$ (green in the bottom panel). This belief belief in GPS navigation integrity remains low and plateaus after about time $t = 52s$.

5 Towards V&V of SWHM

Obviously, we assume that the software to be monitored has undergone substantial verification and validation (V&V) before being deployed. After all, SWHM is not meant to *replace* traditional V&V but to provide an additional layer of safety. That said, what happens if an error does not show up in the application software but in the SWHM system, which itself is a piece of non-trivial software? According to "Quis custodiet ipsos custodes?"[14] we have to postulate that the SWHM must undergo V&V to at least the same level of rigor as the software that it is monitoring. It has to be made sure that an SWHM system never, under any circumstances, causes problems for the overall system. In particular, it must not corrupt any data in memory, cause timing overruns, crash the operating system, or send wrong data over a communications bus (e.g., a 1553 bus). Additionally, the SWHM should have a low rate of false alarms (false positives) and missed adverse events (false negatives). Although false alarms are not as dangerous as missed alarms, they should be avoided as much as possible to achieve optimal operation.

Unfortunately, traditional V&V as defined in many standards and processes (e.g., DO-178B) is not necessarily suitable and sufficient for our purposes, because our Bayesian SWHM approach is model-based and may involve algorithms not readily found in safety critical embedded systems. For our model-based approach, we developed new techniques toward V&V for SWHM, in particular for analysis and V&V on the model level (Bayesian networks) as well as on the code level (arithmetic circuits and circuit evaluators). A sketch of a possible V&V process is shown in Figure 15. The SWHM model is constructed based upon given requirements, background domain knowledge, and reliability data. For the ultimate goal of system integration, the SWHM model is first compiled into arithmetic circuits before it is incorporated into the SWHM implementation. This piece of code contains functionality for obtaining and preprocessing data from sensors and software monitors, as well as the reasoning executive. Finally, this software component is integrated into the overall system.

For V&V, a number of important analyses are carried out on the model level, including parametric model analysis (discussed below), (manual) model review, and testcase generation on the model level (see below for details). All errors detected during this V&V phase directly feed back (dashed lines) into SWHM requirements and the SWHM model (design). After compilation and integration, the SWHM is "just another piece of software," which, however, might incorporate non-standard algorithms. V&V on the code level (see Section 5.2) includes manual code review, static analysis, model checking, testing and test case generation for full code coverage, as well as worst-case execution time analysis (WCET) for the data-driven components of the SWHM implementation.

Even for different approaches to detection and diagnosis techniques (e.g., QSI TEAMS[15] or Livingston [44]), the V&V process will be similar to the one shown here, because in most approaches, the health model is translated or compiled into a highly compact and efficient data structure, which is then accessed by the health management engine to compute system health state.

5.1 Model-level V&V

An SWHM model captures essential information about nominal and off-nominal operation of the software and the host system on various levels of abstraction and is used by the SWHM engine to perform reasoning. Thus, V&V has to make sure that the model is *adequate* for

[14] Juvenal: "Who guards the guardians?"

[15] `http://www.teamqsi.com`

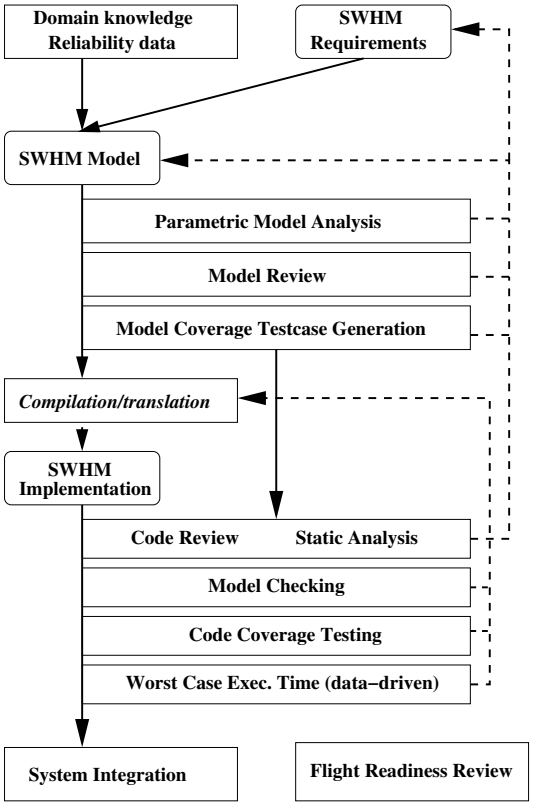

Fig. 15 V&V activities for model-based (including Bayesian network) SWHM that includes a compilation step (such as compilation of Bayesian networks to arithmetic circuits).

