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] *University of Cincinnati, Cincinnati, OH 45221, USA*
Kristin.Y.Rozier@uc.edu

[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

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

ABSTRACT

For unmanned aerial systems (UAS) to be successfully deployed and integrated within the national airspace, it is imperative that they possess the capability to effectively complete their missions without compromising the safety of other aircraft, as well as persons and property on the ground. This necessity creates a natural requirement for UAS that can respond to uncertain environmental conditions and emergent failures in real-time, with robustness and resilience close enough to those of manned systems. We introduce a system that meets this requirement with the design of a real-time onboard system health management (SHM) capability to continuously monitor sensors, software, and hardware components. This system can detect and diagnose 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 software signals; (2) signal analysis, preprocessing, and advanced on-the-fly temporal and Bayesian probabilistic fault diagnosis; and (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. We call this approach rt-R2U2, a name derived from its requirements. 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 flight data from the NASA Swift UAS.

1. INTRODUCTION

Modern unmanned aerial systems (UAS) are highly complex pieces of machinery, combining mechanical and electrical subsystems with complex software systems and controls, such as the autopilot and payload systems (e.g., cameras or scientific instruments). 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 preflight 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 system health capability must be able to detect and diagnose the failure(s) and respond in a timely manner, possibly by triggering mitigation actions. These corrective actions can range from a simple

mode change to following a pre-programmed flight path to continue the mission (in case of minor problems, such as a lost communications link) to a "limp" home. In case of severe problems, a controlled emergency landing in a remote and safe area might be necessary.

Most current UAS systems, however, only have very rudimentary fault detection systems. For example, the open-source ArduPilot software[1] only performs rudimentary sanity checks of sensor data and received commands. Faults that manifest themselves with a more complex failure pattern can often not be detected or diagnosed. Also, 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.

There is a definite 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. These decisions should maximize capabilities to meet mission objectives while maintaining safety requirements and avoiding safety hazards. Although careful system design and pre-deployment verification and validation (V&V) can be very effective in minimizing sensor failures and bugs in on-board software, it is in practice impossible to eliminate all problem sources and software bugs due to the size and complexity of the software as well as unanticipated, and therefore unmodeled, environmental conditions. The need to catch fault scenarios not detected by pre-deployment V&V is perhaps 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 as required by DO-178C (RTCA, 2012) for manned commercial aircraft.

In this paper, we describe a novel framework called rt-R2U2 that enables the design and realization of a powerful, real-time, on-board system (including sensor and software) health management (SHM) system that can (a) dynamically monitor a multitude of sensor and software signals; (b) perform substantial reasoning for fault diagnosis and prognosis; and (c) avoid interfering with the original flight software or impede on scarce on-board computing resources—the original (and certified) behavior of the UAS flight software must not be affected.

In this paper, we design a capability that enables SHM models for complex and large systems to be specified in a concise and modular manner. To this end, we have developed a three-pronged approach that combines the capabilities of temporal logic runtime observers, model-based analysis, and powerful probabilistic reasoning with Bayesian networks (BNs).

In general, any fault detection and diagnosis system uses *abstracted* models of the usually complex system to be monitored in order to decide if the system is operating in a nominal mode or if any faults are occurring. A large number of different SHM modeling paradigms with different levels of abstraction and expressive power have been developed. They exhibit very different model expressiveness and complexity and might require vastly different computational resources for their execution. For an overview, see Section 2. SHM design must find a balance between expressive power, level of abstraction, and required resources.

Figure 1 shows a high-level representation of the design space, spanned by the three major abstraction dimensions. Several well-known paradigms and diagnostic/monitoring systems are mapped into this figure and described in Section 2. The coarsest abstraction for detection and diagnosis uses a set of Boolean conditions, for example, safety requirements that are continuously monitored throughout the UAS mission. Typically, software checks performed using if-then-else statements use this kind of abstraction. The incorporation of model-specific information can substantially increase the detection and diagnosis performance. For example, the commercial system QSI/TEAMS[2] uses hierarchical, multi-signal diagnosis, where information about the system structure and signal types are incorporated. On the other end of the spectrum, systems like HyDE[3] simulate simplified dynamic systems during diagnosis. Although potentially very powerful, such approaches need substantial computational resources, which makes real-time processing on-board a resource-constrained system, like a UAS, difficult, given the current state of the art. Probabilistic information about the components' reliability or the likelihood of certain conditions can be important for the disambiguation of diagnosis results. Bayesian network-based diagnosis systems can represent such information and belong to the corresponding category. Finally, temporal issues can be of extreme importance. Only when properties like fault propagation and other temporal relationships can be expressed can large sets of realistic faults can be described, detected, and diagnosed. Examples here include fault-propagation models (like FACT/TFPG[4]) and detection or monitoring mechanisms that are based upon temporal logic.

In our rt-R2U2 framework, we combine model-based, temporal, and probabilistic approaches in a modular and hierarchical manner. We obtain high expressiveness, yet a clear separation between, for example, temporal and probabilistic aspects. This makes it possible to develop SHM models in a compact and concise manner.

