Towards Real-time, On-board, Hardware-supported Sensor and Software Health Management for Unmanned Aerial Systems

Johann Schumann[1], Kristin Y. Rozier[2], Thomas Reinbacher[3], Ole J. Mengshoel[4], Timmy Mbaya[5], and Corey Ippolito[6]

[1] *SGT, Inc., NASA Ames Research Center, Moffett Field, CA 94035, USA*
johann.m.schumann@nasa.gov

[2,6] *NASA Ames Research Center, Moffett Field, CA 94035, USA*
Kristin.Y.Rozier@nasa.gov
corey.ippolito@nasa.gov

[3] *Vienna University of Technology, Treitlstrasse 3, 1040 Wien, Austria*
treinbacher@ecs.tuwien.ac.at

[4] *Carnegie Mellon University, Moffett Field, CA 94035, USA*
ole.mengshoel@sv.cmu.edu

[5] *University of Southern California, Los Angeles, CA 90033, USA*
mbaya@usc.edu

Abstract

Unmanned aerial systems (UASs) can only be deployed if they can effectively complete their missions and respond to failures and uncertain environmental conditions while maintaining safety with respect to other aircraft as well as humans and property on the ground. In this paper, we design a real-time, on-board system health management (SHM) capability to continuously monitor sensors, software, and hardware components for detection and diagnosis of failures and violations of safety or performance rules during the flight of a UAS. Our approach to SHM is three-pronged, providing: (1) real-time monitoring of sensor and/or software signals; (2) signal analysis, preprocessing, and advanced on-the-fly temporal and Bayesian probabilistic fault diagnosis; (3) an unobtrusive, lightweight, read-only, low-power realization using Field Programmable Gate Arrays (FPGAs) that avoids overburdening limited computing resources or costly re-certification of flight software due to instrumentation. Our implementation provides a novel approach of combining modular building blocks, integrating responsive runtime monitoring of temporal logic system safety requirements with model-based diagnosis and Bayesian network-based probabilistic analysis. We demonstrate this approach using actual data from the NASA Swift UAS, an experimental all-electric aircraft.

1. Introduction

Modern unmanned aerial systems (UASs) are highly complex pieces of machinery combining mechanical and electrical subsystems with complex software systems and controls, such as the autopilot. Rigorous requirements for safety, both in the air and on the ground, must be met so as to avoid endangering other aircraft, people, or property. Even after thorough pre-flight certification, mission-time diagnostics and prognostics capabilities are required to react to unforeseeable events during operation. In case of problems and faults in components, sensors, or the flight software, the on-board diagnosis capability must be able to detect and diagnose the failure(s) and respond in a timely manner, possibly by triggering mitigation actions. These actions can range from a simple mode change to following a pre-programmed flight path (in case of minor problems, such as a lost communications link) to a controlled emergency landing in a remote and safe area (in case of more severe problems).

Most current UAS systems, however, only have very rudimentary fault detection systems. There is a need for advanced health management systems that, in case of anomalies, can quickly and reliably pinpoint failures, carry out accurate diagnosis of unexpected scenarios, and, based upon the determined root causes, make informed decisions that maximize capabilities to meet mission objectives while maintaining safety requirements and avoiding safety hazards.

In this paper, we describe a novel framework for the design and realization of a powerful, real-time, on-board sensor and software health management system that can (a) dynamically monitor a multitude of sensor and software signals; (b) perform substantial reasoning for fault diagnosis; and (c) avoid interfering in any way with the flight software or hardware or impeding on scarce on-board computing resources.

To this end, we have developed a three-pronged approach that combines the capabilities of temporal logic runtime monitors, model-based analysis, and powerful probabilistic reasoning using Bayesian networks (BNs) (Pearl, 1988; Darwiche, 2009). Models are designed using a number of different building blocks for advanced temporal monitoring, model-based filtering, signal processing, prognostics, and Bayesian reasoning. Figure 1 shows a high-level representation of such a model. In this example, raw sensor or software signals are first fed into a smoothing block to weed out sensor noise. Then, one signal is fed into a temporal monitor, which produces a value indicating whether the temporal formula is valid, not valid, or unknown at this point in time. The other signal is fed as a measurement into a Kalman filter. The outputs of both blocks are fed into a Bayesian network block, which performs statistical reasoning and produces posterior probabilities of a fault mode (see also Mengshoel et al., 2008; Ricks & Mengshoel, 2009a, 2009b, 2010; Mengshoel et al., 2010).

The simple example in Figure 1 shows how an SHM capability can be constructed in a scalable, modular, and hierarchical manner and highlights the potential benefit of our three-pronged approach. It separates temporal properties, model-specific properties, and the (time and memory-free) probabilistic reasoning into separate components that are *easy* to model and *efficient* to execute. Our framework encourages this separation of concerns.

In this paper, we discuss in detail the three major building blocks and describe a novel method to implement such a health management system on a Field Programmable Gate Array (FPGA) for highly efficient processing and minimal intrusiveness. We detail how to instrument NASA's Swift UAS with this new SHM capability.

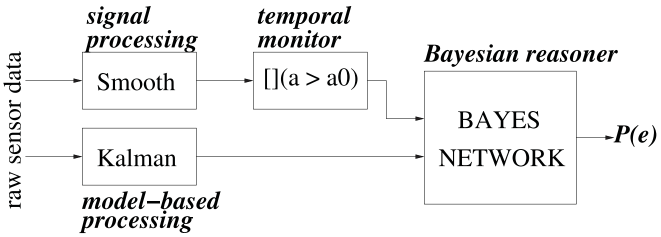


Figure 1. An example instantiation of an SHM model, illustrating one possible interconnection between signal processing, temporal monitoring, model-based processing, and Bayesian reasoning blocks.

Monitoring Sensors and Software. On-board sensor readings are used by on-board software during flight; any flight-time sensor failures should be detected and diagnosed. However, there can be dangerous interactions between the sensors and the software. Perfectly working sensors can trigger software faults, when, for example, operating in an unexpected environment. Alternatively, a faulty sensor can cause unexpected software behavior, e.g., originating from a dormant software bug. Finally, sensor and software failures can trigger issues in entirely different subsystems. For example, a software failure in the navigation system can cause the communication system to fail, as happened when F-22 Raptors crossed the international date-line on their deployment to Kadena in 2007 (Johnson, 2007).[1]

Although pre-deployment verification and validation (V&V) can be very effective in minimizing bugs in on-board software, it is impossible to eliminate all software bugs due to the size and complexity of the software as well as unanticipated, and therefore unmodelled, environmental conditions. The need to catch fault scenarios not detected by pre-deployment V&V is even more pressing when considering software in unmanned systems, since these systems often do not have to undergo the same highly rigorous and costly V&V processes required for manned systems (e.g., according to DO-178C (RTCA, 2012) for commercial transports).

It is therefore mandatory for *both* sensor and software monitoring to be performed during flight, for failure and faults to be detected and diagnosed reliably, and for root-cause analysis to be performed in real time. Only then can appropriate mitigation strategies be activated to recover the UAS or return it to the ground in the safest possible manner.

Temporal and Model-based Data Processing. The collection of all readings of sensors and software state comprises a high-dimensional and fast (around 20Hz or more) real-time data stream that needs to be processed by our health management system. On a high level, our approach to coordinated, multilevel system-wide sensor and software monitor-

[1] See Section 8.3 for a more detailed discussion.

ing transforms this fast, high-dimensional data stream into an informed basis for making intelligent decisions. We discuss a new method for runtime monitoring of temporal logic system safety requirements, in order to enable better probabilistic reasoning compared to what was previously possible. Model-based data processing components include, for example, Kalman filters, Fast Fourier Transforms, or a model-based prognostics component. We can thus reason about sensors, software, and the outputs of prognostics components (e.g., end of useful component life) in a single framework.

Bayesian Reasoning. Our framework uses a Bayesian network to perform diagnostic reasoning and root cause analysis. Although dynamic BNs (DBNs) have, in theory, the capability to directly process high-dimensional time-series data, such an approach may not be realistic in many applications due to scalability limitations and high computational requirements. We therefore *separate* the processing of temporal and model-based aspects of the data from the actual statistical reasoning part.

In order to address practical considerations including sensor-software interdependencies, the demands of real-time temporal and model-based data processing, and Bayesian reasoning for decision making, we present a novel modeling framework for sensor and software health monitoring. The framework separates model-based analysis, temporal monitoring, and statistical reasoning, thus making SHM more efficient, easier to model, and more robust. To enable its application in real-time systems, e.g., on-board of unmanned aerial systems, we will demonstrate how this framework, using temporal logic monitors, model-based preprocessing units, and static Bayesian networks, facilitates modular model design and can be executed highly efficiently on FPGA hardware.

The rest of this paper is structured as follows. After discussing related approaches in Section 2, we introduce our problem domain in Section 3, including the architecture of NASA's Swift UAS and the requirements that must be met for its safe operation. In Section 4, we discuss major design requirements for our approach and present an overview of the building blocks comprising our SHM framework. In the following sections, we give further details of the major components of this framework, namely monitors using temporal logic in Section 5, model-based monitors in Section 6, and Bayesian reasoning components in Section 7. We then provide further details on our implementation of all these components, and discuss experimental results for flight test data from the Swift UAS in Section 8. Section 9 discusses future work and concludes.

2. Related Work

2.1. System Health Management

Vehicle health management performs similar tasks to Fault Detection, Diagnosis, and Recovery (FDDR). There exist many FDDR approaches and (commercial) tools that are being actively used in the aerospace industry. For example, TEAMS[2] is a model-based tool used for diagnosis and test planning. It enables hierarchical, multi-signal diagnosis, but does not model temporal or probabilistic relationships. The underlying paradigm of FACT[3] is fault propagation with temporal constraints. More complex diagnosis systems like HyDE[4] execute simplified dynamical models on various abstraction levels and compare model results against signal values for fault detection and diagnosis. Livingston[5] is a NASA open-source diagnosis and recovery engine that uses a set of high-level qualitative models; the behaviors are specified in temporal logic. Formal V&V for such models have been carried out using the SMV model checker (Lindsey & Pecheur, 2004).

Bayesian networks are also useful for fault detection, diagnosis, and decision making because of their ability to perform deep reasoning using probabilistic models. Likelihood of failures, for example, expressed as mean-time between failure (MTBF), can be cleanly integrated. Whereas there are a number of tools for Bayesian reasoning (e.g., SamIam[6] or Hugin Expert[7]), they have not been used extensively for system health management, in part because of computationally intensive reasoning algorithms.

Fortunately, this situation has started to change. A testbed for electrical power systems in aerospace vehicles, the NASA ADAPT testbed (Poll et al., 2007), has been used to benchmark several system health management techniques. One of them is ProADAPT, a system health management algorithm using Bayesian networks (Ricks & Mengshoel, 2009a, 2009b, 2010). ProADAPT uses compilation of Bayesian networks into arithmetic circuits (Darwiche, 2003; J. Huang, Chavira, & Darwiche, 2006; Chavira & Darwiche, 2007) for efficient sub-millisecond computation. In addition, ProADAPT demonstrates how to diagnose a comprehensive set of faults, including faults of a continuous and dynamic nature, by means of discrete and static Bayesian networks. This work also shows how Bayesian system health models can be generated automatically from electrical power system wiring diagrams (Mengshoel et al., 2008, 2010).

[2] http://www.teamqsi.com/products/teams-designer/

[3] http://w3.isis.vanderbilt.edu/Projects/Fact/Fact.htm

[4] http://ti.arc.nasa.gov/tech/dash/diagnostics-and-prognostics/hyde-diagnostics/

[5] http://ti.arc.nasa.gov/opensource/projects/livingstone2/

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

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

2.2. Runtime Verification

Existing methods for Runtime Verification (RV) (Barringer et al., 2010) assess system status by automatically generating (mainly software-based) observers to check the state of the system against a formal specification. Observations in RV are usually made accessible via software instrumentation (Havelund, 2008); they report only when a specification has passed or failed, e.g., through adding hooks in the code base to detect changes in the state of the system being monitored. Such instrumentation may make re-certification of the system onerous, alter the original timing behavior, or increase resource consumption (Pike, Niller, & Wegmann, 2011); we seek to avoid this problem. Also, reporting only the outcomes of specifications does not provide the real-time responsiveness we require for our SHM framework.

Systems in our applications domain often need to adhere to timing-related flight rules like: *after receiving the command "takeoff" reach an altitude of 600 ft within five minutes*. These flight rules can be easily expressed in temporal logics; often in some flavor of Linear Temporal Logic (LTL) (Bauer, Leucker, & Schallhart, 2010). To reduce runtime overhead, restrictions of LTL to its past-time fragment have been used for RV applications previously, mainly due to promising complexity results (Basin, Klaedtke, & Zălinescu, 2011; Divakaran, D'Souza, & Mohan, 2010). Though specifications including past time operators may be natural for some other domains (Lichtenstein, Pnueli, & Zuck, 1985), flight rules like those we must monitor for the Swift UAS require future-time reasoning. To enable more intuitive specifications, others have studied monitoring of future-time claims; see (Maler, Nickovic, & Pnueli, 2008) for a survey and (Geilen, 2003; Thati & Roşu, 2005; Divakaran et al., 2010; Maler, Nickovic, & Pnueli, 2005, 2007; Basin, Klaedtke, Müller, & Pfitzmann, 2008) for algorithms and frameworks. Most of these RV algorithms, however, were designed with a software implementation in mind and require powerful computers that would far exceed the weight, size, power, bandwidth, and other limits of the Swift UAS.

