Software and System Health Management for Autonomous Robotics Missions

Johann Schumann*, Timmy Mbaya**, Ole J. Mengshoel***

*SGT, Inc., NASA Ames Research Center, USA
e-mail: Johann.M.Schumann@nasa.gov

**University of Massachusetts, USA
e-mail: tmbaya@cs.umb.edu

***Carnegie Mellon University
NASA Ames Research Park, USA
e-mail: Ole.Mengshoel@sv.cmu.edu

Abstract

Advanced autonomous robotics space missions rely heavily on the flawless interaction of complex hardware, multiple sensors, and a mission-critical software system. This software system consists of an operating system, device drivers, controllers, and executives; recently highly complex AI-based autonomy software have also been introduced. Prior to launch, this software has to undergo rigorous verification and validation (V&V). Nevertheless, dormant software bugs, failing sensors, unexpected hardware-software interactions, and unanticipated environmental conditions—likely on a space exploration mission—can cause major software faults that can endanger the entire mission.

Our Integrated Software Health Management (ISWHM) system continuously monitors the hardware sensors *and the software* in real-time. The ISWHM uses Bayesian networks, compiled to arithmetic circuits, to model software and hardware interactions. Advanced reasoning algorithms using arithmetic circuits not only enable the ISWHM to handle large, hierarchical models that are necessary in the realm of complex autonomous systems, but also enable efficient execution on small embedded processors. The latter capability is of extreme importance for small (mobile) autonomous units with limited computational power and low telemetry bandwidth. In this paper, we discuss the requirements of ISWHM. As our initial demonstration platform, we use a primitive Lego rover. A Lego Mindstorms® microcontroller is used to implement a highly simplified autonomous rover driving system, running on the OSEK real-time operating system. We demonstrate that our ISWHM, running on this small embedded microcontroller, can perform fault detection as well as on-board reasoning for advanced diagnosis and root-cause detection in real time.

1 Introduction

The role of robotic explorers in space science missions is set to grow in this new space age. The launch of "the most technologically advanced"[1] exploration rover to date, the Mars Science Laboratory rover, *Curiosity*, boldly illustrate this fact.

The development of advanced robotics space missions is highly multidisciplinary and requires integration of various functional modules from diverse science and engineering teams. Systems integration and software development must proceed rapidly to meet project budget and schedule. Thus, robotic space mission systems consist of several complex subsystems (propulsion, avionics, communications, and science instruments), which must work together flawlessly. However, oftentimes budget constraints might limit the use of redundant hardware or software components, which would otherwise allow the system to detect failures (e.g., by a voting scheme) and to switch to a working component in the case of faults. Software to detect, isolate, and mitigate failures are highly primitive or missing altogether in most current small projects.

Clearly, a powerful FDIR (Fault Detection, Isolation, Recovery) or ISHM (Integrated System Health Management) system for hardware and software has great potential for ensuring the operational reliability of robotic space missions.[2] The role of software in robotic space missions is increasing as the mission complexities and advanced technology require more sophisticated software. Just as software has taken over more functionality in other systems such as aircraft, automobiles and other machinery, more and more functionality for autonomous space exploration systems is realized by using software rather than by dedicated (typically heavy or expensive) hardware

[1] http://www.jpl.nasa.gov/missions/details.cfm?id=5918

[2] Recovery or mitigation aspects, which are especially important for fully autonomous operations, are beyond the scope of this paper.

components. The Mars Science Laboratory Curiosity rover, for instance, is a good illustration of how mission complexity and advanced technology have spurred even more software functionality and autonomy. The critical EDL (Entry, Descent and Landing) scheme—consisting of a combination of parachute, thrusters, "sky crane", and radar—heavily relies on advanced complex software adding up to at least 500,000 lines of code.[3]

The amount and complexity of embedded software running on robotic spacecraft has increased tremendously over the last few years. With budget limitations for development as well as for verification and validation (V&V), problems are unfortunately very likely. In this paper, we focus on robotic space mission system health management, including both software and sensor health management.

A number of missions have failed due to software errors or problematic software-hardware interactions. Although ISWHM will not be able to detect all such problems, the reliability of a software system can be enhanced with a system that can detect and diagnose software failures. The following examples illustrate how simple software problems can jeopardize expensive missions.

In 1999, the Mars Polar Lander (MPL) crashed onto the surface of Mars. An incident investigation concluded that "the most probable cause of the failure was the generation of spurious signals when the lander legs were deployed during descent. The spurious signals gave a false indication that the spacecraft had landed, resulting in a premature shutdown of the engines[...]" [13]. It seems that the software was ill-equipped to deal with such spurious signals. An on-board ISWHM could have taken additional available information into account (e.g., from the radar altimeter) and potentially have mitigated the engine shutdown and consequently the crash.