[1] https://github.com/diydrones/ardupilot

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

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

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

In this paper, we discuss in detail the three major building blocks of our rt-R2U2 approach and describe a novel method to implement such a health management system on a Field Programmable Gate Array (FPGA) for efficient processing and minimal intrusiveness. We demonstrate in detail how to instrument NASA's Swift UAS with this new SHM capability.

This paper is based upon (Schumann, Rozier, et al., 2013) and has been substantially extended and improved. In particular, we refined the framework for the construction of hierarchical SHM models, placed rt-R2U2 into the abstraction space of SHM approaches, and discussed advantages of using MTL and LTL for specifying health models. We furthermore elaborated on the description of our FPGA hardware implementation and improved and extended the presentation of our examples.

After discussing related approaches in Section 2, we introduce in Section 3 our problem domain, 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 rt-R2U2 framework and present an overview of its building blocks. In the subsequent sections, we give further details of the major components of rt-R2U2, namely observers 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 9. Finally, Section 10 discusses future work and concludes.

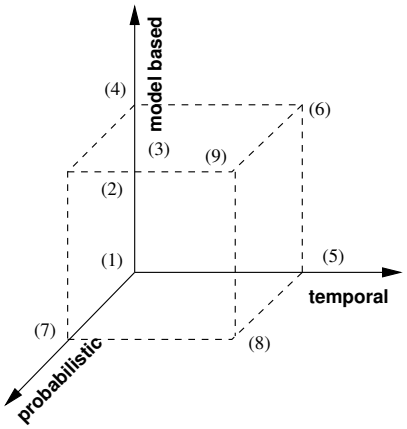


Figure 1. Abstraction space for SHM along the dimensions of temporal, model-based, and probabilistic reasoning. In this figure, (1) corresponds to Boolean conditions, (2) to paradigms similar to QSi/TEAMS, (3) to Livingstone, (4) to HyDe, (5) to temporal logic, (6) to FACT/TFPG. (7) to (static) Bayesian networks (BN), (8) to dynamic BN, and (9) to our rt-R2U2 framework. For further related work see Section 2.

2. RELATED WORK

2.1. System Health Management

System, or 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, QSi/TEAMS 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 is a Temporal Fault Propagation Graph (TFPG) with temporal constraints. More complex diagnosis systems like HyDE 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 propositional 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 (Pearl, 1988; Darwiche, 2009; Koller & Friedman, 2009; Choi, Darwiche, Zheng, & Mengshoel, 2011). Design-time knowledge about component reliability can, for example, be expressed in terms of mean time to failure (MTTF) and cleanly incorporated as priors. Whereas there are several 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, 2014). ProADAPT uses compilation of Bayesian networks into arithmetic circuits (Darwiche, 2003; 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 (Ricks & Mengshoel, 2014). This work also shows how Bayesian system health models can be generated automatically from schematics of electrical power systems such as ADAPT (Mengshoel et al., 2008, 2010).

[5] http://ti.arc.nasa.gov/opensource/projects/livingstone2/
[6] http://reasoning.cs.ucla.edu/samiam/
[7] http://www.hugin.com/

2.2. Software Health Management

With increasing complexity and size of software in safety-critical systems, the burden of pre-deployment verification and validation is extremely high. Despite all efforts, dormant bugs in the software or operation in unforeseen environments can cause catastrophic software and subsequent system failures. (Srivastava & Schumann, 2013) give an introduction into software health management and discuss crucial properties and requirements. (Zhao & Rozier, 2014b) utilizes probabilistic model checking to compare flight software designs at system design time to maximize system health management during deployment. Specific approaches here include (Pike, Goodloe, Morisset, & Niller, 2010; Luo et al., 2014; Huang et al., 2014).

2.3. Runtime Verification

Existing methods for Runtime Verification (RV) (Qadeer & Tasiran, 2012; Legay & Bensalem, 2013; Bonakdarpour & Smolka, 2014) assess system status by automatically generating (mainly software-based) observers to check the state of the software system against a formal specification. Observations in RV are usually made accessible via software instrumentation (Havelund, 2008); they usually report only when a specification has failed, e.g., through adding hooks in the code base to detect changes in the state of the system being monitored. Such instrumentation may unfortunately make recertification of the system necessary, alter the original timing behavior, or increase resource consumption (Pike, Niller, & Wegmann, 2011); we seek to avoid these problems. Also, reporting only the outcomes of specifications does not provide the real-time responsiveness we require for rt-R2U2, because no diagnostic information is provided.

Systems in our applications domain often need to adhere to timing-related flight rules like this one: *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), such as Metric Temporal Logic (MTL) (Alur & Henzinger, 1990). They can be generated specifically for runtime verification or carried over from the design phase. For example, Zhao and Rozier (2012, 2014a) generated LTL specifications in the design phase of an air traffic control system that could be easily carried throughout the system development process, as could the LTL specifications that check human-human communication protocols for air transportation (Bolton & Bass, 2013). LTL formulas have also been used to verify the design of an embedded satellite software control system (Gan, Dubrovin, & Heljanko, 2011).

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 were, however, designed with a complex software implementation in mind and require powerful computers that would far exceed the limits of a small UAS.

2.4. 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; 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 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 (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. GPUs also have power consumption benefits compared to CPUs, which is another important factor for a UAS.

Other authors have used the benefits of a hardware architecture to natively answer statistical queries on BNs. For example, (Lin, Lebedev, & 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 & 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.