2.3. Hardware Architectures

The above approaches to system health management are typically implemented in software executing on traditional CPUs. However, with the recent developments in parallel computing hardware, including in many-core graphics processing units (GPUs), Bayesian inference can be performed more efficiently (Kozlov & Singh, 1994; Namasivayam & Prasanna, 2006; Xia & Prasanna, 2007; Silberstein, Schuster, Geiger, Patney, & Owens, 2008; Kask, Dechter, & Gelfand, 2010; Linderman et al., 2010; Jeon, Xia, & Prasanna, 2010; Low et al., 2010; Bekkerman, Bilenko, & Langford, 2011; Zheng, Mengshoel, & Chong, 2011; Zheng & Mengshoel, 2013). Several of the recent many-core algorithms are based on the junction tree data structure, which can be compiled from a BN (Lauritzen & Spiegelhalter, 1988; Dawid, 1992; C. Huang & Darwiche, 1994; Jensen, Lauritzen, & Olesen, 1990). Junction trees can be used for both marginal and most probable explanation (MPE) inference in BNs. A data parallel implementation for junction tree inference was developed already in the mid-1990s (Kozlov & Singh, 1994), and the basic sum-product computation has been implemented in a parallel fashion on GPUs (Silberstein et al., 2008). Based on the cluster-sepset mapping method (C. Huang & Darwiche, 1994), node-level parallel computing techniques have recently been developed for GPUs (Zheng et al., 2011; Zheng & Mengshoel, 2013), resulting in as much as a 20-fold speed-up in processing compared to sequential techniques.

Other authors have used the benefits of a hardware architecture to natively answer statistical queries on BNs. For example, Lin, Lebedev, and Wawrzynek (2010) discuss a BN computing machine with a focus on high throughput. Their architecture contains two switching crossbars to interconnect process units with memory. Their implementation, however, targets a resource intensive grid of FPGAs, making this approach unsuitable for our purposes. Kulesza and Tylman (2006) present another approach to evaluate Bayesian networks on reconfigurable hardware. Their approach targets embedded systems as execution platforms and is based on evaluating Bayesian networks through elimination trees. The major drawback of their approach is that the hardware structure is tightly coupled with the elimination tree and requires that the hardware be re-synthesized with every change in the BN.

3. System Background

Due to the increasing interest in using unmanned aircraft for different military, civilian, and scientific applications, NASA has been engaged in UAS research since its inception. The Swift aircraft was designed to support NASA's research interests in aeronautics and earth science—particularly in autonomy, intelligent flight control, and green aviation. For safe operation, the UAS must meet a large number of requirements that in large part come from NASA and FAA processes and standards. In the following, we will briefly describe the characteristics of the Swift UAS and discuss types of safety requirements and flight rules.

3.1. The NASA Swift UAS

For full scale flight testing of new UAS concepts, the NASA Ames Research Center has developed the Swift UAS (Ippolito, Espinosa, & Weston, 2010), a 13 meter wingspan all-electric experimental platform based upon a high-performance sailplane (Figure 2). Swift has a full-featured flight computer and control for sensor payloads. The individual components are connected via a common bus interface and running a C/C++ reflection architecture, which provides a component-based plug-and-play infrastructure. Typical sensors include barometric altitude sensor, airspeed indicator, GPS, and a laser altimeter to measure the altitude above ground.

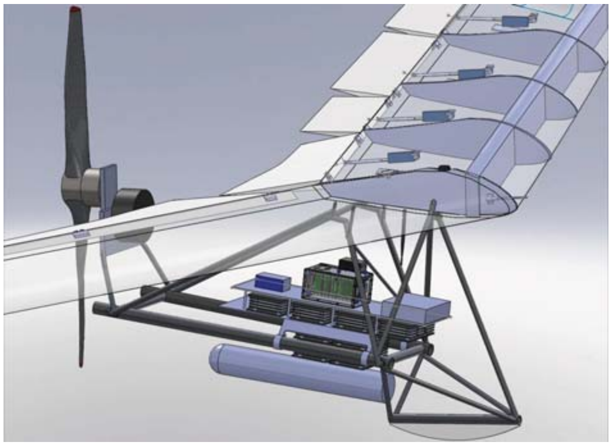


Figure 2. The Swift all-electric UAS.

3.2. Requirements and Flight Rules

The system safety requirements we want to monitor during operation of the Swift UAS can be categorized into these three types: value checks, relationships, and flight rules.

Value Checks test whether data values are plausible. Examples in this category include range checks, e.g., the maximal safe climb or descent rate. For safe operation, the values must always stay within certain ranges. Such checks can be combined with additional conditions (e.g., during the flight phase or above a minimal altitude) or temporal ranges (e.g., the maximal current drawn from the battery must not exceed 50A for more than 60 seconds to avoid overheating).

Relationships encode dependencies among sensor data that may originate from different subsystems. For example, altitude readings obtained by GPS and barometric altitude should be highly correlated. For another example, whenever the Swift UAS is in the air, its indicated airspeed reading should be greater than its stall speed; if not there is certainly a problem.

Finally, *Flight Rules* are defined by national or international institutions (e.g., part 91 of the Federal Aviation Regulations (FAR) in the USA (Federal Aviation Administration, 2013)) or by mission/system constraints that govern flights. For example, a common flight rule defines the minimum altitude an aircraft needs to climb to after takeoff: reach an altitude of 600ft within five minutes after takeoff. In a similar way, we can specify a timeout for the landing procedure of the Swift UAS: after receiving the landing command, touchdown needs to take place within three minutes. We discuss in detail how these requirements and flight rules can be specified in our framework and how they can be translated into efficient hardware.

4. System Health Management Framework

Our modeling framework for sensor and software health management separates signal processing and model-based analysis, temporal monitoring, and statistical reasoning with BNs. We first discuss the overarching design requirements before we focus on the description of the design framework. Each of the framework's three prongs will then be described in detail in the subsequent sections.

4.1. Design Requirements

For autonomous systems running on ultra-portable hardware such as the Swift UAS, the following properties are required for a deployable SHM framework.

Unobtrusiveness The SHM framework must not alter crucial properties of the Swift UAS, such as: functionality (not change its behavior), certifiability (avoid re-certification of, e.g., autopilot flight software or certified hardware), timing (not interfere with timing guarantees), and tolerances (not exhaust size, weight, power, or telemetry bandwidth constraints). The framework must be able to run and perform analysis externally to the (previously developed and tested) Swift architecture.

Responsiveness The framework must continuously and in real time monitor adherence to the safety requirements of the Swift UAS. Changes in the validity of monitored requirements must be detected within tight and a priori known time bounds. Responsive monitoring of specifications enables responsive input to the BN-based probabilistic reasoner. In turn, the BN reasoner must efficiently support decision-making to mitigate any problems encountered (e.g., for the Swift UAS an emergency landing in case the flight computer fails) to avoid damage to the UAS and its environment.

Realizability The framework must operate in a plug-and-play manner by connecting via a read-only interface to existing communication interfaces of the Swift UAS. The framework must be usable by test-engineers without assuming in-depth knowledge of hardware design and must be able to operate on-board existing UAS components without requiring significant re-configuration or additional components. The framework must be reconfigurable so that health models can be updated without a lengthy re-compilation process and can be used both during testing of the UAS and after deployment.

Considering these requirements, it seems natural to implement our SHM framework in hardware. This allows us to build a self-contained unit, operating externally to the established Swift UAS architecture, thereby complying with the UNOBTRUSIVENESS requirement. Multiple safety requirements can be monitored in parallel, with status updates delivered at every tick of the system clock, establishing the RESPONSIVENESS requirement. Previous implementations of system monitors in hardware, however, have often violated the REALIZABILITY requirement as a reconfiguration, e.g., due to changes in the SHM model, necessitates a redesign of the framework's hardware.[8] To provide greater flexibility in this regard, we design an efficient, highly parallel hardware architecture that runs on the Swift UAS' native FPGA hardware, yet keep it programmable and modular to quickly adapt to changes in our SHM models.

4.2. Design Framework

Our SHM model is constructed hierarchically in a graphical manner out of powerful building blocks. In contrast to most existing systems, we do not separate between an (informal) signal preprocessing step and the proper health management model. Rather, we elevate all processing steps to first class status and model them all in the same framework. With that approach, we can, in a principled way, deal with all temporal, probabilistic, and model-based aspects of our health management model. This uniform way of describing the health model not only enables more powerful techniques for V&V but it also directly leads to efficient implementations in programmable FPGA hardware.

All signals considered in our SHM model are streams of data, which are processed at fixed time stamps. At each tick of the system clock, a component reads the input values and calculates the output values. The order of execution is defined by a model graph. In this paper, we assume that there exists one fixed update rate for all of the building blocks of the model.[9]

Such a stream of individual elements of type T is denoted by $T^\star$; vectors are defined naturally. Table 1 shows the different data types that are used. Please note that Boolean data types are implicitly converted into a $\{0, 1\}$ discrete representation.

All data blocks have a number of inputs I_j of a given stream type, and will produce a number of outputs, again as elements of a stream. Table 2 shows a list of selected blocks. Only the current values of the signals are presented to the model. Depending on its functionality, a block can be memory-less (e.g., a Boolean function or a Bayesian reasoning block), or it can contain internal memory (of fixed length) to deal with previous signal values. Blocks with internal memory include the Linear Temporal Logic (LTL) processing blocks and blocks for data smoothing, integration, or model-based Kalman filters (Table 2).

Table 1. Data types for SHM components.

Signal Symbol	Data Type Description
R	floating point number, e.g., sensor reading
D	discrete set $\{1, \dots, n\}$
B	Boolean
B^+	*true*, *false*, *maybe*
p	probability
P	probability density

Table 2. Typical SHM building blocks.

Name	Function	Memory	Description
BF	$B^n \rightarrow B^m$		Boolean function
LTL$_s$	$B^n \rightarrow B^{+m}$		synchronous LTL observer
LTL$_a$	$B^n \rightarrow B^m$	○	asynchronous LTL observer
THR	$R^n \rightarrow D^m$		discretizer/threshold
FLT	$R^n \rightarrow R^n$	○	smoothing filter
KF	$R^n \rightarrow R^{2m+n}$	○	Kalman filter with $\hat{x}$, residual and $diag(P^-)$
BN	$D^n \rightarrow R^m$		discrete Bayes reasoner
BN$_p$	$R^l \rightarrow R^{2m}$		Bayes reasoner with evidence inputs and posterior outputs
P	$R^l \rightarrow R^2$		Prognostics unit

For example, a block to discretize sensor readings would take a floating-point number and calculate its discretized value or a Boolean value for a simple threshold. A smoothing filter, calculating a moving average, would have the functionality $R^n \rightarrow R^n$ and obviously require internal memory. A temporal monitoring component has Boolean inputs and produces a 2 or 3-valued logical output that indicates whether the monitored requirement is $true$, $false$, or $unknown$ given the inputs. In Section 5 we will discuss such monitors in detail. Our BN (see Section 7 for details) takes discrete values as inputs, called *evidence*, and produces a posterior probability. Model-based prognostics units, which take sensor signals as inputs and output estimates of remaining useful life for specific components, can substantially increase the modeling power and reasoning capabilities of our SHM framework. For example, a loss in propeller RPM might be diagnosed differently if it is known that the battery might already be fairly weak.

Beyond the building blocks shown in Table 2, additional filters, Fourier transforms, or model-based components can easily be added to improve the modeling capabilities of our SHM

[8] Or at least a run of a logic synthesis tool, which can easily take tens of minutes to complete.

[9] If signals are to be considered with different rates, rate conversion blocks or sample and hold blocks can be defined and used as needed. Note also that Bayesian networks handle missing data in a natural way and do not need conversion, sample, or hold blocks.

framework. For most of the components, efficient designs for programmable hardware are available; for our temporal monitors and Bayesian reasoning building blocks, our hardware implementations will be discussed in the sections below.

The main goal of our SHM framework is to provide a modeling paradigm that allows the modeler to separate temporal, functional, model-based, and probabilistic reasoning in a clear way while retaining the expressive power of the various formalisms. This framework also avoids powerful but complex modeling mechanisms, like dynamic Bayesian networks (DBNs).