With the extremely high complexity of mission software for the Mars rover Curiosity, ISWHM is even more critical. Indeed, due to radio communication latency between Earth and Mars, the rover Curiosity will be entirely autonomous during the whole 7 minutes of EDL, the so-called "7 minutes of terror" that can break or make this mission. Evidently, it is of utmost importance to efficiently monitor software and hardware interactions to ensure successful autonomous landing and avert a disastrous crash like the MPL's.

The Mars rover SPIRIT also experienced a software problem. A short time after landing, SPIRIT encountered repeated reboots, because a software fault during the booting process caused an immediate reboot. According to reports [1], an on-board file system for intermediate data storage caused the problem. After this storage was filled up, the boot process failed while trying to access that file system. The problem was detected on the ground and solved successfully. This example shows how, despite careful V&V, hard to detect errors can still remain dormant in the software. This example also shows that certain kinds of software failures could be detected by the ISWHM system by monitoring and reasoning before the actual faults occur.

A powerful, on-board Software and Sensor Health Management (ISWHM) has the potential to detect many possible faults as soon as they occur and can thus substantially contribute to overall mission safety and reliability. Such an ISWHM

- *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. Most of this information is readily available without the need to instrument or otherwise modify the software.

- *performs substantial diagnostic reasoning in order to identify the most likely root cause(s) for the fault and a quality measure for that result.* This diagnostic capability is extremely important, because telemetry bandwidth of a small autonomous system (e.g., a rover) is very limited. So, only a small subset of the system information can be down-linked, which makes accurate ground-based failure diagnosis impossible.

Simple diagnostic routines, as often used with current systems, typically process symptoms in isolation, which can result in incorrect diagnoses or a large number of contradictory results. This problem became evident in a recent A380 incident,[4] in which the pilots reported more than 400 diagnostic messages, some of which contradicted each other.

Finally, a proper probabilistic treatment of the diagnosis process, as we accomplish with our Bayesian approach [8, 3], can not only merge information from multiple sources but also provide a posterior distribution for the diagnosis. We note that this approach has been very successful for electrical power system diagnosis [10, 11, 6].

Such an ISWHM system must, to be useful for an autonomous system, satisfy several important properties, including the following.

(1) *Have a small memory and resource footprint.* Here, resources do not only mean low computational requirements, but the may ISWHM also need to "compress" the system and software status into few, relevant, and reliable pieces of health information, which can be transmitted using a low telemetry bandwidth. In such cases, the ISWHM

[3] http://news.sciencemag.org/sciencenow/2012/06/scienceshot-seven-minutes-of-ter.html

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

system performs an intelligent compression of raw sensor data into system health information.

(2) *Be able to fuse information from hardware sensors and software monitors.* For an ISWHM system, it is not sufficient to just monitor software and hardware separately, because many software problems occur due to problematic software-hardware interactions. For example, the Ariane 5 (software) error [14] was triggered because the hardware sensor produced sensor readings with a range of values that was larger than acceptable by the software, which had been constructed for the much smaller Ariane 4. In our ISWHM approach, we use Bayesian networks to model SW and HW, as well as their interaction, in a probabilistically principled way.

(3) *Exhibit a low number of false positives and false negatives.* False alarms (false positives) can produce nuisance signals; missed adverse events (false negatives) can endanger mission success. We ensure a high-quality ISWHM by taking a BN-based approach, where there is a clear mapping from the structure of the rover hardware and software to the structure of the BN.

(4) *Should not require instrumentation or other changes to the software.* Most of the information required by the ISWHM is already available within the software system and can be read from global storage or buses. Thus a powerful ISWHM can be developed, which does not interfere with other software. Where necessary and suitable, sensors implemented in software can provide additional information to the ISWHM.

(5) *Be V&V-able.* Any software that is running on-board a complex system must undergo rigorous V&V. Our approach of using ISWHM models, in the form of BNs that have been compiled into arithmetic circuits, is amenable to V&V [9].

(6) *Be constructed in a modular, hierarchical, and reusable way.* In order to support modular construction of an autonomous system, the health management modeling paradigm also must support modular, hierarchical, and reusable models.

Once a fault has been detected and its causes diagnosed properly, a multitude of different autonomous or ground based mitigation strategies could be applied to deal with that problem. However, successful mitigation requires a reliable detection and diagnosis of the software faults. Thus, in this paper, we focus on the detection and diagnosis part of ISWHM.

The rest of the paper is structured as follows: In Section 2 we present background on Bayesian modeling and reasoning as well as the compilation of Bayesian networks into efficient arithmetic circuit data structures. Section 3 discusses how our sensor and software health management model is constructed. We demonstrate our approach using a small and simple rover (Section 4). We describe the system and software architecture, which is built upon the OSEK operating system, discuss the ISWHM model, and present a number of nominal and off-nominal scenarios, which are executed on the rover. Finally, in Section 5 we conclude.