(Watterson & Heffernan, 2007) review established and emerging approaches for monitoring software executions of embedded systems. They call for future work on runtime verification approaches that utilize existing chip interfaces to provide the observations as events to an external monitoring system. (Pike et al., 2010) worked on runtime verification for real-time systems by defining observers in a data-flow language, which are compiled into programs with constant runtime and memory. If the original system is periodically schedulable with some safety margin, the monitored system can be shown to be schedulable, too. This approach targets software only, whereas we monitor a combination of embedded software and hardware components. Hardware observers that simply probe one or more internal signals have been known in the literature for a few decades. An early instance thereof is the non-interference monitoring and replay mechanism by (Tsai, Fang, Chen, & Bi, 1990). Their monitoring system is based on the MC6800 processor that records the execution history of the target system. A dedicated replay controller then replays stored executions, which supports test engineers in low-level debugging. Although we share a similar idea of probing internal signals, rt-R2U2 detects specification violations on-the-fly, rather than replaying traces from some execution history.

The Dynamic Implementation Verification Architecture (DIVA) exploits runtime verification at an intra-processor level (Austin, 1999). Whenever a DIVA-based microprocessor executes an instruction, the operands and the results are sent to a checker that verifies correctness of the computation; the checker also supports fixing an erroneous operation. (Chenard, 2011) gives a system-level approach to debugging based on in-silicon hardware checkers.

Work of Brörkens and Möller (2002) akin to ours also does not rely on code instrumentation to generate event sequences. Their framework, however, targets Java and connects to the bytecode using the Java Debug Interface (JDI) so as to generate sequences of events.

BusMOP (Pellizzoni, Meredith, Caccamo, & Rosu, 2008) generates observers for past time LTL on FPGAs, which are connected to the Peripheral Component Interconnect (PCI). The commercial Temporal Rover system (Drusinsky, 2003) implements observers for Metric Temporal Logic (MTL) formulas, but the implementation and algorithms used are not published.

The concept of a separate hardware processor to monitor the behavior of the main control computer for safety is, for example, being used in the automotive industry. Barr (2013) discusses the engine control module of a Toyota vehicle, which has a dedicated separate processor for monitoring some safety and health requirements during operation.

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 UAS was designed to support NASA's research interests in aeronautics and earth science, with particular focus on autonomy, intelligent flight control, and green aviation. For safe operation, the UAS must meet a large number of requirements derived from NASA and FAA processes and standards. In this section, 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). The Swift UAS has a full-featured flight computer and a payload control computer for sensor payloads. The individual components are connected via a common bus interface running the Reflection Architecture, which provides a component-based plug-and-play infrastructure. Typical sensors include barometric altitude sensors, airspeed indicator, GPS, and a laser altimeter to measure the altitude above ground. Figure 3 shows the high-level schematics of the Swift flight computer and sensor system.

Figure 2. The Swift all-electric UAS (right) shown with other UAS and some of the authors.

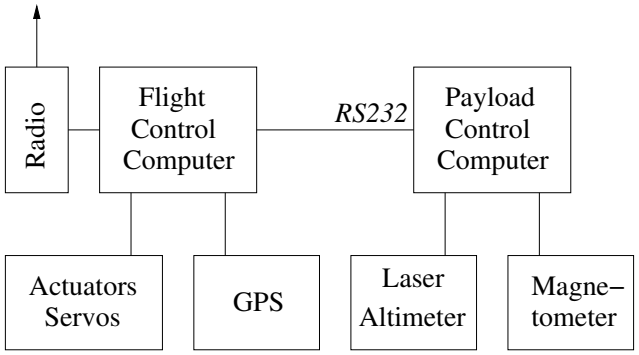


Figure 3. High-level schematics for the Swift flight electronics. Data between the flight control computer and the payload computer are exchanged using a serial RS232 connection.

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 rate checks, e.g., the maximal safe climb (or descent) rate or the maximum rate of change for some sensor value. For safe operation, the values must always stay within certain bounds. Other examples include range checks including the operating windows for sensors and common-sense bounds like that we cannot measure a rate of descent when we are below the minimum range for such a measurement, i.e., parked on the ground. 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. On the software side, value checks can include the size of the current call stack or lengths of message buffers.

Relationships encode dependencies among sensor or software 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 and rt-R2U2 needs to find the most likely root cause, e.g., a broken Pitot tube sensor, a dangerous flight maneuver, or a dead engine.

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 are rules that must be obeyed for mission- or system-specific reasons. 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 in Section 5 how these requirements and flight rules can be specified in rt-R2U2 and how they can be translated into efficient hardware.

4. THE RT-R2U2 SYSTEM HEALTH MANAGEMENT FRAMEWORK

Our modeling framework for system (including 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 from which rt-R2U2 gets its name before we focus on the description of the design framework. Each of the rt-R2U2 framework's three prongs will then be described in detail in the subsequent sections; observers using temporal logic in Section 5, model-based monitors in Section 6, and Bayesian reasoning components in Section 7.

4.1. Design Requirements

For autonomous systems running with severely constrained computing 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 should 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. This paper focuses on the detection and reasoning part; a follow-on mitigation process is left for future work. See Section 10 for a discussion.