5. Monitoring of Temporal sensor data using temporal logic

In order to encapsulate the safety requirements of the Swift UAS in a precise and unambiguous form that can be analyzed and monitored automatically, we write them in temporal logic. Specifically, we use a Linear Temporal Logic (LTL), which allows the expression of requirements over timelines and also pairs naturally with the original English expression of the requirements.[10] For requirements that express specific time bounds, we use a variant of LTL that adds these time bounds, called Metric Temporal Logic (MTL). We can automatically generate runtime monitors for requirements expressed in these logics, enabling real-time analysis of sensor data as well as system health assessment.

Linear temporal formulas consist of:

1. **Variables representing system state:** We include variables representing the data streaming from each sensor aboard the Swift UAS.
2. **Propositional logic operators:** These include the standard operators, logical AND ($\wedge$), logical OR ($\vee$), negation ($\neg$), and implication ($\rightarrow$).
3. **Temporal operators:** These operators express temporal relationships between events including Always, Eventually, NextTime, Until, and Release where the following hold for example system Boolean variables p and q.
    - Always p ($\Box p$) means that p must be true at all times along the timeline.
    - Eventually p ($\Diamond p$) means that p must be true at some time, either now or in the future.
    - NextTime p ($\mathcal{X} p$) means that p must be true in the next time step; in this paper a time step is a tick of the system clock aboard the Swift UAS.
    - p Until q ($p \mathcal{U} q$) signifies that either q is true now, at the current time, or else p is true now and p will remain true consistently until a future time when q must be true. Note that q must be true sometime; p cannot simply be true forever.
    - p Releases q ($p \mathcal{R} q$) signifies that either both p and q are true now or q is true now and remains true unless there comes a time in the future when p is also true. Note that in this case there is no requirement that p will ever become true; q could simply be true forever. The Release operator is often thought of as a “button push” operator: pushing button p triggers event $\neg q$.

For MTL, each of these temporal operators are accompanied by upper and lower time bounds that express the time period during which the operator must hold. Specifically, MTL includes the operators $\Box_{[i,j]}\ p$, $\Diamond_{[i,j]}\ p$, $p\ \mathcal{U}_{[i,j]}\ q$, and $p\ \mathcal{R}_{[i,j]}\ q$ where the temporal operator applies in the time between time i and time j, inclusive. Additionally, we use a *mission bounded* variant of LTL where these time bounds are implied to be the start and end of the mission of the UAS. In all cases, time steps refer to ticks of the system clock. So, a time bound of $[3, 8]$ would designate the time bound between 3 and 8 ticks of the system clock from now. Note that this bound is relative to “now” so that continuously monitoring a formula $\Diamond_{[3,8]}\ p$ would produce *true* at every time step t for which p holds anytime between 3 and 8 time steps after t, and *false* otherwise.

Figures 3 and 4 give pictorial representations of how these logics (mission-bounded LTL and MTL) enable the precise specification of temporal safety requirements in terms of timelines.

Examples of System Requirements in Temporal Logic. Due to their intuitive nature and a wealth of tools and algorithms for analysis of LTL and MTL formulas, these logics are frequently used for expressing avionics system requirements (Zhao & Rozier, 2012; Gan, Dubrovin, & Heljanko, 2011; Bolton & Bass, 2013; Alur & Henzinger, 1990). Recall the example system safety requirements from Section 3.2. We can straightforwardly encode each of value checks, relationship requirements, and flight rules as temporal logic formulas to enable runtime monitoring:[11]

Value Checks:

- The maximal safe climb and descent rate V_z of the Swift UAS is limited by its design and engine characteristics.

$$\Box(-200\frac{\text{ft}}{\text{min}} \leq V_z \leq 150\frac{\text{ft}}{\text{min}})$$

[10] In the temporal logic formulas of this paper, we follow the standard syntax for evaluating temporal properties where = means assignment and == means equality comparison. For example, $(a == b)$ returns *true* if a and b are equal and *false* otherwise. At the same time, we follow the tradition in probability, where = means equality and not assignment. It should be clear from the context whether we are dealing with a temporal logic expression or a probability expression.

[11] The numbers given below are for illustration purpose only and do not reflect the actual properties of the Swift UAS.

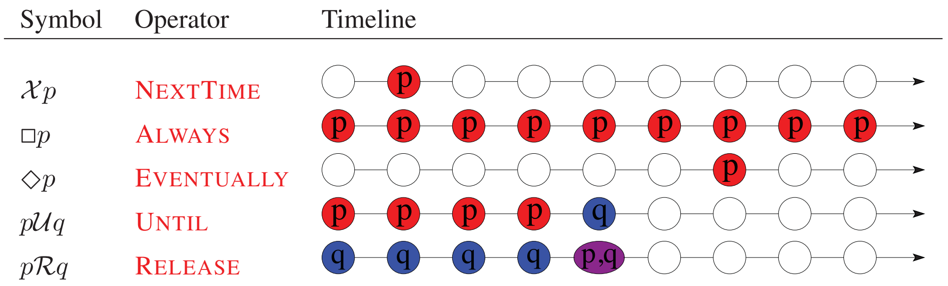


Figure 3. Pictorial representation of LTL temporal operators. For a formal definition of LTL, see for example (Rozier, 2011).

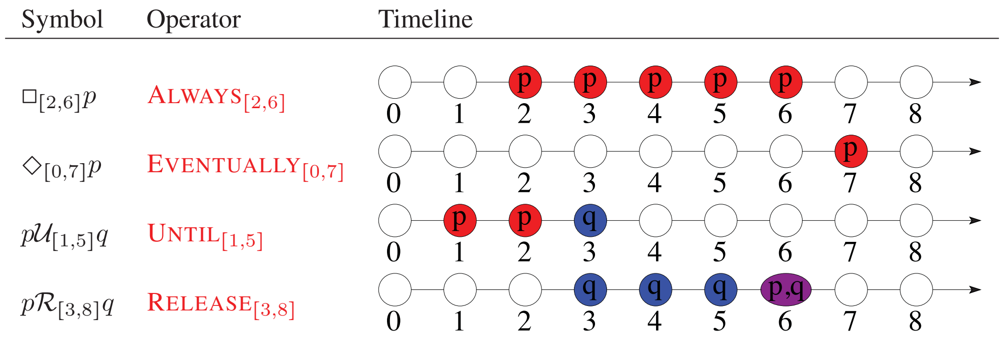

Symbol
Operator
Timeline
$\Box_{[2,6]}p$ ALWAYS$_{[2,6]}$
$\Diamond_{[0,7]}p$ EVENTUALLY$_{[0,7]}$
$p\mathcal{U}_{[1,5]}q$ UNTIL$_{[1,5]}$
$p\mathcal{R}_{[3,8]}q$ RELEASE$_{[3,8]}$
0 1 2 3 4 5 6 7 8
p
q
p,q

Figure 4. Pictorial representation of MTL temporal operators. For a formal definition of MTL, see for example (Alur & Henzinger, 1990).

- The maximal angle of attack α is limited by design characteristics.
$$\Box(\alpha \le 15^\circ)$$
- The Swift roll (p), pitch (q), and yaw rates (r) are for safe operation limited to remain below maximum bounds.
$$\Box(p < 0.99\frac{\text{rad}}{\text{s}} \wedge q < 4.0\frac{\text{rad}}{\text{s}} \wedge r < 2.2\frac{\text{rad}}{\text{s}})$$
- The battery voltage U_{batt} and the current I_{batt} must remain within certain bounds during the entire flight. Furthermore, no more than 50A should be drawn from the battery for more than 30 consecutive seconds in order to avoid battery overheating.

$$\begin{aligned}\Box(\quad & (20V \le U_{batt} \le 26.5V) & \wedge \\ & (I_{batt} \le 75A) & \wedge \\ & ((I_{batt} > 50A)\mathcal{U}_{[0,29s]}(I_{batt} \le 50A)))\end{aligned}$$

Relationships:

- Pitching up (i.e., increasing α) for a sustained period of time (more than 20 seconds) should result in a positive change in altitude, measured by a positive vertical speed V_z. This increase in vertical speed should occur within two seconds after pitch-up.
$$\Box(\Box_{[0,20s]}(\alpha > \alpha_0) \rightarrow \Diamond_{[0,2s]}V_z > 0)$$
This relationship can be refined to only hold if the engine has enough power (as measured by the electrical current to the engine I_{eng}) to cause the aircraft to actually climb.
$$\Box(\Box_{[0,20s]}((\alpha > \alpha_0) \wedge I_{eng} > 30A) \rightarrow \Diamond_{[0,2s]}V_z > 0)$$
Similarly, a rule for the descending can also be defined.
$$\Box(\Box_{[0,20s]}((\alpha < \alpha_0) \vee I_{eng} < 10A) \rightarrow \Diamond_{[0,2s]}V_z < 0)$$
- Whenever the Swift UAS is in the air, its indicated airspeed (V_{IAS}) must be greater than its stall speed V_S. The UAS is considered to be air-bound when its altitude alt is larger than that of the runway alt_0.
$$\Box((alt > alt_0) \rightarrow (V_{IAS} > V_S))$$
- The sensor readings for the vertical velocity V_z and the barometric altimeter alt_b are correlated, because V_z corresponds to the changes in the altitude. This means that whenever the vertical speed is positive, we should measure a certain increase of altitude Δ_{alt_b} within 2 seconds.
$$\Box(V_z > 0 \rightarrow \Diamond_{[0,2s]}\Delta_{alt_b} > \theta)$$
- The precision of the position reading P_{GPS} from the

GPS subsystem depends on the number of visible GPS satellites N_{sat}.

$$\begin{aligned}
&\Box(\\
&\Box(N_{sat} == 1) && \rightarrow P_{GPS} \le P^{1}_{GPS} && \wedge \\
&\Box(N_{sat} == 2) && \rightarrow P_{GPS} \le P^{2}_{GPS} && \wedge \\
&\Box(N_{sat} == 3) && \rightarrow P_{GPS} \le P^{3}_{GPS} && \wedge \\
&\Box(N_{sat} \ge 4) && \rightarrow P_{GPS} \le P^{+}_{GPS})
\end{aligned}$$

Flight Rules:

- After receiving a command (cmd) for takeoff, the Swift UAS must reach an altitude of 600ft within 40 seconds.

$$\Box((\text{cmd} == \text{takeoff}) \rightarrow \Diamond_{[0,40s]}(alt \ge 600\text{ ft}))$$

- After receiving the landing command, touchdown needs to take place within 40 seconds, unless the link (lnk) is lost. Otherwise, the aircraft should reach a loitering altitude around 425ft within 20 seconds.

$$\begin{aligned}
&\Box((\text{cmd} == \text{landing}) \rightarrow \\
&((s_{lnk} == \text{ok}) \rightarrow \Diamond_{[0,40s]}(alt < 10\text{ ft}) \vee \\
&(s_{lnk} == \text{lost}) \rightarrow \Diamond_{[0,20s]}(400\text{ft} \le alt \le 450\text{ft})))
\end{aligned}$$

- The Swift default mode is to stay on the move; it should not loiter in one place for more than a minute unless it receives the loiter command (which may not ever happen during a mission). Let *sector_crossing* represent a Boolean variable, which is true if the UAS crosses the boundary between the small subdivision of the airspace in which the UAS is currently located, as determined by the GPS, and another subdivision. After receiving the loiter command, the UAS should stay in the same sector, at an altitude between 400 and 450ft until it receives a landing command. The UAS has 30 seconds to reach loitering position.

$$\begin{aligned}
&\Box(\big[(\text{cmd} == \text{loiter})\,\mathcal{R}\,(\Diamond_{[0,60s]}\ \text{sector\_crossing})\big] \wedge \\
&\quad [(\text{cmd} == \text{loiter}) \rightarrow \\
&\qquad (\Box_{[30s,end]}((\neg\text{sector\_crossing}) \wedge \\
&\qquad\qquad\qquad (400\text{ft} \le alt \le 450\text{ft})) \\
&\qquad\ \ \mathcal{U}\,(\text{cmd} == \text{landing})) \\
&\quad])
\end{aligned}$$

- All messages sent from the guidance, navigation and control (GN&C) component to the Swift actuators must be logged into the on-board file system (FS). Logging has to occur before the message is removed from the queue. In contrast to the requirements stated above, this flight rule specifically concerns properties of the flight software.

$$\begin{aligned}
&\Box((\text{addToQueue}_{\text{GN\&C}} \wedge \Diamond\text{removeFromQueue}_{\text{Swift}}) \rightarrow \\
&\quad \neg\text{removeFromQueue}_{\text{Swift}}\ \mathcal{U}\ \text{writeToFS})
\end{aligned}$$

Advantages of Temporal Logic Requirements. Encoding the set of system requirements in temporal logic offers several significant advantages. It yields a very precise, unambiguous list of the system requirements that aids in project planning and organization. It enables us to automatically synthesize runtime monitors to track these requirements onboard the Swift UAS directly from the temporal logic specifications. It also enables other types of automatic checks, such as automatic requirements debugging (i.e., satisfiability checking (Rozier & Vardi, 2010)) and design-time V&V techniques such as model checking (Rozier, 2011).