the given domain and SWHM requirements, and further that it is as *complete* and *consistent* as possible. State-of-the-art V&V approaches for IVHM systems include exhaustive model enumeration using a model checker, for example, Livingston Pathfinder [44].

Larger and hybrid models can be analyzed using parametric testing. This statistical approach combines n-factor combinatorial exploration with advanced data analysis [71] to exercise the SWHM model with wide ranges of sensor inputs and internal parameters. This approach scales to large systems, like fault detection models for the Ares I rocket [70] or hardware health management systems [63].

Lvl	Coverage	Tests trigger ...
1	Output	all states of each HEALTH node
2	Input	all states of each SENSOR/CMD node
3	Node	all states of each node
4	CPT	all entries of each CPT table

Table 4 Coverage metrics for Bayesian SWHM.

For the purpose of complete model coverage during model validation, we have defined SWHM-specific coverage metrics. Table 4 shows our metrics: full output coverage is obtained when there are test cases that exercise each health node in the model. In our example, we would need test cases to trigger all the nodes, like the `H_File_System` or `H_Msg_queue` nodes. A full coverage ensures that all possible diagnoses can actually be triggered. A more strict coverage is the input coverage: here, all possible values of all sensor and command nodes of the network must be exercised. This results in a full coverage of all possible behaviors of the SWHM, which can be obtained by stimuli from the outside.

Even with a fully covered input, the model still might contain inner status nodes, which are not yet fully covered. For example, the state of the `U_File_System` node might never be set to `full`, because of some error in the model (e.g., a missing edge from a sensor node). Our node coverage metric covers these nodes and makes it possible to expose such modeling errors.

Finally, the CPT metric looks into the individual nodes: for a full coverage, each entry in the CPT (conditional probability) table must be covered (i.e., used for posterior calculation) at least once. The aim of this metric is to detect fairly common modeling errors: uninitialized probabilities, i.e., no model-specific values were entered when the model was designed. In such cases, modeling tools like SamIam[16] will initialize the CPT table with uniform probabilities of $p = 1/N$, where N is the number of states associated with a node. This initialization yields a syntactically correct Bayesian network. Unfortunately, it may not correspond to the intended model. Other errors, like when a CPT has orders of magnitude too many CPT entries (e.g., 10^5 instead of 10×5), should also be detected, as they can indicate situations where model parameters (probabilities) may not have been specified.

For the generation of test cases for coverage metrics 1, 2, and 3, we have used tools based on the Java Pathfinder model checker and Symbolic Pathfinder [56]. For the CPT coverage metric, a specialized algorithm has been developed, which, taking the actual Bayesian network reasoning algorithm into account, generates a surprisingly small number of test cases for full CPT coverage [74].

Finally, sensitivity analysis of Bayesian Networks [18] is useful to assess the quality of the model parameters. Typical questions, which can be addressed with this analysis, are: "Do I still get the same diagnoses if the reliability for the sensor is off by an order of magnitude?" or "How does reasoning change if the file-system-full threshold is moved from 95% to 99%?" Certain Bayesian network modeling tools, such as SamIam, provide sensitivity analysis tools.

[16] `http://reasoning.cs.ucla.edu/samiam/`

5.2 Code-level V&V

Even after the SWHM model has been fully tested and verified on the model level, it can still cause severe problems. After compilation of the model into an Arithmetic Circuit (AC), the actual implementation of the SWHM evaluates the AC in order to obtain the proper diagnosis. As becomes evident from Figure 4, there is ample room for potential errors on the *code level* of SWHM. In particular, critical areas include the interprocess communication for obtaining the software and hardware sensor data, preprocessing and discretization, evaluation of the AC, and returning and processing of the SWHM results. Typical errors might include buffer overflows, memory leaks, arithmetic exceptions, roundoff errors, and deadlocks or race conditions (when accessing shared data).