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 compo-

nents without requiring significant reconfiguration or additional components. The framework must be reconfigurable so that health models can be updated without a lengthy recompilation 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 existing Swift UAS computing architecture, and 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., changes in the SHM model necessitate a redesign of the framework's hardware.[8] We create a realizable framework that avoids pushing system constraints by running on-board isolated but previously flight-certified, integrated hardware. Our framework is designed to uphold all of these requirements, thus we call it the **r**eal-**t**ime, **R**ealizable, **R**esponsive, **U**nobtrusive **U**nit (rt-R2U2).

4.2. Design Framework

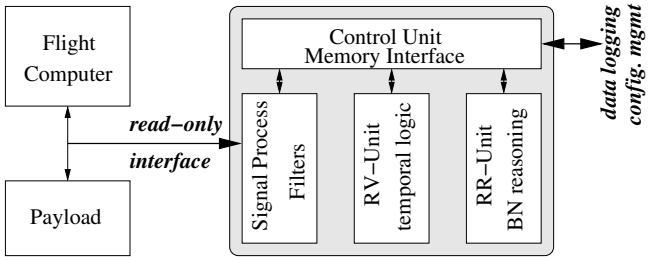


Figure 4. Principled architecture of rt-R2U2 with read-only interfaces to flight computer and communication bus.

4.2.1. Overview

Figure 4 shows the high-level schematics of our FPGA-based framework and its connection to the Swift flight hardware. The FPGA monitor obtains the sensor and software data from the flight computer and sensor systems using a hardware-based read-only communications channel. For example, an RS-232 connection with two wires (RxD and Gnd) can be used; optical isolation can protect the flight hardware from any electrical interference. That way, it can be guaranteed that the operation of the flight computer will not be disturbed under any circumstances; an important part of our UNOBTRUSIVENESS requirement. The implementation architecture of rt-R2U2 on the FPGA is straightforward: incoming sensor and software signals are preprocessed, filtered, and made ready for processing by the RV-Unit (Runtime Verification), which performs temporal reasoning using MTL. The RR-unit (Runtime Reasoning) contains a reasoning engine for Bayesian diagnostic models. Data transfer and overall control is performed by a specialized control unit. In Section 8, we discuss the architecture in more detail and describe the special-purpose execution engines for the RV- and RR-units.

The health models in rt-R2U2 are hierarchical, graphical models; see Figure 5. Inputs to the models consist of signals carrying sensor values or values of software variables of the system to be monitored. Outputs can be results of the evaluation of formulas in temporal logic or probability values, indicating the health of a component or subsystem. The signals connecting the processing blocks are of different types as shown in Table 1. The width, in bits, of each signal type depends on the concrete FPGA realization and the size of the FPGA. The numbers here correspond to our current implementation, which is detailed in Section 8 and (Geist, Rozier, & Schumann, 2014). Sensor signals $\mathbb{S}$ carry analog sensor readings, represented as, in our case, 18-bit fixed-point values. In a similar way, we represent probabilities $\mathbb{P}$. Discrete values $\mathbb{D}$ are found, e.g., after discretization of sensor signals and can have the values $0 \ldots 15$. Time-stamps $\mathbb{T}$ correspond to ticks of the global clock and also have a width of 18 bits, Finally, $\mathbb{B}$ (*true*, *false*) and $\mathbb{B}_+$ (*true*, *false*, *maybe*) carry the results of temporal logic operators.

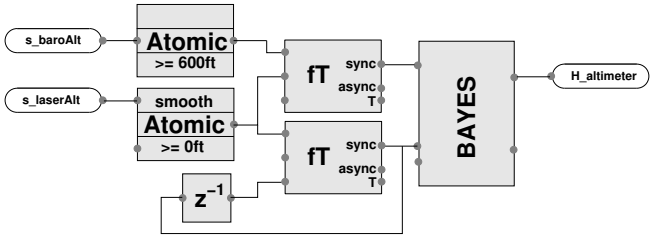


Figure 5. An rt-R2U2 configuration block diagram: two sensor signals are read in, processed and discretized by atomic blocks. The signals are then subjected to various temporal formulas before they are fed into a Bayesian reasoner.

Table 1. Data types for signals in rt-R2U2.

Type	Width (bits)	Description
$\mathbb{S}$	18	analog sensor reading
$\mathbb{P}$	18	probability
$\mathbb{D}$	4	discrete value $0 \ldots 15$
$\mathbb{T}$	18	time stamp
$\mathbb{B}$	1	Boolean
$\mathbb{B}_+$	2	*true*, *false*, *maybe*

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

Table 2. Typical SHM building blocks; $n, m \in \mathbb{N}$.