2 Bayesian Software and Sensor Health Management

The goal of our Bayesian ISWHM is to detect and isolate bugs that may be software-only or involve both software and hardware. In our approach, we use the well-established technology of Bayesian networks (BNs).

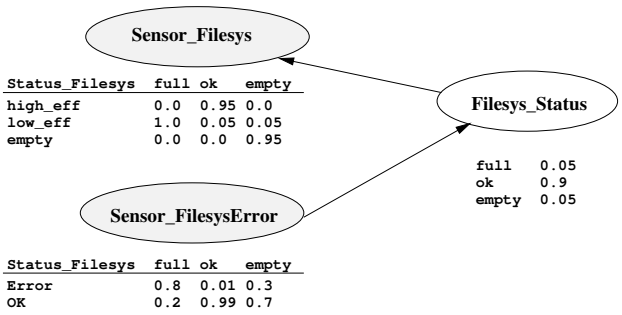

Figure 1. Simple Bayesian network for system health management. Conditional probability tables (CPTs) are shown near each node.

Figure 1 shows a very simple BN; in fact it is a small fragment of the network used for experimentation in this paper. We have a node `Filesys Status` (FS) representing the status of a file system in a rover, a sensor node `Sensor Error` (SE) representing whether a file system error is reported or not, and a node `Sensor Filesys` (SF) representing file system utilization. Clearly, the sensor readings (nodes SE and SF) depend on the status of the file system, 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 a problem, the probability that $p(FS \neq \text{ok})$ increases. To obtain the status of the file system, we input (or clamp) the states of the BN sensor nodes and compute the posterior distribution (or belief) over FS. The prior distribution of failure, as reflected in the CPT shown next to FS, is also taken into account in this calculation.

Our example network in Figure 1 represents the joint probability $p(FS, SE, SF)$ and is shown in Table 1. For simplicity, we replace all CPT entries with θ_x (i.e., $\theta_{ok} \leftrightarrow$ "FS is ok," and $\theta_f \leftrightarrow$ "FS is full"). Here, we also use indicators λ_x to incorporate evidence when the state of

a variable is observed [3], where $\lambda_x = 1$ if x is consistent with our observations and $\lambda_x = 0$ if x is inconsistent with our observations.[5] Part of the probability distribution $p(FS, SE, SF)$ represented by the Bayesian network is shown in Table 1.

FS	SE	SF	$p(FS, SE, SF)$
ok	er	he	$\lambda_{ok}\lambda_{er}\lambda_{he}\theta_{er\|ok}\theta_{ok}\theta_{he\|ok}$
ok	er	le	$\lambda_{ok}\lambda_{er}\lambda_{le}\theta_{er\|ok}\theta_{ok}\theta_{le\|ok}$
ok	er	e	$\lambda_{ok}\lambda_{er}\lambda_{e}\theta_{er\|ok}\theta_{ok}\theta_{e\|ok}$
ok	ok	he	$\lambda_{ok}\lambda_{ok}\lambda_{he}\theta_{ok\|ok}\theta_{ok}\theta_{he\|ok}$
ok	ok	le	$\lambda_{ok}\lambda_{ok}\lambda_{le}\theta_{ok\|ok}\theta_{ok}\theta_{le\|ok}$
ok	ok	e	$\lambda_{ok}\lambda_{ok}\lambda_{e}\theta_{ok\|ok}\theta_{ok}\theta_{e\|ok}$
f	er	he	$\lambda_{f}\lambda_{er}\lambda_{he}\theta_{er\|f}\theta_{f}\theta_{he\|f}$
f	er	le	$\lambda_{f}\lambda_{er}\lambda_{le}\theta_{er\|f}\theta_{f}\theta_{le\|f}$
f	er	e	$\lambda_{f}\lambda_{er}\lambda_{e}\theta_{er\|f}\theta_{f}\theta_{e\|f}$
f	ok	he	$\lambda_{f}\lambda_{ok}\lambda_{he}\theta_{ok\|f}\theta_{f}\theta_{he\|f}$
f	ok	le	$\lambda_{f}\lambda_{ok}\lambda_{le}\theta_{ok\|f}\theta_{f}\theta_{le\|f}$
f	ok	e	$\lambda_{f}\lambda_{ok}\lambda_{e}\theta_{ok\|f}\theta_{f}\theta_{e\|f}$
e	er	he	$\lambda_{e}\lambda_{er}\lambda_{he}\theta_{er\|e}\theta_{e}\theta_{he\|e}$
...	...	...	...

Table 1. Probability distribution for $p(FS, SE, SF)$.