Therefore, all pieces of the SWHM must undergo rigorous V&V. While there are many standards and tools for V&V of traditional safety critical code, the reasoning algorithm used to evaluate the arithmetic circuit is non-standard and thus needs to face heightened scrutiny.

Our minimalistic SWHM reasoning engines, which are the target of model translation and compilation, might even be amenable to formal verification. Because of the high complexity of the compilation process from BNs to ACs, however, it is hard to provide any guarantees about the compiler implementation. Also, the data-driven nature of the algorithm requires special care.

On the other hand, BN model compilation [18] eliminates the problems often associated with complex reasoning algorithms associated with BNs. The resulting algorithms and data structures can be formally shown to have limited resource bounds and do not require dynamic memory allocation. By construction, all data structures are static, there is no nondeterminism, and the execution times are bounded. Many V&V techniques that have been developed for traditional software can be used or extended easily. In particular, software model checking for the automatic proof of safety properties, static analysis, automatic generation of test cases and worst-case execution time analysis seem to be suitable approaches to demonstrate software safety and reliability and have been used for V&V of SWHM.

6 Conclusions

The ever-increasing reliance of aerospace on software for mission and safety critical operations, such as aircraft control and navigation, attests to the need for Software Health Management (SWHM) to diagnose and avert software-related faults in real-time. Bayesian networks are powerful in capturing subsystem interdependencies —as well as uncertainty factors—in order to perform real-time SWHM. A SWHM system dynamically monitors the target system (software, sensors, and hardware) and uses a health model in the form of a (compiled) Bayesian network to detect and reliably diagnose software-related faults in real-time. Bayesian networks are an ideal framework for modeling software health, because it allows complex reasoning with little computational overhead—an important prerequisite for onboard software health monitoring.

We have presented our approach and illustrated our Bayesian SWHM with a simplified aircraft control system. We have modeled several relevant failure scenarios, which were efficiently detected by the SWHM inference engine and which demonstrate the SWHM system's diagnostic reasoning capabilities.

Because a SWHM system, which monitors safety critical software, must be considered as a safety critical component as well, its V&V is paramount. In this paper, we have presented a two-stage V&V process, covering both the model level and the code level. Several specific technologies and approaches can be customized toward the specific requirements of SWHM V&V.

However, SWHM is still a young discipline and additional research should be carried out to mature our approach: automatic generation of SWHM models from requirements or code, as well hierarchical and modular modeling with Bayesian networks can substantially increase the scalability of our approach. The use of specific architectures, which provide built-in capabilities for SWHM integration (e.g., [23]), can facilitate instrumentation and implementation.

Despite all the potential advantages of SWHM to dynamically monitor the behavior of software in real-time, we have to note that SWHM cannot (and should not) lessen the burden of pre-deployment V&V and certification of the entire software. SWHM only provides an additional layer of protection during run-time, as it is capable of detecting and identifying faults that have not been found during traditional V&V or that are caused by unexpected environmental circumstances. SWHM does not intend to replace pre-deployment V&V.

Acknowledgements This work is supported by a NASA NRA grant NNX08AY50A "ISWHM: Tools and Techniques for Software and System Health Management". This paper is in part based upon prior conference papers [81, 73, 72], which discuss specific aspects of this work. We would also like to thank the anonymous reviewers for valuable feedback and comments.