Signal processing		
A_1	$\mathbb{S} \to \mathbb{B}$	atomic preprocessor
A_2	$\mathbb{S}^2 \to \mathbb{B}$	atomic preprocessor
FLT	$\mathbb{S} \to \mathbb{S}$	smoothing filter
$RATE$	$\mathbb{S} \to \mathbb{S}$	rate filter
FFT	$\mathbb{S} \to \mathbb{S}^n$	FFT
KF	$\mathbb{S}^n \to \langle \mathbb{S}^n, \mathbb{S}^m \rangle$	Kalman filter
Temporal logic		
pT	$\mathbb{B}^n \to \mathbb{B}$	past time observer
fT_s	$\mathbb{B}^n \to \mathbb{B}_+$	synchronous fT observer
fT_a	$\mathbb{B}^n \to \langle \mathbb{B}, \mathbb{T} \rangle$	asynchronous fT observer
Bayesian reasoning		
BN	$\mathbb{D}^n \to \mathbb{P}^m$	discrete Bayesian reasoner
Miscellaneous		
Ev	$\mathbb{B}_+ \to \mathbb{D}$	observer discretization
$\max_i$	$\mathbb{P}^n \to \langle \mathbb{D}, \mathbb{P} \rangle$	find maximum value and index
$arg\max_B$	$\mathbb{P}^n \to \mathbb{B}^n$	select maximum marginal
$\min_i$	$\mathbb{P}^n \to \langle \mathbb{D}, \mathbb{P} \rangle$	find minimum value and index
z^{-1}	$\mathbb{X} \to \mathbb{X}$	unit delay for discrete type $\mathbb{X}$
SPL	$\mathbb{X} \to \mathbb{X}$	rate conversion/subsampling

4.2.2. Model Building Blocks

Table 2 shows an overview of the available modeling blocks. New types of blocks can be added to our framework. The most prominent and flexible blocks for processing of sensor and software signals are the unary and binary atomic blocks. Each discretizer block can process one or two signals s_1, s_2 according to $(\pm 2^{p_1} \times F_1^2(F_1^1(s_1)) \pm 2^{p_2} \times F_2^2(F_2^1(s_2))) \bowtie c$ for integer scaling constants p_1, p_2, comparison constant c, filters F_j^i, and a comparison operator $\bowtie \in \{=, <, \leq, \geq, >, \neq\}$. With this basic architecture, multiple discretization operators can be defined. For example, the detection of a smoothed positive climb rate given the altitude alt could be done by $lp(alt_t - alt_{t-1}) > 5$, where lp is a low-pass filter. In this case, our atomic block would be instantiated with $p_1 = 0$, F_1^1 a rate filter, $F_1^2 = lp$ the smoothing filter, $\bowtie$ = " > ", and $c = 5$. In this case, p_2, F_2^2, F_2^1 and s_2 are not used. This method would be used to specify, for example, the condition when the measured GPS altitude is larger than the barometric altitude: $alt_{GPS} > alt_{baro}$.

Additional blocks can specify special purpose processors like a Fast Fourier Transform, Kalman Filters, or potentially a model-based prognostics block, which is part of future work. These model-based processing units are discussed in Section 6. Temporal logic blocks execute observers for variations of past time and future time linear temporal logic to provide efficient evaluation of signals over time; see Section 5. Probabilistic reasoning and Bayesian diagnosis is performed using a discrete Bayesian reasoner block. It receives, as evidence, multiple discrete values, e.g., results from temporal processing or discretized signals, and estimates posterior marginals that indicate the health of the system or components. Additional blocks are defined to process the output of the Bayesian reasoner such that its results can be used by temporal reasoning. The reasoner and its hardware implementation are described in detail in Section 7.

4.2.3. Execution Mechanism

The execution of an SHM model is performed in several steps that are governed by the FPGA implementation of our framework (Geist et al., 2014). New sensor data is obtained through a read-only interface at each time stamp. An internal time base provides these time stamps. Our system can also use an external time base, e.g., derived from the GPS system, to issue new time stamps. In a first step, these analog signals are processed using the signal processing and atomic blocks. Then, in a second step, temporal logic expressions are evaluated using the current values of the discretized signals. In a third step, evidence is set in a Bayesian Network and the marginal posterior probabilities carrying the health information are computed. In a fourth and final step, results are transported to the output. The communication between the different blocks is accomplished through memory blocks on the FPGA.

Complex models can require multiple processing cycles. For example, signals going from the BN reasoning block "back" to the temporal logic processor require a delay block in order to avoid cyclic dependencies; see the example in Fig. 6 below.

4.2.4. Examples

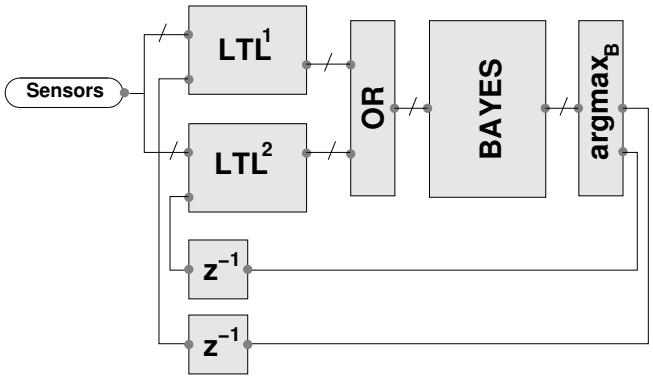


Figure 6. Example of an rt-R2U2 block diagram: depending on the reasoning outcome, different sets of temporal specifications (LTL^1 vs. LTL^2) are used for monitoring.