According to this joint probability distribution table, the first row ($\lambda_{ok}\lambda_{er}\lambda_{he}\theta_{er|ok}\theta_{ok}\theta_{he|ok}$) is representing the probability that the status of the file system is okay (FS = ok and $\lambda_{ok} = 1$) and that file system error and high efficiency are observed (SE = er, SF = he, $\lambda_{er} = 1$, and $\lambda_{he} = 1$). Given the corresponding numerical CPT entries, the probability can be calculated as indicated in Table 1.

In general, the rows of this table define a joint distribution:

$$p(FS, SE, SF) = \prod_{\lambda_x} \lambda_x \prod_{\theta_{x|\mathbf{u}}} \theta_{x|\mathbf{u}},$$

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. Posterior marginals representing the health state of the system can be computed from the joint distribution.

To avoid the inefficiencies associated with the full distribution shown in Table 1, BNs are often compiled, off-line, to secondary data structures such as clique trees or arithmetic circuits [3]. These secondary data structures are then used for on-line computation. For resource-restricted and embedded settings, this compilation approach is especially suitable [5]. During on-line computation, the arithmetic circuit-based diagnostic software reads commands and sensor data, performs preprocessing including discretization, inputs evidence into the arithmetic circuit, evaluates it, and outputs diagnostic results based on the posterior distribution over the health random variables [10, 11, 6].

3 Constructing the Model

As discussed above, a BN has several kinds of different nodes and has certain directed arcs between nodes. In the following, we discuss in more detail how the Bayesian networks for our ISWHM models are constructed.

Nodes. Our Bayesian ISWHM 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.

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. For our reasoning it is assumed that this action is in fact happening.[6] The CMD nodes are root nodes (no incoming edges). During the execution of the ISWHM, these nodes are always directly connected (clamped) by 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.

HEALTH node H: The health nodes are nodes that reflect the health status of a sensor or component. Their posterior probabilities comprise the output of an ISWHM 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.

STATUS node U: A status node reflects the (unobservable) status of the software component or subsystem.

[5] For example, if we observe an error on sensor SE, then $\lambda_{er} = 1$ and $\lambda_{ok} = 0$; if we observe no vibration on sensor SE, then $\lambda_{ok} = 1$ and $\lambda_{er} = 0$. If there is no observation for the sensor SE, we leave $\lambda_{ok} = 1$ and $\lambda_{er} = 1$.

[6] 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 as further discussed below. Alternatively, one might consider using a sensor node instead.

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 a behavioral node.

The following informal way to think about *edges* in Bayesian networks are useful for knowledge engineering purposes: An edge (arrow) from node C to node E indicates that the state of C has a (causal) influence on the state of E. Generally, the types of influences typically seen in ISWHM BNs are as follows:

$\{H, C\} \rightarrow U$ represents how status node 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.

$\{H, U\} \rightarrow S$ represent 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).

$\{U\} \rightarrow S$ represents how system status U influences a sensor S, as demonstrated in Figure 1. Sensor noise and failure can both be rolled into the CPT of S.

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 [10, 6]. 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-to-failure 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.

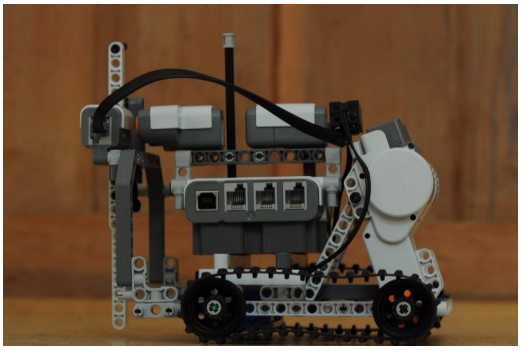

Figure 2. Small Lego rover demonstration platform.

We note that traditional diagnostics ("flight rules") can easily be integrated into 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 ISWHM BN, Bayesian network can be exploited for sensor validation as well as for diagnosis [7].

4 Experiments and Results

4.1 Demonstration System

In order to conduct realistic experiments, we used a simple autonomous hardware platform. The aim of this study is to demonstrate that our approach to ISWHM can be executed on a small embedded system in real-time. A small rover, built with Lego® and Lego NXT® served as the hardware platform. Figure 2 shows a photo of this simple platform. Each of the treads are controlled by a servo motor (DC motor with attached wheel-rotation sensor). Additional sensors, like another wheel rotation sensor for each axis, a 3-axes accelerometer, an ultrasound distance measuring sensor (in lieu of a RADAR instrument), and a (on purpose poorly designed) touch sensor complements this hardware system. Figure 3 shows a schematic of the sensors and actuators. The NXT uses a 32-bit ARM7 processor with 64kB of RAM and 256kB of flash[7]. We have implemented a simple software version of sensor and rover control in C and use the OSEK real-time operating system[8] as the underlying operating system. A NXT-specific version[9] provides a tool chain and drivers for that platform. While the OSEK RTOS is most widely

[7] http://en.wikipedia.org/wiki/Lego_Mindstorms_NXT
[8] http://www.osek-vdx.org/
[9] http://lejos-osek.sourceforge.net/

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[10] and GreenHills' INTEGRITY.[11] OSEK's basic functionalities and availability were sufficient for the purpose of our experiments [12] and show how ISWHM can be used even in very restrictive embedded systems. In addition, Lego and its microcontroller have already been used for robotics research [4, 2].