References

1. Adler, M.: The Planetary Society Blog: Spirit Sol 18 Anomaly (2006). URL http://www.planetary.org/blog/article/00000702/
2. Anderson, T., Lee, P.A.: Fault Tolerance. Prentice-Hall International (1981)
3. Avižienis, A.: Fault-tolerant systems. IEEE Transactions on Computers **25**(12), 1304–1312 (1976)
4. Avižienis, A.: The N-version approach to fault-tolerant software. IEEE Transactions on Software Engineering **11**(12), 1491–1501 (1985)
5. Avižienis, A., Laprie, J.C., Randell, B., Landwehr, C.: Basic concepts and taxonomy of dependable and secure computing. IEEE Transactions on Dependable and Secure Computing **1**(1), 11–33 (2004)
6. Barringer, H., Falcone, Y., Finkbeiner, B., Havelund, K., Lee, I., Pace, G.J., Rosu, G., Sokolsky, O., Tillmann, N. (eds.): Runtime Verification - First International Conference (RV 2010). *Lecture Notes in Computer Science*, vol. 6418. Springer (2010)
7. Khurshid, S. and Sen, K. (eds.): Runtime Verification - Second International Conference, RV 2011. *Lecture Notes in Computer Science*, vol. 7186. Springer (2012)
8. Bernstein, L., Kintala, C.M.R.: Software rejuvenation. CrossTalk: The journal of defense software engineering pp. 23–26 (2004)
9. Binder, R.V.: Design for testability in object-oriented systems. Commun. ACM **37**(9), 87–101 (1994)
10. Bochmann, G., Dssouli, R., Zhao, J.: Trace analysis for conformance and arbitration testing. IEEE Transactions on Software Engineering **15**(11), 1347–1356 (1989)
11. Brown, D., Roggio, R., Cross J.H., I., McCreary, C.: An automated oracle for software testing. IEEE Transactions on Reliability **41**(2), 272–280 (1992)
12. Chavira, M., Darwiche, A.: Compiling Bayesian networks using variable elimination. In: Proceedings of the Twentieth International Joint Conference on Artificial Intelligence (IJCAI-07), pp. 2443–2449. (2007)
13. Chen, L., Avižienis, A.: N-version programming: A fault-tolerance approach to reliability of software operation. Twenty-Fifth International Symposium on Fault-Tolerant Computing, ' Highlights from Twenty-Five Years'. pp. 113– (1995)
14. Codetta-Raiteri, D., Portinale, L., Guiotto, A., Yushstein, Y.: 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) (2012)
15. Costa, M., Crowcroft, J., Castro, M., Rowstron, A., Zhou, L., Zhang, L., Barham, P.: Vigilante: End-to-end containment of Internet worms. In: Proceedings of the Symposium on Systems and Operating Systems Principles (SOSP), pp. 133–147 (2005)
16. Darwiche, A.: Recursive conditioning. Artificial Intelligence **126**(1-2), 5–41 (2001)
17. Darwiche, A.: A differential approach to inference in Bayesian networks. JACM **50**(3), 280–305 (2003)
18. Darwiche, A.: Modeling and Reasoning with Bayesian Networks. Cambridge University Press (2009)
19. Deconinck, G., Vounckx, J., Lauwereins, R., Peperstraete, J.A.: Survey of backward error recovery techniques for multicomputers based on checkpointing and rollback. International Journal of Modeling and Simulation **18**, 262–265 (1993)
20. Delgado, N., Gates, A.Q., Roach, S.: A taxonomy and catalog of runtime software-fault monitoring tools. IEEE Trans. Softw. Eng. **30**(12), 859–872 (2004)