5.1. Monitoring Approach

From each temporal logic requirement, we automatically generate two kinds of monitors, which we call *synchronous* and *asynchronous* monitors, working in coordination to provide real-time system health updates. A *synchronous monitor* provides an update on the requirement with every update of the system clock. This is important because it provides blocks such as the Bayesian reasoner with better real-time information and therefore improves prognostics capabilities by enabling monitoring input to be considered by the reasoner. Our synchronous runtime monitors keep up-to-date information on how much time is left until a requirement must pass. An *asynchronous monitor* provides an update on the final outcome of the requirement at an a priori known time. Our asynchronous monitors report if a requirement is satisfied or fails earlier than expected or yield the final result (pass or fail) of the requirement when its time bound has elapsed. For details on the construction of these monitors, and formal proofs that our constructions are correct, see (Reinbacher, Rozier, & Schumann, 2013).

This dual-monitor construction is a key element of our SHM framework, because it enables our runtime monitors to be used as building blocks in combination with the other blocks described in this paper. Traditional runtime monitoring techniques only operate asynchronously and only report when a monitored property *fails*. Our monitors provide much more useful output. For example, it can be important in computing prognostics to know that a requirement that must happen within a specified time bound has not yet been satisfied and that the time bound is almost up. This allows mitigating actions to be considered in time. For another example, if a requirement states that ($\textsc{Eventually}_{[3,2005]}\ p$) and p occurs at time 5 it is important to utilize this information for real-time calculations of system health. Traditional monitoring techniques do not yield any output in this case, either at time 5 or 2005 since no property failure occurred. Finally, it is key that our runtime monitors can provide this information without any modifications to certified flight software or hardware, operating in isolation aboard an FPGA with a read-only interface, whereas most runtime monitoring techniques utilize more obtrusive techniques for gathering system data.

6. Model-based monitoring of Temporal sensor data

Highly accurate and detailed information about system health could be obtained if the actual system is compared with a high-fidelity simulation model. Model complexity and resource limitations make such an approach infeasible in most cases. However, a comparison of system behavior with an abstracted dynamical model is an attractive option. HyDE, for example, performs health management using simplified and abstracted system models.

For our framework, we provide the capability to use model-based monitoring components to various degrees of abstraction. The most common of such components is a Kalman filter. Here, a linearized model of the (sub-)system dynamics is used to predict the system state from past sensor readings. Besides this state prediction, the residual of the Kalman filter is of importance for our purposes, as it reflects how well the model represents the actual behavior (Brown & Hwang, 1997). A sudden increase of the filter residual, for example, can give an indication of a malfunctioning sensor. For implementation, we use our tool AutoFilter (Whittle & Schumann, 2004) to automatically generate customized Kalman filter algorithms from high-level requirements. As we refine our configuration to handle more complex SHM capabilities needed for future flight tests of the Swift UAS, we are planning to extend the AutoFilter tool in order to directly produce corresponding FPGA designs (see, e.g., Pasricha & Sharma, 2009). In a similar manner, non-linear models could be handled using Particle Filters (Ristic, Arulampalam, & Gordon, 2004), though these require more computational efforts.

A very simple temporal monitoring technique is the use of FFT in order to obtain an estimate of the frequency spectrum of the monitored signals. This information is, for example, important to detect oscillations of the aircraft (see Section 8.3), or to detect unexpected software behavior, like a reboot loop.

Though our implementation at this time is limited to standard filtering monitors, we envision creating more powerful model-based monitors using prognostics models to produce statistical distributions for the end-of-life of system components based upon sensor readings. For example, a prognostics model to estimate the remaining useful life of the laser altimeter could be used to effectively encode a dynamical MTBF into our health management system. Again, both the mean remaining life as well as information about its probability distribution can be directly used for reasoning. Although such model-based health management components can be very powerful, a number of issues still need to be addressed, including model validity, implementation in efficient hardware, and possible model adaptation to better detect and handle certain kinds of failures.

7. Bayesian Health Management Reasoning

The major reasoning component in our SHM framework is a Bayesian network (BN) used to perform diagnostic reasoning and root causes analysis. A BN is a multivariate probability distribution that enables reasoning and learning under uncertainty (Pearl, 1988; Darwiche, 2009). In a BN, random variables are represented as nodes in a Directed Acyclic Graph (DAG), while conditional dependencies and independencies between variables are induced by graph edges (see Figure 5 for an example). A BN's graphical structure often represents a domain's causal structure, and is typically a compact representation of a joint probability table. Each node in the BN's graphical structure is associated with a corresponding conditional probability table (CPT) that captures its (causal) relationship with parent nodes.

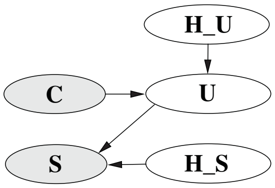


Figure 5. Simple Bayesian network.

In our framework, the BN inputs are comprised of discrete or discretized values (e.g., *low*, *high*), and reasoning is performed at each tick of the system clock. We are using discrete and static BNs, which do not perform any reasoning in the temporal domain. All temporal reasoning, as well as other processing, has been cleanly separated out within our modeling framework. Although, in general, multiple different probabilistic queries can be formulated, our framework aims to compute marginal posterior probabilities of selected nodes, which then give an indication (probability) of component or system health. Thus our Bayesian reasoning components compute a posteriori probabilities as their output. Different BN inference algorithms can be used to compute a posteriori probabilities. These algorithms include junction tree propagation (Lauritzen & Spiegelhalter, 1988; Jensen et al., 1990; Shenoy, 1989), conditioning (Darwiche, 2001), variable elimination (Li & D'Ambrosio, 1994; Zhang & Poole, 1996), stochastic local search (Park & Darwiche, 2004; Mengshoel, Roth, & Wilkins, 2011; Mengshoel, Wilkins, & Roth, 2011), and arithmetic circuit evaluation (Darwiche, 2003; Chavira & Darwiche, 2007).

7.1. Bayesian Health Models

For the Bayesian models, we follow an approach that "glues together" Bayesian network fragments (Schumann, Mengshoel, & Mbaya, 2011; Schumann et al., 2013). For example, consider the Bayesian network in Figure 5. It consists of four different types of interconnected nodes, namely: command node C, health node H, sensor node S, and status node U. The health node H has subtypes H_S for sensor nodes and H_U for status nodes. Inputs to a BN is provided by connecting an input signal to the state of a node ("clamping"). Command nodes C are handled as ground truth and used to indicate commands, actions, or other known states. Command nodes do not have incoming edges. Sensor nodes S are also input nodes, but the input signal is not necessarily correct (e.g., it could result from a failed sensor). This behavior is modeled by connecting S to a health node H that reflects the health of the input to S. Note that inputs to the BN can be outputs of any block in our framework, for example, a smoothed and discretized sensor reading, the result (binary or ternary) of a temporal monitor, or the output of another reasoning block.

Status nodes U, and similar behavior nodes B, are internal nodes and reflect the (latent) status of the subsystem or component, or recognize a specific behavior, such as pilot-induced oscillation. Typically, health nodes H are attached to status nodes, but not to behavior nodes. Associated with each node is a Conditional Probability Table (CPT), which defines the conditional probability of node X, given the states of the parent nodes of X.

For modeling the edges of the BN, we follow the rule that an edge from node X to node Y indicates that the state of X has a (causal) influence on the state of Y. Table 3 gives an overview of the different kinds of edges in our modeling framework.

Table 3. Types of edges typically used in BN models for the SHM reasoning blocks.

edge E	E represents how ...
$\{H, C\} \overset{E}{\to} U$	status U, with health H, is controlled through unreliable command C
$\{C\} \overset{E}{\to} U$	status U is controlled through unreliable command C
$\{H, U\} \overset{E}{\to} S$	status U influences sensor S, which may fail as reflected in health H
$\{H\} \overset{E}{\to} S$	health H directly influences sensor S without modeling of status
$\{U\} \overset{E}{\to} S$	status U influences sensor S

7.2. Compilation to Arithmetic Circuits

We select arithmetic circuit evaluation as the inference algorithm used in our framework, and therefore compile our Bayesian network into an arithmetic circuit. In real-time avionics systems, where there is a strong need to align the resource consumption of diagnostic computation with resource bounds (Musliner et al., 1995; Mengshoel, 2007), algorithms based upon arithmetic circuit evaluation are powerful, as they provide predictable real-time performance (Chavira & Darwiche, 2005; Mengshoel et al., 2010).

An arithmetic circuit (AC) is a DAG in which leaf nodes λ represent parameters and indicators while other nodes represent addition and multiplication operators.

Posterior marginals in a Bayesian network can be computed from the joint distribution over all variables $X_i \in \mathcal{X}$:

$$p(X_1, X_2, \ldots) = \prod_{\lambda_x} \lambda_x \prod_{\theta_{x|\mathbf{u}}} \theta_{x|\mathbf{u}}$$

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_x indicate whether or not state x is consistent with the observations. For efficient calculation, we rewrite the joint distribution into the corresponding network polynomial f (Darwiche, 2003):

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

An arithmetic circuit is a compact representation of a network polynomial (Darwiche, 2009) which, in its uncompact form, is exponential in size and thus unrealistic in the general case. Hence, answers to probabilistic queries, including marginals and MPEs, are computed using algorithms that operate directly on the arithmetic circuit. The marginal probability (see Corollary 1 in (Darwiche, 2003)) for x given evidence $\mathbf{e}$ is calculated as

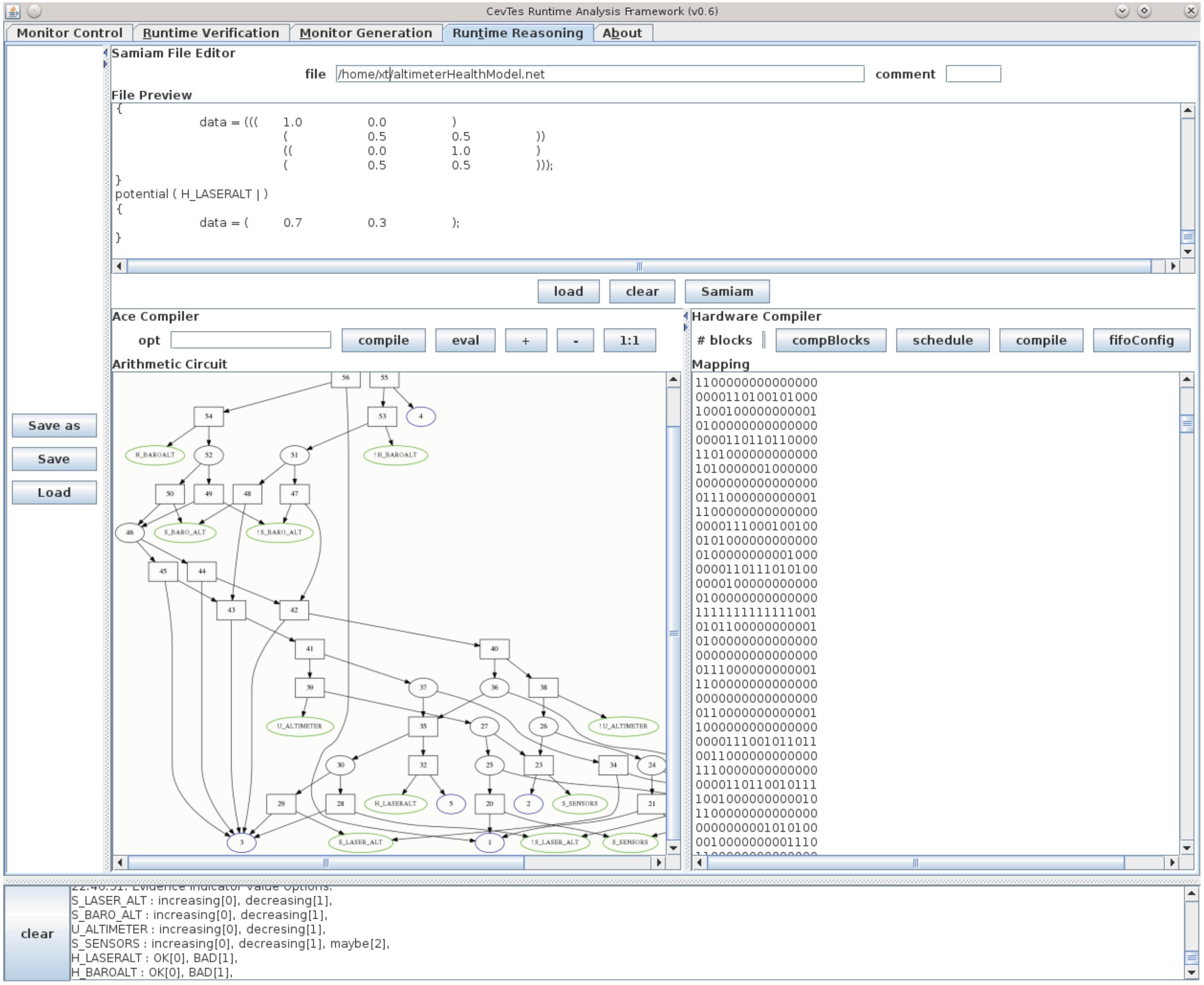
$$\Pr(x \mid \mathbf{e}) = \frac{1}{\Pr(\mathbf{e})} \cdot \frac{\partial f}{\partial \lambda_x}(\mathbf{e}) \qquad (1)$$

where $\Pr(\mathbf{e})$ is the probability of the evidence. In a bottom-up pass over the circuit, the probability of a particular evidence setting (or clamping of λ parameters) is evaluated. A subsequent top-down pass over the circuit computes the partial derivatives $\frac{\partial f}{\partial \lambda_x}$. This mechanism can also be used to provide information about how change in a specific node affects the whole network (sensitivity analysis), and to perform MPE computation (Darwiche, 2003, 2009).