Figure 6 shows an example where the behavior of the SHM changes with the most likely system state as determined by the health model. In nominal mode (according to the SHM reasoner output), a set of temporal rules (LTL^1) is being used for monitoring. Those rules can contain past-time and future time components. If an anomaly is detected by the Bayesian network, different health nodes will become active. Then, the specifications LTL^2 will be used. LTL^2 may encode a different, reduced, set of mission goals and perhaps relaxed requirements. The $argmax_B$ block finds the position of the maximum of the posterior marginals as produced

by the Bayesian network and generates a Boolean signal to select the appropriate set of temporal rules. Because the evaluation of the selected rules happens a timestamp later, delay blocks denoted z^{-1} are used.

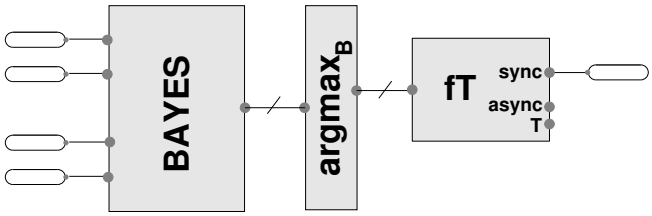


Figure 7. Example of an excerpt of an rt-R2U2 configuration block diagram: diagnostic results are arguments of LTL formulas.

With a model as shown Figure 7, requirements like "a diagnosed altimeter sensor problem should last at most 5 seconds," or "the health of the battery should not drop more than three times within 120 seconds," can be modeled. Here, the output of the Bayesian network is analyzed by an $arg\max_B$ block, which sets a Boolean value if a root cause is identified (e.g., an altimeter sensor or battery problem).

5. MONITORING OF TEMPORAL SENSOR DATA USING TEMPORAL LOGIC

5.1. Linear and Metric 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 Linear Temporal Logic (LTL) (Pnueli, 1977), which allows the expression of requirements over timelines and also pairs naturally with the original English expression of the requirements.[9] For requirements that express specific time bounds, we use a variant of LTL that adds these time bounds, called Metric Temporal Logic (MTL) (Alur & Henzinger, 1990). We can automatically generate runtime observers for requirements expressed in these logics, thereby enabling real-time analysis of sensor data as well as system health assessment. LTL formulas consist of:

1. Variables representing system state: we include variables representing the data streaming from each sensor onboard 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, for example, ALWAYS, EVENTUALLY, NEXTTIME, UNTIL, and RELEASE. Given Boolean variables p, q, we define the temporal operators as:

- 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 on-board 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, at the same time that q is 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 $\neg q$.

In 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 a 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.

Figure 8 gives pictorial representations of how these logics (mission-bounded LTL and MTL) enable the precise specification of temporal safety requirements in terms of timelines.

5.2. Examples of System Requirements in TL

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, 2014a; Gan et al., 2011; Bolton & Bass, 2013; Alur & Henzinger, 1990). Recall the example system safety requirements from Section 3.2. In a straightforward manner, we can encode each of the value checks (V), relationship requirements (R), and flight rules (F) as temporal

[9] 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.

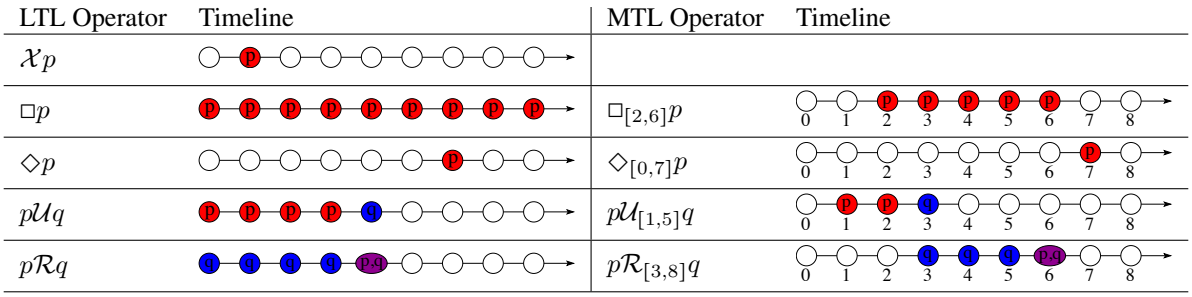


Figure 8. Pictorial representation of LTL temporal operators (left) and MTL operators (right). For a formal definition of LTL, see for example (Rozier, 2011); for MTL, see for example (Alur & Henzinger, 1990).

logic formulas to enable runtime monitoring, as demonstrated by the following examples:[10]

V1: the maximal safe climb and descent rate V_z of the Swift UAS is limited by its design and engine characteristics:

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

For a compact notation, we allow relational expressions inside the formula. Within our framework, the relational expressions are evaluated by atomic blocks returning Boolean values that then comprise the input to the temporal observers.