21. Djurdjanovic, D., Liu, J., Marko, K.A., Ni, J.: Immune systems inspired approach to anomaly detection, fault localization and diagnosis in automotive engines. In: J. Schumann, Y. Liu (eds.) Applications of Neural Networks in High Assurance Systems, *Studies in Computational Intelligence*, vol. 268, pp. 141–163. Springer (2010)
22. Doong, R.K., Frankl, P.G.: The Astoot approach to testing object-oriented programs. ACM Trans. Softw. Eng. Methodol. **3**(2), 101–130 (1994)
23. Dubey, A., Karsai, G., Kereskenyi, R., Mahadevan, M.: A Real-Time Component Framework: Experience with CCM and ARINC-653. IEEE Intl. Symposium on Object-Oriented Real-Time Distributed Computing (2010)
24. Elnozahy, E.N.M., Alvisi, L., Wang, Y.M., Johnson, D.B.: A survey of rollback-recovery protocols in message-passing systems. ACM Comput. Surv. **34**(3), 375–408 (2002)
25. Firesmith, D.: Testing object-oriented software. In: Proceedings of the 11th International Conference on Technology of Object-Oriented Languages and Systems (TOOLS), pp. 407–426 (1993)
26. Forrest, S., Beauchemin, C.: Computer immunology. Immunological Reviews **216**(1), 176–197 (2007)
27. Gärtner, F.C.: Fundamentals of fault-tolerant distributed computing in asynchronous environments. ACM Comput. Surv. **31**(1), 1–26 (1999)
28. George, S., Evans, D., Marchette, S.: A biological programming model for self-healing. In: Proceedings of the 2003 ACM workshop on Survivable and self-regenerative systems (SSRS '03), pp. 72–81. ACM (2003)
29. Ghosh, D., Sharman, R., Rao, R.H., Upadhyaya, S.: Self-healing systems - survey and synthesis. Decis. Support Syst. **42**(4), 2164–2185 (2007)
30. Groce, A., Joshi, R.: Exploiting traces in static program analysis: better model checking through printfs. Int. J. Softw. Tools Technol. Transf. **10**(2), 131–144 (2008)
31. Hamou-Lhadj, A., Braun, E., Amyot, D., Lethbridge, T.: Recovering behavioral design models from execution traces. Software Maintenance and Reengineering, 2005. CSMR 2005. Ninth European Conference on pp. 112–121 (2005)
32. Harrold, M., McGregor, J., Fitzpatrick, K.: Incremental testing of object-oriented class structure. In: Proc. 14th International Conference of Software Engineering. pp. 68–80 (1992)
33. Hart, E., Timmis, J.: Application areas of AIS: The past, the present and the future. Applied Soft Computing **8**(1), 191 – 201 (2008)
34. Havelund, K., Rosu, G.: Efficient monitoring of safety properties. Int. J. Softw. Tools Technol. Transf. **6**(2), 158–173 (2004)
35. Hecht, H.: Fault-tolerant software for real-time applications. ACM Comput. Surv. **8**(4), 391–407 (1976)
36. Huang, Y., Kintala, C., Kolettis, N., Fulton, N.D.: Software rejuvenation: analysis, module and applications. Twenty-Fifth International Symposium on Fault-Tolerant Computing (FTCS-25), pp. 381–390 (1995)
37. Jensen, F.V., Lauritzen, S.L., Olesen, K.G.: Bayesian updating in causal probabilistic networks by local computations. SIAM Journal on Computing **4**, 269–282 (1990)
38. Johnson, D.: Raptors Arrive at Kadena (2007). URL http://www.af.mil/news/story.asp?storyID=123041567