7.3. Efficient Hardware Realization

Bayesian reasoning blocks in our framework are provided with values produced by other blocks, as input to C and S nodes. In our BN hardware implementation, these evidence values are used to calculate posterior marginals for the health nodes H of the Bayesian SHM model. For efficient hardware realization of this kind of BN reasoning, we note that posterior marginals are evaluated in the arithmetic circuit by traversing the nodes of the circuit in a bottom-up and a subsequent top-down manner.

We make the following observations regarding the structure



file: /home/xt/altimeterHealthModel.net

```
{
                data = (((     1.0          0.0          )
                        (      0.5          0.5          ))
                        ((     0.0          1.0          )
                        (      0.5          0.5          )));
}
potential ( H_LASERALT | )
{
                data = (       0.7          0.3          );
}
```

```
1100000000000000
0000110100101000
1000100000000001
0100000000000000
0000110110110000
1101000000000000
1010000001000000
0000000000000000
0111000000000001
1100000000000000
0000111000100100
0101000000000000
0100000000001000
0000110111010100
0000100000000000
0100000000000000
1111111111111001
0101100000000001
0100000000000000
0000000000000000
0111000000000001
1100000000000000
0000000000000000
0110000000000001
1000000000000000
0000111001011011
0011000000000000
1110000000000000
0000110110010111
1001000000000010
1100000000000000
0000000001010100
0010000000001110
```

```
22:40:51: Evidence indicator value options:
S_LASER_ALT : increasing[0], decreasing[1],
S_BARO_ALT : increasing[0], decreasing[1],
U_ALTIMETER : increasing[0], decresing[1],
S_SENSORS : increasing[0], decreasing[1], maybe[2],
H_LASERALT : OK[0], BAD[1],
H_BAROALT : OK[0], BAD[1],
```

Figure 6. Screenshot of our GUI-based BN synthesis tool. There is a textual description of the altimeter health model Bayesian network (top), a compiled arithmetic circuit of the network (bottom left), and a binary configuration for our μBayes unit (bottom right).

of arithmetic circuits:

(i) The structure alternates between addition and multiplication nodes. Nodes labeled with "+" are addition nodes; those labeled with "×" are multiplication nodes.

(ii) Each multiplication node has a single parent.

(iii) Input nodes (i.e., leaf nodes) are always children of multiplication nodes.

Hardware Architecture of μBayes. The above observations, concerning the structure of arithmetic circuits, led us to a hardware architecture that evolves around parallel units called computing blocks. A computing block, as shown in Figure 7, is designed to match the structural properties (i-iii) of an arithmetic circuit. A single computing block supports three basic modes to process the different shapes found in subtrees of arithmetic circuits. Re-arrangement of the arithmetic circuit using commutativity properties of the operators enables us to tile the entire AC with instances of these three modes in Figure 7.

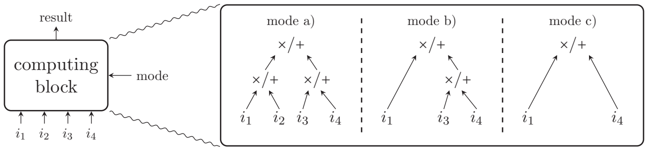


Figure 7. A computing block and its three modes of operation.

These computing blocks are the building blocks of our

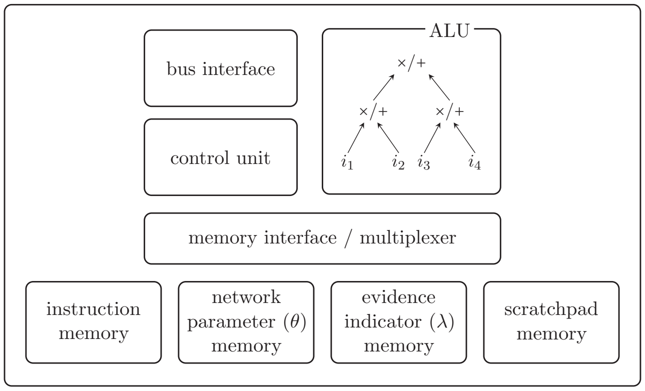

bus interface
control unit
ALU
×/+
×/+
×/+
i_1
i_2
i_3
i_4
memory interface / multiplexer
instruction memory
network parameter (θ) memory
evidence indicator (λ) memory
scratchpad memory

Figure 8. Internals of a computing block.

Bayesian SHM hardware unit, which we call μBayes.

Figure 8 shows the internals of a computing block. The unit is loaded with network parameters from the CPT of the health model at configuration time. At each SHM update cycle, inputs are provided as evidence indicators and stored in a separate evidence indicator memory. An offline compiler translates the structure of the arithmetic circuit into native instructions for the μBayes unit. Instructions encode the operation (either addition or multiplication) of each individual node of the Arithmetic Logic Unit (ALU), control the multiplexer to load/store operands from/to memory, trigger transfers of results, and coordinate loads of inputs. Instructions are decoded and forwarded by the control unit. Each computing block manages a scratchpad memory to save intermediate local results, computed during the bottom-up traversal, which can be reused during the top-down traversal. The memory blocks of the μBayes unit are mapped to block RAMs of the FPGA.

Figure 9 shows the architecture of our Bayesian health management hardware unit. It interconnects and controls multiple computing blocks to process arithmetic circuits in parallel.

The master unit manages bus accesses, stores intermediate global results, and computes posterior marginals according to Equation 1. The inverse of the probability of the evidence, $\frac{1}{\Pr(\mathbf{e})}$, in this equation can be computed within the master in parallel to the top-down traversal of the arithmetic circuit once the bottom-up traversal is completed. Posterior marginals can then be computed efficiently by multiplying the partial derivatives $\frac{\partial f}{\partial \lambda_x}$ obtained by the top-down traversal with the cached value of $\frac{1}{\Pr(\mathbf{e})}$.

For our implementation, we designed the μBayes unit in the hardware description language VHDL and use the logic-synthesis tool ALTERA QUARTUS II[12] to synthesize the design onto an Altera Cyclone IV EP4CE115 FPGA. In our implementation, we chose to represent fractional values in a fixed-point representation. This decision avoids the considerable blow-up in hardware requirements that we would incur if all of the computing blocks had to be equipped with a full-fledged floating-point unit. Instead, we instantiate fixed-point multipliers, available on our target FPGA, to realize the arithmetic operations within the computing blocks. Modern-day FPGAs provide several hundred of such multiplier units.

Synthesizing an Arithmetic Circuit into a μBayes Program. A (GUI-based) application (see Figure 6) on a host computer compiles an arithmetic circuit into a tuple $\langle \Pi, C \rangle$, where Π is a native program for the μBayes unit and C is a configuration for the network parameter memory. The synthesis of $\langle \Pi, C \rangle$ from an arithmetic circuit involves the following steps:

(1) Parse the circuit into a DAG and use compile-time information from the Ace package[13] to relate nodes in the DAG to evidence indicators and network parameters. Assemble network parameter values according to the CPTs and add them to C. Perform equivalence transformations on the DAG to ensure that the available modes of a computing block are able to cover all parts of the arithmetic circuit.

(2) Apply a variant of the Bellman-Ford algorithm (Bellman, 1958) to the DAG to determine the distance of each node to the root node. Based on the distances and the width of the arithmetic circuit, determine the number of required computing blocks. Rearrange computing blocks to optimize the number of results that can be reloaded from the same computing block in the next computation cycle.

(3) For each computing block c: generate an instruction π for each node in the arithmetic circuit that is computed by c. Finally, add π to Π.

To configure the μBayes unit, the tuple $\langle \Pi, C \rangle$ is transferred at configuration time, i.e., before deployment, to the master unit, which then programs the individual computing blocks. During operation, the entries for the evidence indicator memory are broadcast by the master unit at each tick of the system clock when new input values are available.

Hardware Resource Consumption. We synthesized the hardware design of the μBayes unit for various target FPGAs using the industrial logic synthesis tool ALTERA QUARTUS II. To study the hardware resource consumption of our design we synthesized the design several times with varying numbers of computing blocks. For our implementation, we used a fixed-point number representation with 18 bits to internally represent probabilities. We have chosen this representation

[12] Available at http://www.altera.com. We used v11.1 in our experiments.

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

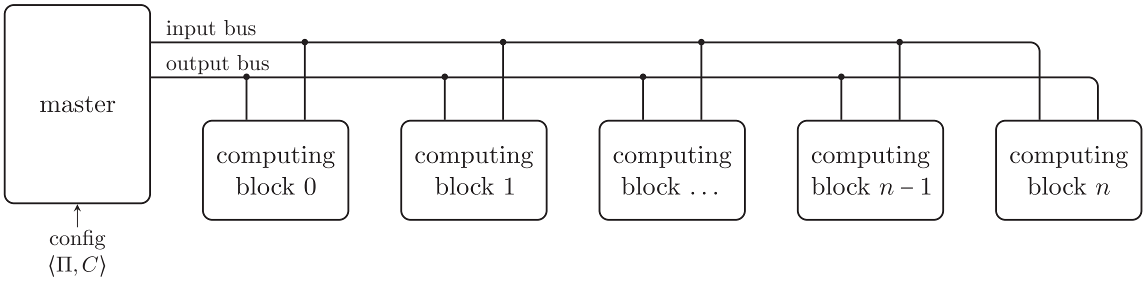


Figure 9. Architecture of the μBayes unit with parallel computing blocks.

mainly because our target FPGA provides fixed point multipliers that support vectors of up to 18 bits.

For example, an instantiation of the μBayes unit with 7 parallel computing blocks accounts for a total of 25,719 logic elements (22.5 % of the total logic elements) and 20,160 memory bits (2.5 kByte, 0.5 % of the total memory bits) and allows for a maximum operating frequency f_{max} of 115 MHz (for the slow timing model at 85 °C) on an Altera Cyclone IV EP4CE115 FPGA. We note that the operating frequency can easily be increased by moving to a more powerful FPGA. Figure 10 shows the influence of the number of computing blocks on maximum operating frequency, number of logic elements, and the number of memory bits.

8. Experiments and Results

In this section, we present results of experiments. In order to illustrate our three-pronged approach, we first discuss monitoring of requirements using examples of temporal logic monitors as presented in Section 5. In all of the examples, actual sensor and signal values are prefixed by "s_", e.g., s_baroAlt comprises a stream of sensor readings of the barometric altitude. We next discuss an example of how to determinate the health of sensors using BNs and show results using actual flight data, where the laser altimeter failed. The final part of this section is devoted to an example of how our framework can be used for reasoning about software.

8.1. Monitoring of Requirements

Recall from Section 3.2 and Figure 1 that our SHM framework operates on a set of requirements, which are interpreted via paths through a network of building blocks to achieve our diagnostics and prognostics goals. We create model-based monitors (Section 6) and Bayesian reasoning components (Section 7) to support monitoring these requirements. We create synchronous and asynchronous runtime monitors in hardware, aboard FPGAs, from our temporal logic translations of the requirements (Section 5). In this way, requirements form the backbone of our SHM framework.

Here, we exemplify the monitoring process for our temporal logic-based runtime monitors, including how they take input from and pass input to other blocks in our SHM framework. We demonstrate the power of generating monitors from temporal logic requirements.

For example, consider the requirement $\Box((\text{s\_cmd} == \text{takeoff}) \rightarrow \Diamond_{[0,40s]}(\text{s\_baroAlt} \geq 600 \text{ ft}))$ from Section 3.2 that states, "*After takeoff, the Swift UAS must reach an altitude of* 600*ft within 40 seconds*." Recall that we encoded this requirement in MTL in Section 5 and discussed creating a pair of runtime monitors that yield both a synchronous monitor that updates with each tick of the system clock and an asynchronous monitor that determines the satisfaction of the requirement as soon as there is enough information to do so.