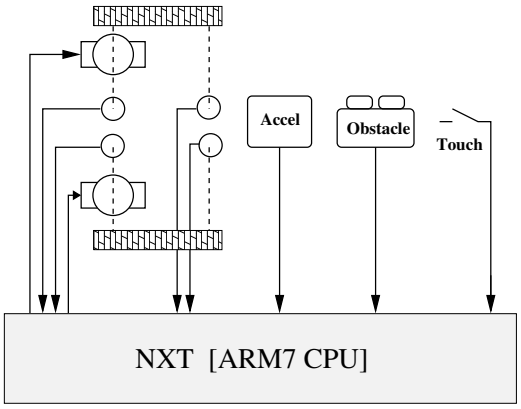

Figure 3. Sensor schematics for our rover hardware platform.

4.2 Software Architecture

The entire control software and the ISWHM is running as a set of OSEK Tasks on the NXT processor. All tasks used predefined task priorities, stack sizes, and rates, which range from a fast 10ms sensor acquisition task to the ISWHM, and data logging tasks, which are only executed every 200*ms*, and 500*ms*, respectively. Drivers for sensors and motors were used as provided in NXT-OSEK.[12] Figure 4 shows the software architecture; OSEK resources were used to govern access to global rover status variables. Despite the simple structure, this software architecture shows all typical ingredients needed for the implementation of GN&C and autonomy software on a small embedded system. For our experiments we assume that the entire "software load" (except for the ISWHM task) is given. Injected errors and on-purpose software design flaws are used to drive the scenarios as discussed below.

The control software consists of a fast sensor acquisition task and a control task. Sensor data and actuator data are transmitted using a message queue with a fixed length. For some scenarios, the task handling the message queue

[10] http://www.windriver.com
[11] http://www.ghs.com
[12] http://lejos-osek.sourceforge.net/

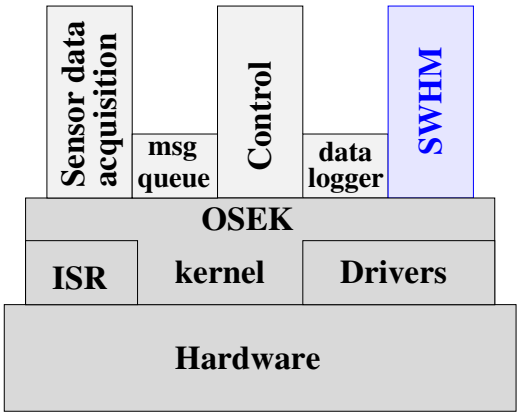

Figure 4. Software architecture: The ISWHM task is highlighted as it is the only addition to the given software load.

can log each message into an on-board file store.

Listing 1 Core functionality of the ISWHM task (resource protection is not shown.)

```
TASK(TaskISWHM_500ms) {
    iswhm_preprocess_data();
    iswhm_set_evidence();
    iswhm_reason();
    iswhm_results();
    TerminateTask();
    }

 void iswhm_set_evidence(){
   observe(varIndex("Sensor_FileSystem"),
           rover_psensors.sensor_FS);
   ...
   }
 void iswhm_reason(){
   evaluate();
   differentiate(false);
   evaluationResults();
   }
 void iswhm_results(){
   post = getPosteriors();
   rover_iswhm.health_FS=post[7];
   }
```

4.3 ISWHM

The ISWHM task (Listing 1) runs on a low priority and only interacts with the rest of the software system by reading the sensor values and returning the posterior probabilities of the health nodes for logging. More specifically, this task preprocesses the sensor values, and provides them as evidence (`observe()`) to the Arithmetic Circuit. Then reasoning (`reason()`) is carried out, evaluating and differentiating the Arithmetic Circuit to extract

Name	Type	States
Cmd_fwd	CMD	idle, on
Cmd_backwd	CMD	idle, on
Sensor_touch	SENSOR	idle, active
Sensor_obstacle	SENSOR	invalid, near, far
Sensor_acc[3]	SENSOR	negative, zero, positive
Sensor_spd[2]	SENSOR	negative, zero, positive
Sensor_motorspd[2]	SENSOR	negative, zero, positive
Sensor_batt	SENSOR	ok, load, low
SW_FS	SENSOR	ok, filling, full
SW_stack	SENSOR	ok, ovflow
SW_time	SENSOR	ok, deadline
SW_res	SENSOR	ok, deadlock
Health_L_motor	HEALTH	ok, low_eff, bad
Health_R_motor	HEALTH	ok, low_eff, bad
Health_obstacle	HEALTH	ok, fault
Health_touch	HEALTH	ok, fault_open, fault_closed
Health_acc[3]	HEALTH	ok, fault
Health_SW	HEALTH	ok, ctrl_fault

Table 2. Command, sensor (hardware and software) signals, and health nodes for ISWHM (selection).