39. Keromytis, A.: The case for self-healing software. In: Aspects of Network and Information Security: Proceedings NATO Advanced Studies Institute (ASI) on Network Security and Intrusion Detection (2007)
40. Keromytis, A.D.: Characterizing self-healing software systems. In: Proceedings of the 4th International Conference on Mathematical Methods, Models and Architectures for Computer Networks Security (MMM-ACNS) (2007)
41. Koo, R., Toueg, S.: Checkpointing and rollback-recovery for distributed systems. IEEE Transactions on Software Engineering **13**(1), 23–31 (1987)
42. Lauritzen, S., Spiegelhalter, D.J.: Local computations with probabilities on graphical structures and their application to expert systems (with discussion). Journal of the Royal Statistical Society series B **50**(2), 157–224 (1988)
43. Li, Z., D'Ambrosio, B.: Efficient inference in Bayes nets as a combinatorial optimization problem. International Journal of Approximate Reasoning **11**(1), 55–81 (1994)
44. Lindsey, A.E., Pecheur, C.: Simulation-based verification of autonomous controllers via Livingstone Pathfinder. In: Proc. 10th Intl. Conference on Tools and Algorithms for the Construction and Analysis of Systems (TACAS) *Lecture Notes in Computer Science*, vol. 2988, pp. 357–371. Springer (2004)
45. Lyu, M.R.: Software Fault Tolerance. John Wiley & Sons, Inc., New York, NY, USA (1995)
46. Mengshoel, O.J.: Designing resource-bounded reasoners using Bayesian networks: System health monitoring and diagnosis. In: Proceedings of the 18th International Workshop on Principles of Diagnosis (DX-07), pp. 330–337. Nashville, TN (2007)
47. Mengshoel, O.J., Chavira, M., Cascio, K., Poll, S., Darwiche, A., Uckun, S.: Probabilistic model-based diagnosis: An electrical power system case study. IEEE Trans. on Systems, Man, and Cybernetics **40**(5), 874–885 (2010)
48. Mengshoel, O.J., Darwiche, A., Uckun, S.: Sensor validation using Bayesian networks. In: Proceedings of the 9th International Symposium on Artificial Intelligence, Robotics, and Automation in Space (iSAIRAS-08) (2008)
49. Mengshoel, O.J., Roth, D., Wilkins, D.C.: Portfolios in stochastic local search: Efficiently computing most probable explanations in Bayesian networks. Journal of Automated Reasoning **46**(2), 103–160 (2011)
50. Mengshoel, O.J., Wilkins, D.C., Roth, D.: Initialization and restart in stochastic local search: Computing a most probable explanation in Bayesian networks. IEEE Transactions on Knowledge and Data Engineering **23**(2), 235–247 (2011)
51. Milea, N.A., Khoo, S.C., Lo, D., Pop, C.: Nort: Runtime anomaly-based monitoring of malicious behavior for windows. In: [7] (2011)
52. Musliner, D., Hendler, J., Agrawala, A.K., Durfee, E., Strosnider, J.K., Paul, C.J.: The challenges of real-time AI. IEEE Computer **28**, 58–66 (1995)
53. Neumann, P.: 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`
54. O'Malley, T.O., Richardson, D.J., Dillon, L.K.: Efficient specification-based oracles for critical systems. In: Proceedings of the California Software Symposium, pp. 50–59 (1996)
55. Park, J.D., Darwiche, A.: Complexity results and approximation strategies for MAP explanations. Journal of Artificial Intelligence Research (JAIR) **21**, 101–133 (2004)
56. Pasareanu, C.S., Rungta, N.: Symbolic pathfinder: symbolic execution of Java bytecode. In: Proc. Conference on Automated Software Engineering (ASE), pp. 179–180. ACM (2010)
57. Pearl, J.: Probabilistic Reasoning in Intelligent Systems: Networks of Plausible Inference. Morgan Kaufmann, San Mateo, CA (1988)
58. Pearl, J.: Causal diagrams for empirical research. Biometrika **82**(4):669–710 (1995)
59. Peters, D.K., Member, S., David, I., Parnas, L., Member, S.: Using test oracles generated from program documentation. IEEE Transactions on Software Engineering **24**, 161–173 (1998)
60. Pierce, W.H.: Failure-tolerant computer design. Academic Press (1965)
61. Poll, S., Patterson-Hine, A., Camisa, J., Garcia, D., Hall, D., Lee, C., Mengshoel, O.J., Neukom, C., Nishikawa, D., Ossenfort, J., Sweet, A., Yentus, S., Roychoudhury, I., Daigle, M., Biswas, G., Koutsoukos, X.: Advanced diagnostics and prognostics testbed. In: Proceedings of the 18th International Workshop on Principles of Diagnosis (DX-07), pp. 178–185. Nashville, TN (2007)
62. Randell, B.: System structure for software fault tolerance. In: Proceedings of the international Conference on Reliable Software, pp. 437–449. ACM, New York, NY, USA (1975)
63. Reed, E., Schumann, J., Mengshoel, O.J.: Verification and Validation of system health management models using parametric testing. In: Proc. Infotech@Aerospace (2011)
64. Richardson, D.J., Aha, S.L., O'Malley, T.O.: Specification-based test oracles for reactive systems. In: ICSE '92: Proceedings of the 14th international conference on Software engineering, pp. 105–118. ACM (1992)
65. Ricks, B.W., Mengshoel, O.J.: Methods for probabilistic fault diagnosis: An electrical power system case study. In: Proc. of Annual Conference of the PHM Society, 2009 (PHM-09) (2009)
66. RTCA: DO-178B: Software considerations in airborne systems and equipment certification (1992). URL `http://www.rtca.org`
67. Rushby, J.: Runtime certification. In: Proc. Runtime Verification (RV 2008), *Lecture Notes in Computer Science*, vol. 5289, pp. 21–35. Springer (2004)
68. SafeCode, L.: Qantas flight 72 accident caused by a software bug. URL `http://safecodellc.net/component/content/article/1-latest-news/112-qf-72-software-bug`
69. Schroeder, B.: On-line monitoring: a tutorial. Computer **28**(6), 72–78 (1995)
70. Schumann, J., Bajwa, A., Berg, P.: Parametric testing of launch vehicle FDDR models. In: AIAA Space (2010)
71. Schumann, J., Gundy-Burlet, K., Pasareanu, C., Menzies, T., Barrett, T.: Software V&V support by parametric analysis of large software simulation systems. In: Proc. IEEE Aerospace. IEEE Press (2009)
72. Schumann, J., Mbaya, T., Mengshoel, O.: Bayesian software health management for aircraft guidance, navigation, and control. In: Proc. of Conference on Prognostics and Health Management (PHM-2011) (2011)
73. Schumann, J., Mengshoel, O., MBaya, T.: Integrated software and sensor health management for small spacecraft. In: Proceedings of the 2011 IEEE Fourth International Conference on Space Mission Challenges for Information Technology, SMC-IT '11. IEEE (2011)