V2: the maximal angle of attack α is limited by Swift design characteristics:

$$\Box(\alpha \leq 15^\circ)$$

V3: the Swift roll (p), pitch (q), and yaw rates (r) are limited to remain below maximum bounds for safe operation:

$$\Box\left(p < 0.99\frac{\text{rad}}{\text{s}} \wedge q < 4.0\frac{\text{rad}}{\text{s}} \wedge r < 2.2\frac{\text{rad}}{\text{s}}\right)$$

V4: the battery voltage U_{batt} and 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:

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

R1: pitching up, i.e., increasing the angle of attack α from its default value α_0, 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 the Swift starts to 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 descent can also be defined:

$$\Box(\Box_{[0,20s]}((\alpha < \alpha_0) \vee (I_{eng} < 10A)) \rightarrow \Diamond_{[0,2s]}V_z < 0)$$

R2: 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 greater than that of the runway alt_0:

$$\Box((alt > alt_0) \rightarrow (V_{\text{IAS}} > V_S))$$

Here, we assume that the altitude of the runway is always lower than that of the flying aircraft. In a scenario where the airfield is on a hill and the UAS is flying through a nearby valley, **R2** would not be valid and would need to refined. This is an example demonstrating that the definition of concise and correct requirements is far from trivial.

R3: the sensor readings for the vertical velocity V_z and the barometric altimeter alt_b are correlated, because V_z corresponds to changes in altitude. This means that whenever the vertical speed is positive, we should measure within 2 seconds that the increase of altitude, Δ_{alt_b} is larger than 5ft/s or 300ft/min. In order to avoid triggering that rule by very short pulses of positive V_z, a positive V_z must be measured for at least 5 consecutive seconds:

$$\Box(\Box_{[0,5s]}(V_z > 0) \rightarrow \Diamond_{[0,2s]}(\Delta_{alt_b} > 300\frac{\text{ft}}{\text{min}}))$$

R4: the precision of the position reading P_{GPS} from the GPS subsystem depends on the number of visible GPS satellites

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

N_{sat}, i.e., it must be larger than a given precision threshold P^i_{GPS} for i satellites:

$$\begin{array}{ll} \Box(& \\ \Box(N_{sat} == 1) & \rightarrow P_{GPS} \geq P^1_{GPS} \quad \wedge \\ \Box(N_{sat} == 2) & \rightarrow P_{GPS} \geq P^2_{GPS} \quad \wedge \\ \Box(N_{sat} == 3) & \rightarrow P_{GPS} \geq P^3_{GPS} \quad \wedge \\ \Box(N_{sat} \geq 4) & \rightarrow P_{GPS} \geq P^+_{GPS}) \end{array}$$

F1: after receiving a command (cmd) for takeoff, the Swift UAS must reach an altitude of 600ft within 40 seconds:

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

F2: after receiving the landing command, touchdown needs to take place within 40 seconds, unless the link (lnk) is lost. The status of the link is denoted by s_{lnk}. In a lost-link situation, the aircraft should reach a loitering altitude around 425ft within 20 seconds. Landing is indicated by returning to runway altitude (alt_0) and turning off power to the engine (I_{eng}):

$$\begin{array}{l} \Box((\text{cmd} == \text{landing}) \rightarrow \\ \quad ((s_{lnk} == \text{ok}) \rightarrow \Diamond_{[0,40s]}((alt_0 - 1\text{ft} \leq alt \leq alt_0 + 1\text{ft}) \\ \qquad\qquad \wedge (I_{eng} < 1\text{A})) \vee \\ \quad (s_{lnk} == \text{lost}) \rightarrow \Diamond_{[0,20s]}(400\text{ft} \leq alt \leq 450\text{ft}))) \end{array}$$

F3: 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* be a Boolean variable that is true if the UAS crosses the boundary between the small subdivision of the airspace in which the UAS is currently located and another subdivision. After receiving the loiter command, the UAS should stay in the same sector, at an altitude between 400ft and 450ft until it receives a landing command. The UAS has 30 seconds to reach loitering position:

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

F4: 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:

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

F5: a working laser altimeter should be available if the barometric altitude of the Swift is less than 1000ft. This flight rule requires reasoning about the health of a component itself and can be expressed as:

$$\Box((s\_baroAlt < 1000\text{ft}) \rightarrow (p(\text{H\_laserAlt} = \text{healthy}) > 0.8))$$

Here, we require that the health of the laser altimeter is at least 80% when flying at low altitudes. In our framework, the output of the BN is, after discretization, fed back into the temporal observer block in a configuration similar to Figure 6.

5.3. Advantages of Temporal Logic Requirements.

Encoding the set of system requirements in temporal logic offers several significant advantages. Temporal logic can be used to produce a very precise, unambiguous specification of requirements and this set of specifications can be carried throughout the entire process from initial system requirements, through system design, through testing, and finally to runtime monitoring. The use of temporal logic enables the use of automated tools and processes, such as automated requirements debugging (i.e., satisfiability checking (Rozier & Vardi, 2010)) and design-time V&V techniques such as model checking (Rozier, 2011). We note that writing large formulas in temporal logic can be tricky, as with writing good requirements in general; specification debugging is a required step in system development.

Our rt-R2U2 framework simultaneously handles past-time and future-time temporal logics; we include evaluation of past-time formulas since this is easier than the future-time implementation required for our system. Past-time LTL or MTL might be convenient to express specific requirements. Furthermore, automatic evaluation of strictly past-time formulas is easier since we have already seen all of the data needed to complete the evaluation. Although such temporal relationships could be formalized using Dynamic Bayesian Networks (DBNs), reasoning over larger time-spans would require extremely large networks whereas we can evaluate temporal logic observers very efficiently. Alternatively, reducing the DBN network size would diminish accuracy substantially.