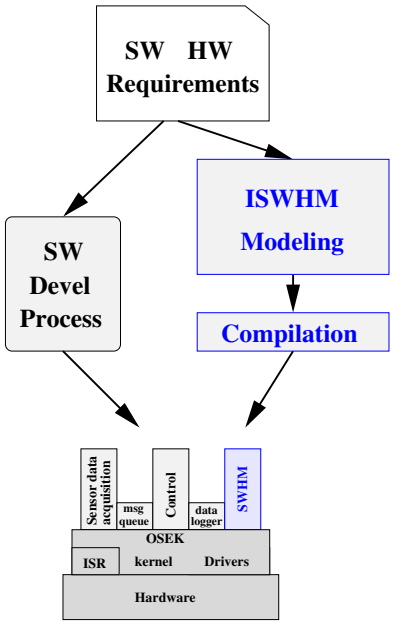

Figure 5. Integration of ISWHM.

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` (not shown here).

Table 2 shows selected sensor nodes for hardware and software sensors, their type and possible states. In general, additional preprocessing steps might take place to derive the appropriate data from the stream of sensor signals (e.g., discretization, moving average, minimum, maximum).

The overall development process and tool chain is shown in Figure 5. Independently of the actual rover software development, the ISWHM Bayesian model is developed (based upon requirements and SW/HW characteristics) using the tool SamIam.[13] The Bayesian network then is compiled into an arithmetic circuit, which serves as the ISWHM model (the knowledge base) using UCLA's ACE[14] Arithmetic Circuit compiler package. The resulting C data structure is integrated with ISWHM inference engine and the rest of the software system. The Bayesian network model is compiled "offline"—only once—into an Arithmetic Circuit serving as the knowledge base—amortizing compilation overheads with real-time execution in the running system. Figure 6 shows a graphical representation of the BN. On the given architecture, a full reasoning cycle using the compiled arithmetic circuit could be accomplished in approximately $15ms$. Most of that time is spent on floating-point additions and multiplications. It is expected that with the use of fixed-point arithmetic, this time can be reduced substantially.

[13] `http://reasoning.cs.ucla.edu/samiam/`
[14] `http://reasoning.cs.ucla.edu/ace/`

4.4 Analyzed Scenarios

During each experiment, the status of the rover, sensor and command signals as well as the health information of the ISWHM task are transferred to an external PC using Bluetooth.

In the nominal case for this Lego rover ISWHM demonstrator, the inference engine reports a high degree of belief in the health of the software and hardware systems. After idling, the rover executes a drive-forward command toward its target and stops after a few seconds. It is driving toward an obstacle at a larger distance. The health of the motors, wheel, battery, touch and obstacle sensors, as well as the posterior probabilities of the health of the software and control systems are consistently high except for transient lows—i.e., during idle time. All probabilities are well above 0.5, indicating nominal (no-fault) operation.

However, the ISWHM inference engine indicates a low degree of belief in the health of the software and control systems in the the following failure scenario—which is loosely modeled after the Mars Polar Lander incident discussed above. A mechanical touch sensor is supposed to stop both motors whenever the rover hits an obstacle. Figure 7 shows a temporal trace. The rover starts moving forward around $t = 4500ms$. Before then, as the system is idle, the posterior probability of the health of the software is "neutral," 50/50, (red in bottom panel). The obstacle sensor indicates that the rover is gradually approaching the

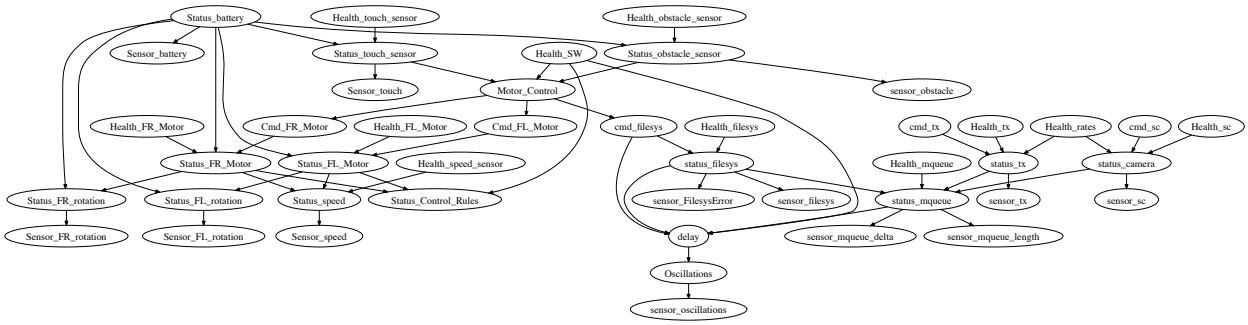

Figure 6. Simplified ISWHM model for the rover demonstrator.