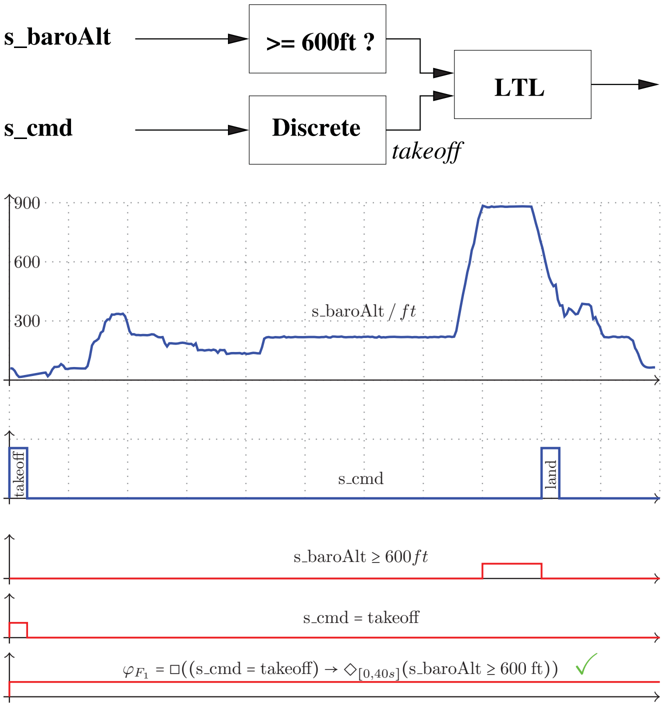


Figure 11. Top panel: SHM block diagram for monitoring requirement $\Box((\text{s\_cmd} == \text{takeoff}) \rightarrow \Diamond_{[0,40s]}(\text{s\_baroAlt} \geq 600 \text{ ft}))$. Middle two panels: flight data collected from the Swift UAS. Bottom three panels: output of our runtime monitors for flight rules.

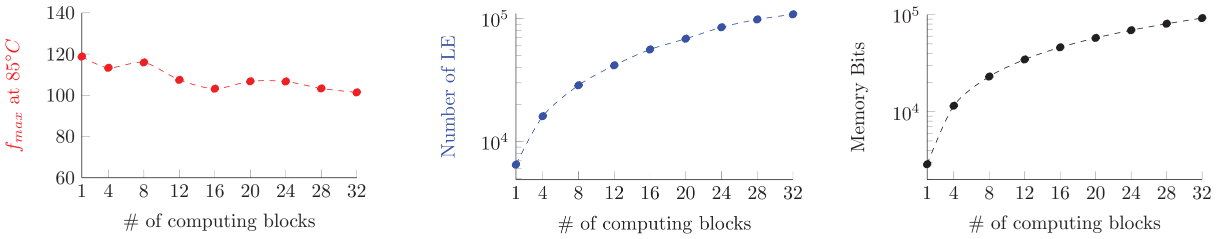


Figure 10. Logic synthesis results of our μBayes unit for an Altera Cyclone IV EP4CE115 FPGA: maximum operating frequency f_{max}, number of logic elements (LE), and required memory bits versus number of parallel computing blocks

Figure 11 breaks down how we monitor this requirement. First, the raw data from the barometric altimeter is passed through one of our smoothing filter blocks, as described in Section 4, to take out sensor noise that might serve to obscure the altimeter readings. The data stream from this smoothing filter and the raw data from the flight command data stream are the two inputs to our pair of temporal logic monitors for this requirement. These two inputs are shown in blue in Figure 11. In the bottom three panels, in red, are the output data streams from the asynchronous monitor. The top line is the result of monitoring the subformula $(\text{s\_baroAlt} \geq 600 \text{ ft})$ and the middle line is the result of monitoring the subformula $(\text{s\_cmd} == \text{takeoff})$. These two signals are combined inside our compositional monitor construction to form the result illustrated in the bottom panel. The panel's straight red line shows that the requirement holds at every time point during the flight. This bottom line is the output from our asynchronous monitor and can be used as the input to another block in our SHM framework, such as a Bayesian reasoning block.

During UAS flight, this output data stream will not be provided in real time, but will experience delays as there is not enough data at every time point of flight to determine that this requirement always holds. Therefore, any blocks in our SHM framework making real-time decisions could utilize the output from the paired synchronous monitor for this formula. It would differentiate, in real time, when we know that the flight rule holds and when we do not have enough information, at the present time, to know.

For another example, consider the requirement

$$\Box(\Box_{[0,5s]}(\text{v\_vel} > 0) \rightarrow \Diamond_{[0,2s]}\Delta_{\text{s\_baroAlt}} > \theta)$$

stating that a significant positive vertical velocity needs to be followed by a climb in altitude. Figure 12 breaks down how we monitor this requirement.

Again, we take the raw data from the barometric altimeter, pass it through one of our smoothing filter blocks to reduce the sensor noise, and feed this stream as an input to our temporal monitor. (Again, the smoothed barometric altimeter data stream appears in blue.) We also need to reason about the vertical velocity reading; we show the raw data stream in red. We feed this sensor data stream through a moving average filter; the result is shown in blue.

These two filtered data streams are then processed by components of our asynchronous runtime monitor; results are shown in the bottom three panels of Figure 12. The red line at the top, our vertical velocity monitor, checks for a "significant positive vertical velocity." System designers equate this to a steady positive reading of the filtered vertical velocity reading for five seconds. The red line in the middle, our barometric altimeter monitor, flags time points that fall within a two second time interval when the change in altitude is above the threshold θ. These components comprise our runtime monitor, which continuously verifies that "every occurrence of significant positive vertical velocity is indeed followed by a corresponding positive change in altitude." This is reflected by the straight red line in the bottom-most panel of Figure 12.

8.2. Sensor Health Management

The continuous monitoring of the UAS's flight-critical sensors is very important. Faulty, iced, or clogged pitot tubes for measuring speed of the aircraft has caused several catastrophes. For example, the crash of Birgenair Flight 301, which claimed 189 lives, was caused by a pitot tube being blocked by wasp nests[14]. Similarly, faults in the determination of the aircraft's altitude can lead to dangerous situations. In many cases, however, the health of a sensor cannot be established independently. Only by taking into account information from other sources can a reliable result be obtained. However, these other sources of information are also not independently reliable, thus creating a non-trivial SHM problem.

In the following example, we use information from a barometric altimeter measuring altitude above sea level, a laser altimeter measuring altitude above ground level (AGL), and information about the vertical velocity and the pitch angle provided by the Inertial Measurement Unit (IMU). Table 4 lists the signals and their intended meanings. Our corresponding SHM framework instantiation is shown in Figure 13. The input signals are smoothed and the current vertical velocity is estimated from the laser and barometric altimeters by calculating $x_t - x_{t-1}$ using a single delay block. Then, the val-

[14] http://en.wikipedia.org/wiki/Birgenair_Flight_301

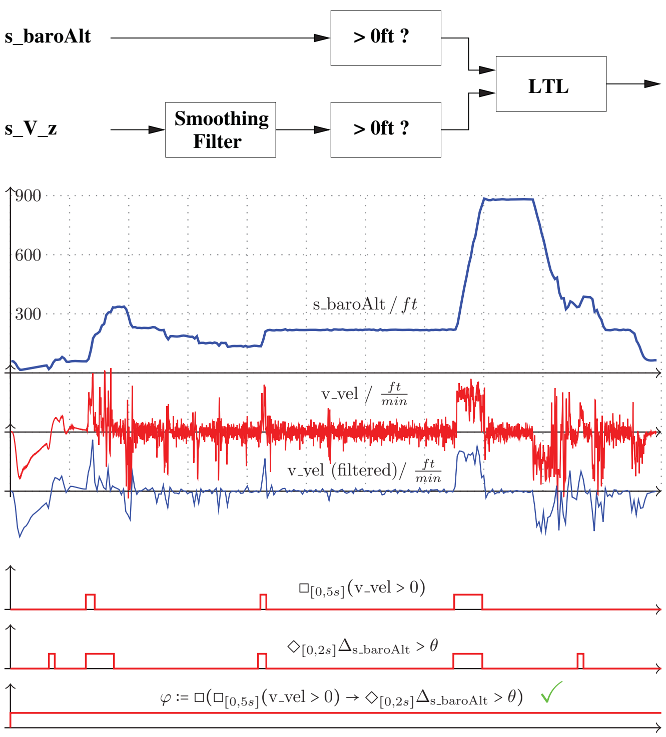


Figure 12. Top panel: SHM block diagram for monitoring a requirement. Middle three panels: Smoothed barometric altimeter (blue) and vertical velocity readings, raw (red) and smoothed (blue), as collected from the Swift UAS. Bottom three panels: outputs of temporal logic monitors.

ues are discretized into increasing (*inc*) and decreasing (*dec*), before the information is fed into the reasoning component.

Table 4. Signals and their intended meanings.

Signal name	Description
s_baroAlt	altitude reading from barometric altimeter
s_laserAlt	altitude reading from laser altimeter
s_velUp	vertical velocity reading from IMU
s_pitch	Euler pitch reading from IMU

Figure 14 shows the BN model for reasoning about altimeter failures. Sensor nodes (inputs) for each of the different sensor types are at the bottom. The latent state U_A, describing whether the altitude of the UAS is increasing or decreasing, obviously influences the sensor readings, hence there are edges from U_A to S_L, S_S, and S_B. The laser altimeter can fail. Therefore, the sensor node S_L is connected with a node H_L, reflecting the health of the laser altimeter. A similar structure can be found for the barometric altimeter. Because the laser altimeter is prone to errors, its probability of being healthy is only 0.7, compared to the more reliable barometric altitude with a probability of being healthy of 0.9. For simplicity, the health of the IMU is not modeled here.

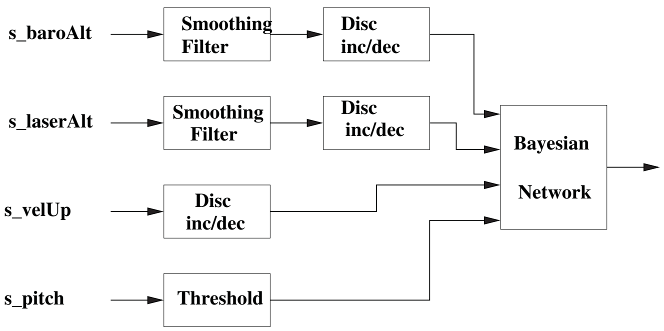
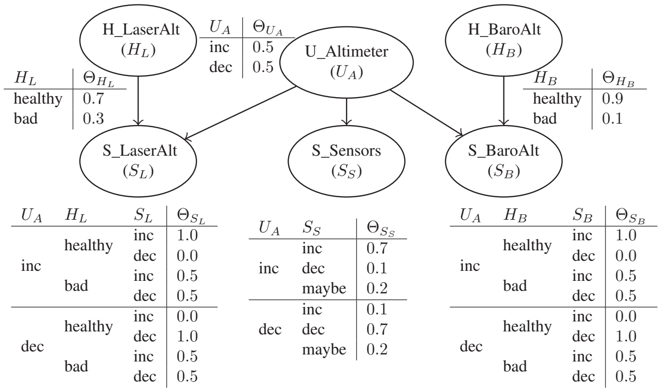


Figure 13. SHM framework instantiation: model for altimeter health.



H_L	Θ_{H_L}
healthy	0.7
bad	0.3

U_A	Θ_{U_A}
inc	0.5
dec	0.5

H_B	Θ_{H_B}
healthy	0.9
bad	0.1

U_A	H_L	S_L	Θ_{S_L}
inc	healthy	inc	1.0
		dec	0.0
	bad	inc	0.5
		dec	0.5
dec	healthy	inc	0.0
		dec	1.0
	bad	inc	0.5
		dec	0.5

U_A	S_S	Θ_{S_S}
inc	inc	0.7
	dec	0.1
	maybe	0.2
dec	inc	0.1
	dec	0.7
	maybe	0.2

U_A	H_B	S_B	Θ_{S_B}
inc	healthy	inc	1.0
		dec	0.0
	bad	inc	0.5
		dec	0.5
dec	healthy	inc	0.0
		dec	1.0
	bad	inc	0.5
		dec	0.5

Figure 14. Bayesian network and CPT tables for reasoning about altimeter failure.

The CPT tables for the sensors are read as follows: if the (latent) status U_A is increasing and the laser altimeter is healthy, then the probability that it is reading an increasing value is 1; no decreasing measurement is reported ($p = 0$). In the case of a failing laser altimeter, no result can be obtained; hence $p = 0.5$. The same model is used for the barometric altitude. The IMU sensors are modeled somewhat differently. If they report an upward velocity, it is likely ($p = 0.7$) that this has been caused by an upward movement of the UAS ($U_A = inc$). Due to high sensor and integration noise, the results are not unique and $U_A = maybe$ indicates a value within a zero-centered deadband. Figure 15 breaks down how we evaluate this Bayesian network and how our architecture automatically detected a temporary outage of the laser altimeter.

With our current implementation of the μBayes unit and a configuration as shown in Figure 13, running at a system clock frequency of 115 MHz, the unit is able to evaluate the Altimeter Health Model 660 times per second.