74. Schumann, J., Mengshoel, O.J., Pasareanu, C.S., Reed, E., Yang, G.: D1: Report on initial results of parametric analysis and prototype definition of model-based test case generation. Technical Report NASA/OSMA (SARP) (2010)
75. Schumann, J., Mengshoel, O.J., Srivastava, A.N., Darwiche, A.: Towards software health management with Bayesian networks. In: Proceedings of the FSE/SDP workshop on Future of Software Engineering Research, FoSER '10, pp. 331–336. ACM (2010)
76. Schumann, J., Morris, R., Mbaya, T., Mengshoel, O., Darwiche, A.: Report on Bayesian approach for dynamic monitoring of software quality and integration with advanced IVHM engine for ISWHM. Technical Report USRA-RIACS (2011)
77. Shenoy, P.P.: A valuation-based language for expert systems. International Journal of Approximate Reasoning **5**(3), 383–411 (1989)
78. Shepard, P, Bhatti, J.A., and Humphreys, T.E.: Drone Hack: Spoofing Attack Demonstration on a Civilian Unmanned Aerial Vehicle. GPS World (2012)
79. Smith, R., Korel, B.: Slicing event traces of large software systems. In: Proceedings of the Fourth International Workshop on Automated Debugging (AADEBUG). (2000)
80. Somayaji, A., Hofmeyr, S., Forrest, S.: Principles of a computer immune system. In: In Proceedings of the Second New Security Paradigms Workshop, pp. 75–82 (1997)
81. Srivastava, A.N., Schumann, J.: The case for software health management. In: Proceedings of the 2011 IEEE Fourth International Conference on Space Mission Challenges for Information Technology, SMC-IT '11, pp. 3–9. IEEE Computer Society, Washington, DC, USA (2011)
82. Tippenhauer, N.O., Popper, C., Rasmussen, K., Capkun, S.: On the requirements for successful GPS spoofing attacks. In: Proc. Chicago Communications Security Conference (2011)
83. Wang, J., Guo, C., Liu, F.: Self-healing based software architecture modeling and analysis through a case study. Networking, Sensing and Control, 2005. Proceedings. 2005 IEEE pp. 873–877 (2005)
84. Wang, Y., King, G., Court, I., Ross, M., Staples, G.: On testable object-oriented programming. SIGSOFT Softw. Eng. Notes **22**(4), 84–90 (1997)
85. Wang, Y., King, G., Wickburg, H.: A method for built-in tests in component-based software maintenance. Software Maintenance and Reengineering, 1999. Proceedings of the Third European Conference on pp. 186–189 (1999)
86. Wang, Y., Patel, D., King, G., Court, I., Staples, G., Ross, M., Fayad, M.: On built-in test reuse in object-oriented framework design. ACM Comput. Surv. pp. 7–12 (2000)
87. History's worst software bugs. Wired.com (2009)
88. Zhang, N.L., Poole, D.: Exploiting causal independence in Bayesian network inference. Journal of Artificial Intelligence Research **5**, 301–328 (1996)