obstacle though it is still distant (second panel from top). The battery voltage sensor indicates an expected drop at start-up around $t = 4700ms$ as the load is applied (pink in top panel), while wheel and speed sensors indicate activation (black and green respectively in third panel from top). And the ISWHM inference engine infers a high degree of belief in the health of the software system (high posterior distribution) as the rover operates smoothly from start-up to about $t = 6000ms$ (red in bottom panel). A small rock under one of the treads triggers a short (and spurious) signal of the touch sensor around $t = 6000ms$. The poorly designed software interprets this signal as the rover hitting the obstacle, and thus shuts down the motors. The battery voltage sensor subsequently indicates an increase around $t = 6500ms$ as the wheels stop moving (pink in top panel and black in third panel from top respectively). However, the obstacle sensor signals that the obstacle was not even close (second panel from top). This scenario, though not nearly as dangerous for a driving rover as for a landing probe, illustrates how faulty signal processing and software can wreak havoc. In the bottom panel of Figure 7, the ISWHM inference engine clearly indicates that the signal processing and control software are to blame, as the posterior distribution (degree of belief) of the software health dips (red in bottom panel). The ISWHM inference engine uses additional information from the obstacle, speed, and motor sensors for reasoning and disambiguation. In addition, the actual length of the signal could be used to further strengthen the ISWHM reasoning.

A number of pre-analyzed scenarios were executed in our experiments to validate the ISWHM model. Many of these experimental scenarios were inspired by actual incidents. Automated reasoning with our ISWHM approach can perform real-time diagnosis and disambiguation in cases such as these experimental software and hardware related fault scenarios:

- right tread not operational: motor blocked
- right tread not operational: tread "stuck" in sand (see Mars rover SPIRIT)
- accelerometer fault
- faulty software-induced violation of control laws (overlapping dead-bands)
- a software fault in the on-board data storage system and signal delay induces oscillations

5 Conclusions

In this paper, we have presented a powerful approach for the monitoring of software and sensors of an autonomous system, while it is in operation. Our ISWHM approach uses advanced Bayesian methods to decide if the system status is nominal or if any problems occur. In the latter case, the posterior probability of the Bayesian health nodes provide information on which component(s) are most likely faulty. By combining health management of sensors with dynamic monitoring of the on-board software, we are able to detect and identify a multitude of software errors, which may have gone undetected during the V&V phase for the autonomous software, or which arise from unexpected hardware-software interactions.

The Bayesian methods underlying the ISWHM provide a powerful modeling mechanism, which can incorporate information from hardware sensors, the operating system, and available status signals from selected software components. Local conditional probability values help to disambiguate the diagnosis.

The Bayesian health model is compiled into an arithmetic circuit (AC), which can be executed very efficiently even in severely constrained real-time environments. We demonstrate our approach with a small rover-like vehicle built out of Lego, which is controlled by software running under the OSEK realtime operating system on the Lego NXT (ARM7) processor. Our compiled ISWHM model is running as a separate low-priority task and can provide onboard diagnostic results. Our approach does not require any modifications to the autonomy software itself.

Since size and complexity of software for even tiny autonomous systems increase dramatically, we think that

powerful on-board means for real-time fault detection and diagnosis can provide a crucial additional layer of reliability.

Acknowledgments. This work was supported by a NASA NRA grant NNX08AY50A "ISWHM: Tools and Techniques for Software and System Health Management". The Lego rover was designed and built by Martin L. Schumann.

References

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

[2] A. Cruz-Martin, J. A. Fernández-Madrigal, Cipriano Galindo, J. González-Jiménez, C. Stockmans-Daou, and J. L. Blanco-Claraco. A Lego Mindstorms NXT approach for teaching at data acquisition, control systems engineering and real-time systems undergraduate courses. *Computers & Education*, 59(3):974–988, 2012.

[3] A. Darwiche. *Modeling and Reasoning with Bayesian Networks*. Cambridge University Press, Cambridge, UK, 2009.

[4] Lloyd Greenwald, Donovan Artz, Yogi Mehta, and Babak Shirmohammadi. Using educational robotics to motivate complete AI solutions. *AI Magazine*, 27(1):83–95, 2006.