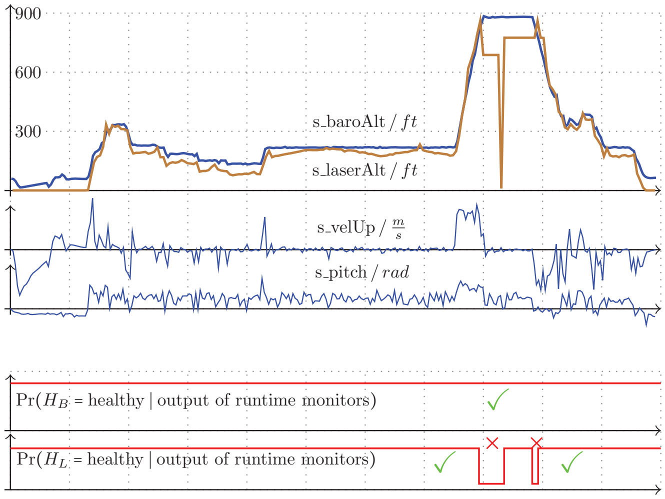


Figure 15. Flight data collected from the Swift UAS (top three panels) and output of our Bayesian SHM model, given as probabilities (bottom two panels).

8.3. Reasoning about Software

In principle, SHM models for software components are structured in a similar way to those for sensor monitoring. Signals are extracted from a communications bus between components, from specific memory locations using shared variables, or from the operating system. No specific instrumentation of the safety-critical control code is necessary. Compared to hardware and sensor management, the complexity of software health models is usually higher, because of the often substantial functionality of the code including the existence of modes. Furthermore, substantial reasoning can be required, because individual failures (due to dormant software bugs or problematic hardware-software interaction) might pervade large portions of the software system and can cause seemingly unrelated failures in other components. Such a situation occurred when a group of six F-22 Raptors was first deployed to the Kadena Air Base in Okinawa, Japan (Johnson, 2007). When crossing the international dateline (180° longitude), a dormant software bug caused multiple computer crashes. Not only was the navigation completely lost, but also the seemingly unrelated communications computer crashed. “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” (Johnson, 2007).

We now consider how such an unfortunate interplay between software design and poor implementation could cause adverse effects on the flight hardware. Figure 16 shows a mock-up of a flawed architecture for a flight-control computer. In this architecture, all components, like GN&C, the drivers for the aircraft sensors and actuators, as well as payload components including a science camera and the transmitter for the video stream, communicate via a message queue. The message queue is fast enough to push through all messages at the required speed. However, for debugging and logging purposes, all message headers are written (in blocking mode) into an on-board file system. A corresponding requirement appears as an example flight rule in Section 5:

$$\Box((\mathrm{addToQueue}_{\mathrm{GN\&C}} \wedge \Diamond\mathrm{removeFromQueue}_{\mathrm{Swift}}) \rightarrow \neg\mathrm{removeFromQueue}_{\mathrm{Swift}}\, \mathcal{U}\, \mathrm{writeToFS}).$$

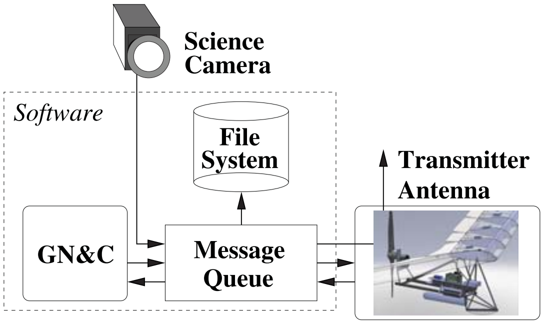


Figure 16. Flawed system architecture for file system-related scenario.

This architecture works perfectly when the system is started and the file system is empty or near empty. However, after some time of operation, as the file system becomes increasingly populated (but writes can still occur), sudden aircraft oscillations, similar to pilot-induced-oscillations (PIO), take place. No software error whatsoever (e.g., overfull file system or overfull message queue) is reported and the situation worsens if the science camera, which also uses this message queue, is in operation.

The underlying root cause is that writes into the file system take an increasing amount of time as the file system fills up (due to long searches for free blocks). This situation accounts for longer delays in the message queue, which cause delays in the seemingly unrelated control loop, ultimately causing oscillations of the entire UAS. For a software health model, therefore, non-trivial reasoning is important, because such failures can manifest themselves in seemingly unrelated components of the aircraft.

Table 5 and Figure 17 show details of our model. All signals except the barometric altitude are extracted from the operating system running on the flight computer. In the diagram in Figure 17, discrete signals are fed directly into the Bayesian networks; continuous signals like the length of the message queue or the amount of data in the file system are categorized to enable discretization into threshold blocks, e.g., the file system is empty, filled to more than 50%, filled to more than 90%, or full. The barometric altitude is fed through a Fast Fourier Transform (FFT) in order to obtain the frequency

Table 5. Signals and their intended meanings.

Signal name	Description
s_FS_Error	error in file system
s_W_FS	writing into file system
s_FS	"df" of file system
s_Queue_lng	length of message queue
s_baroAlt	barometric altitude
s_delta_q	dynamic queue behavior (derived)
s_osc	UAS oscillation (derived)

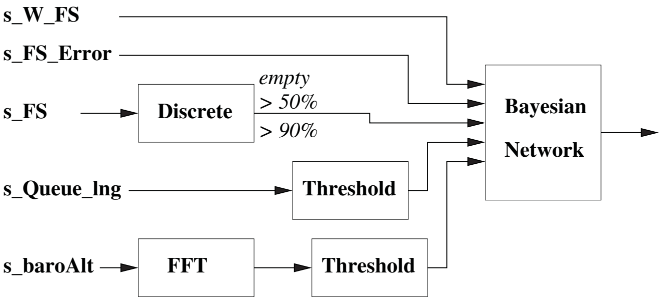


Figure 17. Structure of the SHM model for the file system scenario.

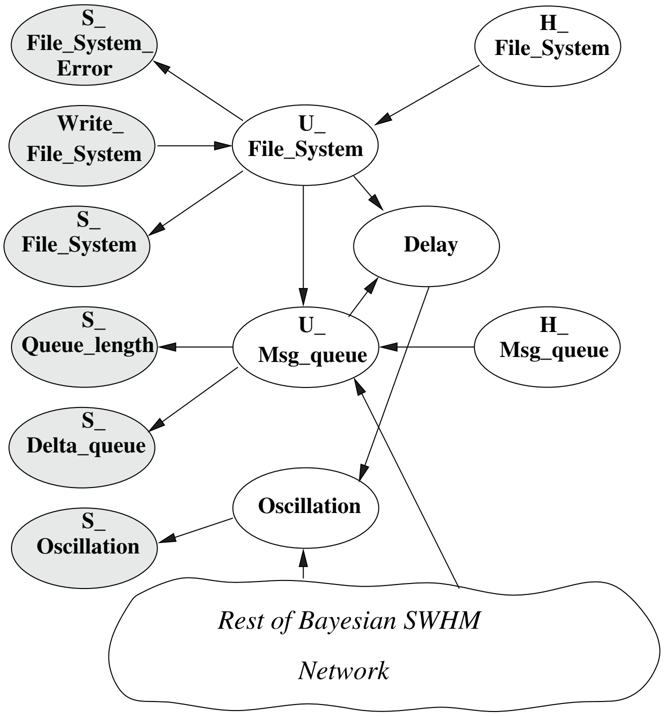


Figure 18. Relevant nodes from Bayesian system health model for oscillation detection.

spectrum. Again, a threshold block is used to determine if amplitudes are above a certain threshold indicating oscillation (low frequency) or strong vibrations (higher frequency).

Figure 18 shows the relevant excerpt from our Bayesian SHM model for this scenario, including the file system and the message queue. The software-related sensor nodes for this model are located on the left-hand side of the network: a sensor to detect writes to the file system and a sensor providing information on storage capacity in the file system (with states: *empty*, *medium*, *almost- full*, and *full*). Similarly, *S_Queue_length* provides information about the length of the message queue. Finally, *S_Delta_queue* senses whether the length of the message queue is increasing or decreasing. Nodes for the internal status of components, such as the file system and the message queue, are connected via sensor and health nodes. The behavior nodes for system oscillation and delay build the foundation for reasoning about this and similar scenarios.

Figure 19 shows the temporal traces of a file system-induced fault scenario (Schumann, Morris, Mbaya, Mengshoel, & Darwiche, 2011) in simulation. The flight controller's pitch-up and pitch-down commands to the actuators (top panel) are impacted by faults originating from the file system, causing the aircraft to oscillate up and down rather than maintaining the desired altitude. For the purpose of this experiment, we set the file system to almost full at the start of the simulation run; no other faults or errors occur. After about 30 seconds, the delays caused by the message queue have accumulated in such a way that flight-control induced oscillations of the aircraft occur, indicated by recurrent climbs and descends (middle panel). Eventually, these altitude oscillations are detected and picked up by the Fast Fourier Transform, and a signal is sent to *S_Oscillation*. The bottom panel of Figure 19 shows the posteriors for selected health nodes. It indicates that the actual aircraft sensors and actuators are healthy. However, the health status of the software (blue) decreases substantially a little after 100 seconds, indicating a problem in the on-board software. In this scenario, the health of the file system and of the message queue, when considered individually, do not drop significantly. Also, the software itself does not flag any error.

9. Conclusions

We presented a three-pronged approach to sensor and software health management in real time, on-board a UAS. Health models are constructed in a modular and scalable manner using a number of different building blocks. Major block types provide advanced capabilities of temporal logic runtime monitors, model-based analysis and signal processing, and powerful probabilistic reasoning using Bayesian networks. For our overarching design requirements of unobtrusiveness, responsiveness, and realizability, we automatically transform the health model into efficient FPGA hardware designs. We demonstrated the capabilities of this approach on a set of

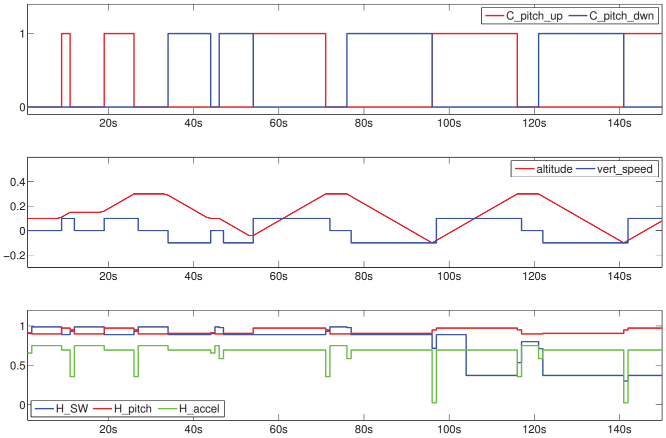


Figure 19. Traces of simulation experiment with file-system related failure scenario. Top panel: actuator messages sent through the message queue. Middle panel: vertical speed and altitude of the aircraft showing oscillations. Bottom panel: posterior probabilities of selected health nodes.

requirements and flight rules, both for sensor and software health management. We presented experimental results for this approach using actual data from the NASA Swift UAS.

Our approach enables the designer to build complex models and reasoning modes. For example, temporal reasoning over the results of probabilistic health outputs can be formulated easily: $(alt < 1000\text{ft}) \rightarrow \Diamond_{[0,10s]}(P(\text{H\_laserAlt} = \text{healthy}) > 0.8)$ would require a working laser altimeter at altitudes of less than 1000ft. In a similar manner, results of prognostics components can be smoothly integrated into our framework.

However, the results shown here are only the first steps towards a real-time on-board sensor and software health management system. For the proof of concept demonstration in this paper, we analyzed recorded data streams from the Swift UAS on the ground as if they were happening in real time. There are two clear options for reading this data on-board the Swift UAS instead: reading sensor data passed on the common bus or having sensor data sent to our framework by the flight computer. In the near future, we plan to define and build unobtrusive read-only interfaces that will enable us to get real-time sensor and software data from the common bus or flight computer while providing the guarantee that under no circumstances would our framework disturb the bus or any other UAS component. This is a major requirement for certification and carrying out actual flight tests on the Swift UAS.

On a broader level, research needs to be performed on how to automatically generate advanced system health management models from requirements, designs, and architectural artifacts. In particular for monitoring the health of a complex and large software system, automatic model generation is essential. We are confident that our approach, which allows us to combine monitoring of sensors, prognostics, and software while separating out (model-based) signal processing, temporal, and probabilistic reasoning will substantially facilitate the development of improved and powerful on-board health management systems for unmanned aerial systems.

Acknowledgments

This work was in part supported by NASA NRA grant NNX08AY50A "ISWHM: Tools and Techniques for Software and System Health Management." The work of Thomas Reinbacher was supported within the FIT-IT project CevTes managed by the Austrian Research Agency FFG under grant 825891.

References

Alur, R., & Henzinger, T. A. (1990). Real-time Logics: Complexity and Expressiveness. In *Lics* (pp. 390–401). IEEE Computer Society Press.

Barringer, H., et al. (Eds.). (2010). *Runtime verification - first international conference, rv 2010, proceedings* (Vol. 6418). Springer Verlag.

Basin, D., Klaedtke, F., Müller, S., & Pfitzmann, B. (2008). Runtime monitoring of metric first-order temporal properties. In *Fsttcs* (pp. 49–60).