[5] O. J. Mengshoel. 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)*, pages 330–337, Nashville, TN, 2007.

[6] O. J. Mengshoel, M. Chavira, K. Cascio, S. Poll, A. Darwiche, and S. Uckun. Probabilistic model-based diagnosis: An electrical power system case study. IEEE Trans. on Systems, Man, and Cybernetics 40(5):874–885, 2010.

[7] O. J. Mengshoel, A. Darwiche, and S. Uckun. Sensor validation using Bayesian networks. In *Proceedings of the 9th International Symposium on Artificial Intelligence, Robotics, and Automation in Space (iSAIRAS-08)*, 2008.

[8] J. Pearl. *Probabilistic Reasoning in Intelligent Systems: Networks of Plausible Inference*. Morgan Kaufmann, San Mateo, CA, 1988.

[9] E. Reed, J. Schumann, and O. J. Mengshoel. Verification and validation of system health management models using parametric testing. In *Proc. Infotech@Aerospace*, 2011.

[10] B. W. Ricks and O. J. Mengshoel. Methods for probabilistic fault diagnosis: An electrical power system case study. In *Proc. of Annual Conference of the PHM Society, 2009 (PHM-09)*, San Diego, CA, 2009.

[11] B. W. Ricks and O. J. Mengshoel. Diagnosing intermittent and persistent faults using static Bayesian networks. In *Proc. of the 21st International Workshop on Principles of Diagnosis (DX-10)*, Portland, OR, 2010.

[12] J. Schumann, R. Morris, T. Mbaya, O. Mengshoel, and A. Darwiche. Report on Bayesian approach for dynamic monitoring of software quality and integration with advanced IVHM engine for ISWHM. USRA-RIACS, 2011.

[13] P. Wilhide. Mars Program Assessment Report Outlines Route to Success, `http://mars.jpl.nasa.gov/msp98/news/news71.html`, 2000.

[14] History's worst software bugs. Wired.com, 2009.

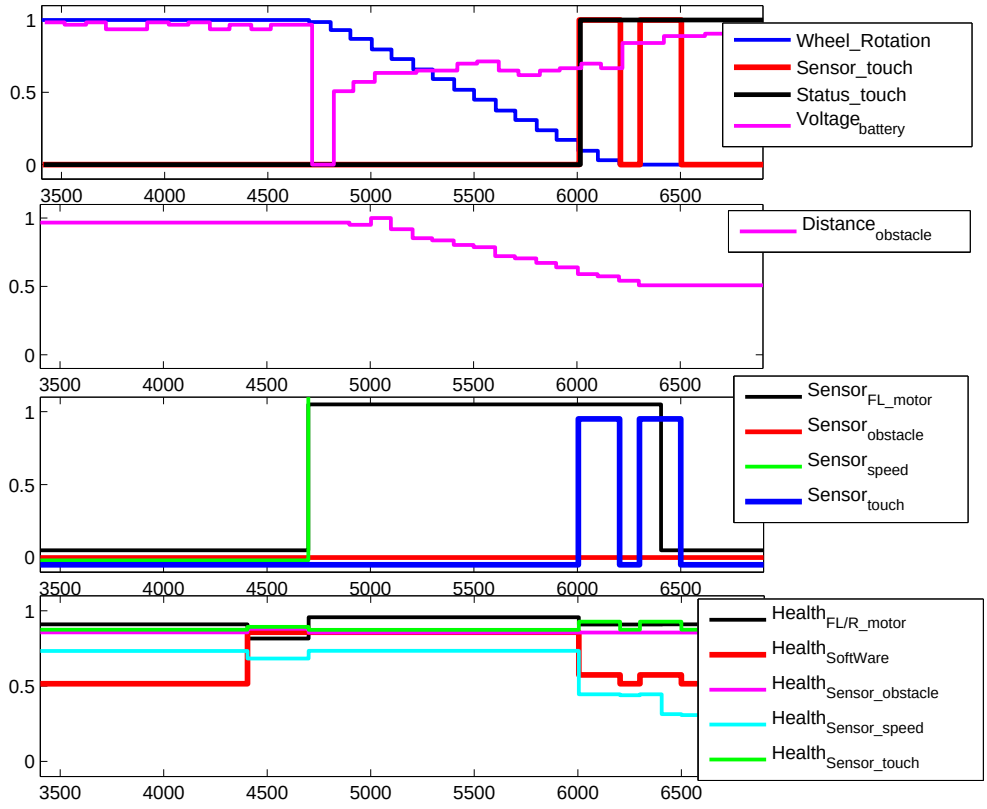

Figure 7. Temporal traces for failure scenario: spurious signals. The ISWHM inference engine indicates a low degree of belief in the health of the software despite signals from the touch sensor.