Basin, D., Klaedtke, F., & Zălinescu, E. (2011). Algorithms for monitoring real-time properties. In *Rv* (Vol. 7186, pp. 260–275). Springer Verlag.

Bauer, A., Leucker, M., & Schallhart, C. (2010). Comparing LTL semantics for runtime verification. *J. Log. and Comput.*, *20*(3), 651–674.

Bekkerman, R., Bilenko, M., & Langford, J. (Eds.). (2011). *Scaling up machine learning: Parallel and distributed approaches*. Cambridge University Press.

Bellman, R. (1958). On a routing problem. *Quarterly of Applied Mathematics*, *16*, 87–90.

Bolton, M., & Bass, E. (2013). Evaluating human-human communication protocols with miscommunication generation and model checking. In *Nasa formal methods symposium* (Vol. TBD). Springer.

Brown, R., & Hwang, P. (1997). *Introduction to random signals and applied kalman filtering* (3rd ed.). John Wiley & Sons.

Chavira, M., & Darwiche, A. (2005). Compiling Bayesian networks with local structure. In *Proceedings of the 19th international joint conference on artificial intelligence (ijcai)* (pp. 1306–1312).

Chavira, M., & Darwiche, A. (2007). Compiling Bayesian networks using variable elimination. In *Proc. of the twentieth international joint conference on artificial intelligence (ijcai-07)* (pp. 2443–2449). Hyderabad, India.

Darwiche, A. (2001). Recursive conditioning. *Artificial Intelligence*, *126*(1-2), 5-41.

Darwiche, A. (2003). A differential approach to inference in Bayesian networks. *Journal of the ACM*, *50*(3), 280–305.

Darwiche, A. (2009). *Modeling and reasoning with bayesian* *networks*. Cambridge, UK: Cambridge University Press.

Dawid, A. P. (1992). Applications of a general propagation algorithm for probabilistic expert systems. *Statistics and Computing*, *2*, 25–36.

Divakaran, S., D'Souza, D., & Mohan, M. R. (2010). Conflict-tolerant real-time specifications in Metric Temporal Logic. In *Time* (p. 35-42). IEEE Computer Society Press.

Federal Aviation Administration. (2013). *Federal Aviation Regulation §91.*

Gan, X., Dubrovin, J., & Heljanko, K. (2011). A symbolic model checking approach to verifying satellite onboard software. *ECEASST*, *46*.

Geilen, M. (2003). An Improved On-The-Fly Tableau Construction for a Real-Time Temporal Logic. In *Cav* (Vol. 2725, pp. 394–406). Springer Verlag.

Havelund, K. (2008). Runtime verification of C programs. In *Testcom/fates* (pp. 7–22). Springer Verlag.

Huang, C., & Darwiche, A. (1994). Inference in belief networks: A procedural guide. *International Journal of Approximate Reasoning*, *15*(3), 225-263.

Huang, J., Chavira, M., & Darwiche, A. (2006). Solving MAP exactly by searching on compiled arithmetic circuits. In *Proc. of the twenty-first national conference on artificial intelligence* (p. 143-148). Boston, MA.

Ippolito, C., Espinosa, P., & Weston, A. (2010, April). Swift UAS: An electric UAS research platform for green aviation at NASA Ames Research Center. In *CAFE eas iv.*

Jensen, F. V., Lauritzen, S. L., & Olesen, K. G. (1990). Bayesian updating in causal probabilistic networks by local computations. *SIAM Journal on Computing*, *4*, 269–282.

Jeon, H., Xia, Y., & Prasanna, V. K. (2010). Parallel exact inference on a CPU-GPGPU heterogenous system. In *Proc. of the 39th international conference on parallel processing* (p. 61-70).

Johnson, D. (2007). *Raptors Arrive at Kadena.* Retrieved from http://www.af.mil/news/story.asp?storyID=123041567

Kask, K., Dechter, R., & Gelfand, A. (2010). BEEM: bucket elimination with external memory. In *Proc. of the 26th annual conference on uncertainty in artificial intelligence (uai-10)* (pp. 268–276).

Kozlov, A. V., & Singh, J. P. (1994). A parallel Lauritzen-Spiegelhalter algorithm for probabilistic inference. In *Proc. of the 1994 acm/ieee conference on supercomputing* (pp. 320–329).

Kulesza, Z., & Tylman, W. (2006). Implementation of Bayesian network in FPGA circuit. In *Mixdes* (p. 711 -715). IEEE Computer Society Press.

Lauritzen, S. L., & Spiegelhalter, D. J. (1988). Local computations with probabilities on graphical structures and their application to expert systems. *Journal of the Royal Statistical Society*, *50*(2), 157–224.

Li, Z., & D'Ambrosio, B. (1994). Efficient inference in Bayes nets as a combinatorial optimization problem. *International Journal of Approximate Reasoning*, *11*(1), 55–81.

Lichtenstein, O., Pnueli, A., & Zuck, L. (1985). The glory of the past. In *Logics of programs* (Vol. 193, p. 196-218). Springer Verlag.

Lin, M., Lebedev, I., & Wawrzynek, J. (2010). High-throughput Bayesian computing machine with reconfigurable hardware. In *Fpga* (pp. 73–82). ACM Press.

Linderman, M. D., Bruggner, R., Athalye, V., Meng, T. H., Asadi, N. B., & Nolan, G. P. (2010). High-throughput Bayesian network learning using heterogeneous multicore computers. In *Proc. of the 24th acm international conference on supercomputing* (p. 95-104).

Lindsey, A. E., & Pecheur, C. (2004). Simulation-based verification of autonomous controllers via livingstone pathfinder. In K. Jensen & A. Podelski (Eds.), *Proceedings tacas 2004* (Vol. 2988, pp. 357–371). Springer.

Low, Y., Gonzalez, J., Kyrola, A., Bickson, D., Guestrin, C., & Hellerstein, J. (2010). GraphLab: A new framework for parallel machine learning. In *Proc. of the 26th annual conference on uncertainty in artificial intelligence (uai-10)* (p. 340-349).

Maler, O., Nickovic, D., & Pnueli, A. (2005). Real time temporal logic: Past, present, future. In *Formats* (Vol. 3829, p. 2-16). Springer Verlag.

Maler, O., Nickovic, D., & Pnueli, A. (2007). On synthesizing controllers from bounded-response properties. In *Cav* (Vol. 4590, p. 95-107). Springer Verlag.

Maler, O., Nickovic, D., & Pnueli, A. (2008). Checking temporal properties of discrete, timed and continuous behaviors. In *Pillars of comp. science* (p. 475-505). Springer Verlag.

Mengshoel, O. J. (2007). 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.

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

Mengshoel, O. J., Darwiche, A., Cascio, K., Chavira, M., Poll, S., & Uckun, S. (2008). Diagnosing faults in electrical power systems of spacecraft and aircraft. In *Proc. of the twentieth innovative applications of artificial intelligence conference (iaai-08)* (pp. 1699–1705). Chicago, IL.

Mengshoel, O. J., Roth, D., & Wilkins, D. C. (2011). Portfolios in stochastic local search: Efficiently computing most probable explanations in Bayesian networks. *Journal of Automated Reasoning*, *46*(2), 103–160.

Mengshoel, O. J., Wilkins, D. C., & Roth, D. (2011). Initialization and restart in stochastic local search: Computing a most probable explanation in Bayesian networks. *IEEE Transactions on Knowledge and Data Engineering*.

Musliner, D., Hendler, J., Agrawala, A. K., Durfee, E., Strosnider, J. K., & Paul, C. J. (1995, January). The challenges of real-time AI. *IEEE Computer*, *28*, 58–66. Retrieved from citeseer.comp.nus.edu.sg/article/musliner95challenges.html

Namasivayam, V. K., & Prasanna, V. K. (2006). Scalable parallel implementation of exact inference in Bayesian networks. In *Proc. of the 12th international conference on parallel and distributed system* (p. 143-150).

Park, J. D., & Darwiche, A. (2004). Complexity results and approximation strategies for MAP explanations. *Journal of Artificial Intelligence Research (JAIR)*, *21*, 101-133.

Pasricha, R., & Sharma, S. (2009). An FPGA-based design of fixed-point Kalman filter. *ICGST International Journal* *on Digital Signal Processing, DSP*, *9*, 1–9.

Pearl, J. (1988). *Probabilistic reasoning in intelligent systems: Networks of plausible inference*. San Mateo, CA: Morgan Kaufmann.

Pike, L., Niller, S., & Wegmann, N. (2011). Runtime verification for ultra-critical systems. In *Rv* (Vol. 7186, pp. 310–324). Springer Verlag.

Poll, S., Patterson-Hine, A., Camisa, J., Garcia, D., Hall, D., Lee, C., ... Koutsoukos, X. (2007). Advanced diagnostics and prognostics testbed. In *Proc. of the 18th international workshop on principles of diagnosis (dx-07)* (pp. 178–185). Nashville, TN.

Reinbacher, T., Rozier, K. Y., & Schumann, J. (2013). Temporal-logic based runtime observers for system health mangement of real-time systems.. (under submission)

Ricks, B. W., & Mengshoel, O. J. (2009a). The diagnostic challenge competition: Probabilistic techniques for fault diagnosis in electrical power systems. In *Proc. of the 20th international workshop on principles of diagnosis (dx-09)* (pp. 415–422). Stockholm, Sweden.

Ricks, B. W., & Mengshoel, O. J. (2009b). 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.

Ricks, B. W., & Mengshoel, O. J. (2010). Diagnosing intermittent and persistent faults using static Bayesian networks. In *Proc. of the 21st international workshop on principles of diagnosis (dx-10).* Portland, OR.

Ristic, B., Arulampalam, S., & Gordon, N. (2004). *Beyond the Kalman Filter: Particle Filters for Tracking Applications*. Artech House.

Rozier, K. Y. (2011). Linear temporal logic symbolic model checking. *Computer Science Review*, *5*(2), 163–203.

Rozier, K. Y., & Vardi, M. Y. (2010, March). LTL satisfiability checking. *International Journal on Software Tools for Technology Transfer (STTT)*, *12*(2), 123 - 137. Retrieved from http://dx.doi.org/10.1007/s10009 -010-0140-3 doi: DOI10.1007/s10009-010-0140-3

RTCA. (2012). *DO-178C/ED-12C: Software considerations in airborne systems and equipment certification.* Retrieved from http://www.rtca.org

Schumann, J., Mbaya, T., Mengshoel, O. J., Pipatsrisawat, K., Srivastava, A., Choi, A., & Darwiche, A. (2013). Software health management with Bayesian networks. *Innovations in Systems and Software Engineering*, *9*(2), 1-22.

Schumann, J., Mengshoel, O. J., & Mbaya, T. (2011). Integrated software and sensor health management for small spacecraft. In *Proc. of the 2011 ieee fourth international conference on space mission challenges for information technology* (pp. 77–84).

Schumann, J., Morris, R., Mbaya, T., Mengshoel, O. J., & Darwiche, A. (2011). *Report on Bayesian approach for dynamic monitoring of software quality and integration with advanced IVHM engine for ISWHM* (Tech. Rep.). USRA/RIACS.

Shenoy, P. P. (1989). A valuation-based language for expert systems. *International Journal of Approximate Reasoning*, *3*(5), 383 – 411.

Silberstein, M., Schuster, A., Geiger, D., Patney, A., & Owens, J. D. (2008). Efficient computation of sum-products on GPUs through software-managed cache. In *Proc. of the 22nd acm international conference on supercomputing* (pp. 309–318).

Thati, P., & Roşu, G. (2005). Monitoring algorithms for Metric Temporal Logic specifications. *ENTCS*, *113*, 145–162.

Whittle, J., & Schumann, J. (2004, December). Automating the implementation of Kalman filter algorithms. *ACM Transactions on Mathematical Software*, *30*(4), 434–453.

Xia, Y., & Prasanna, V. K. (2007). Node level primitives for parallel exact inference. In *Proc. of the 19th international symposium on computer architecture and high performance computing* (p. 221-228).

Zhang, N. L., & Poole, D. (1996). Exploiting causal independence in Bayesian network inference. *Journal of Artificial Intelligence Research*, *5*, 301-328. Retrieved from citeseer.nj.nec.com/article/zhang96exploiting.html

Zhao, Y., & Rozier, K. Y. (2012). Formal specification and verification of a coordination protocol for an automated air traffic control system. In *Proceedings of the 12th international workshop on automated verification of critical systems (avocs 2012)* (Vol. 53). European Association of Software Science and Technology.

Zheng, L., & Mengshoel, O. J. (2013, August). Optimizing parallel belief propagation in junction trees using regression. In *Proc. of 19th ACM SIGKDD conference on knowledge discovery and data mining (kdd-13).* Chicago, IL.

Zheng, L., Mengshoel, O. J., & Chong, J. (2011). Belief propagation by message passing in junction trees: Computing each message faster using gpu parallelization. In *Proc. of the 27th conference in uncertainty in artificial intelligence (uai-11).* Barcelona, Spain.