5.4. Monitoring Approach

From each temporal logic requirement, we automatically generate two kinds of observers, which we call *synchronous* and *asynchronous* observers. This dual-observer construction operates in parallel to provide real-time system health updates. Our synchronous runtime observer outputs a verdict (`true`, `false`, or `maybe`) at every tick of the system clock. This is important because it provides blocks such as the Bayesian reasoner with better real-time information and therefore improves diagnostic and prognostic capabilities by enabling monitoring input to be considered by the reasoner. If the verdict is `true` or `false`, the synchronous observer was able to determine the result of the satisfaction check of the temporal

logic requirement immediately. If the verdict is `maybe` the result of the satisfaction check depends on an event that will happen at a future point in time; our asynchronous observer will refine the preliminary `maybe` verdict of the satisfaction check into either `true` or `false`.

An *asynchronous observer* provides the final outcome of the requirement at an a priori known time. An asynchronous observer reports if a requirement is satisfied or fails earlier as soon as this can be known or else yields the final result of the requirement when its time bound has elapsed. For details on the construction of these observers, and formal proofs that our constructions are correct, see (Reinbacher, Rozier, & Schumann, 2014).

The availability of dual observers is a key element of our rt-R2U2 framework, because it enables our runtime observers 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 observers provide much more useful output. Such output can, for example, 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 $(\Diamond_{[3,2005]}\ p)$ and p occurs at time 5 it is important to utilize this information for real-time calculations of system health. Traditional runtime monitoring techniques do not yield any output in this case, either at time 5 or 2005, since no property failure occurred.

6. MODEL-BASED MONITORING OF TEMPORAL SENSOR DATA

Highly accurate and detailed information about system health could be obtained if the actual system were compared in real time against 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 rt-R2U2, 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. In our rt-R2U2 framework, we can define Kalman filters that can be directly mapped to 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); (Ye & Zhang, 2009) describe an FPGA implementation.

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 9.3), or to detect unexpected software behavior, like a reboot loop.

All model-based components primarily interact with atomic blocks, which discretize their outputs to ready the signals for processing by the temporal and probabilistic model components. 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; see Section 10 for a more detailed discussion.

7. BAYESIAN HEALTH MANAGEMENT REASONING

We use a Bayesian network (BN) 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 the edges in the DAG. Figure 9 is a simple example of a BN. 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 a BN is associated with a corresponding conditional probability table (CPT) that typically captures its causal relationship with parents and children in the DAG. It should be noted that BNs are not necessarily causal (see (Pearl, 2000)), and a developer is free to introduce non-causal edges as well.

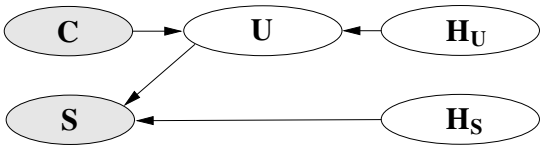


Figure 9. Simple Bayesian network.

In rt-R2U2, the BN inputs are comprised of discrete values, e.g., Boolean, three-valued outputs of synchronous observers, or discretized sensor values, 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, many different prob-

abilistic queries can be formulated, our rt-R2U2 framework aims to compute marginal posterior probabilities of selected nodes, giving an indication (probability) of sensor or software health. Thus our Bayesian reasoning components output a posteriori probabilities over system health nodes.

7.1. Bayesian Health Models

For the Bayesian models, we follow an approach that "glues together" BN fragments (Mengshoel et al., 2010; Schumann, Mengshoel, & Mbaya, 2011; Schumann, Mbaya, et al., 2013; Ricks & Mengshoel, 2014). For example, consider the BN in Figure 9. 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 a sensor node and H_U for a status node.

Command nodes C are handled as ground truth and used to indicate commands, modes, actions, or other known states. Command nodes do not have incoming edges in the network. Sensor nodes S are also input nodes, but the input signal is not assumed correct, e.g., it could result from a failed sensor or excessive noise. This behavior is modeled by connecting an S node to a health node H that reflects the health of the input to S. 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. By definition, a health node H is attached to a status node U, but not to a behavior node B.

For modeling the edges of a BN, we generally 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 ...
$\{H_U, C\} \xrightarrow{E} U$	status U, with health H_U, is controlled through command C
$\{C\} \xrightarrow{E} U$	status U is controlled through command C
$\{H_U\} \xrightarrow{E} U$	health H_U influences status U
$\{H_S, U\} \xrightarrow{E} S$	status U influences sensor S, which may fail as reflected in health H_S
$\{H_S\} \xrightarrow{E} S$	health H_S influences sensor S
$\{U\} \xrightarrow{E} S$	status U influences sensor S

Once the BN nodes and edges are in place, the BN parameters found in conditional probability tables (CPTs) need to be decided. The CPT associated with each node defines the conditional probability of a state in a node, given the states of its parent nodes. Table 4 shows two examples of CPTs for our network in Figure 9. Nodes that have no incoming edges (e.g., H_S) only contain prior probabilities for each state. In Table 4 (left) the CPT for H_S shows that the sensor S is healthy with a probability of 0.99. Table 4 (right) shows the CPT for the sensor node S, assuming it has the two states "low" and "high," and U has the states "up" and "down." Since this node has two incoming edges, it contains the conditional probabilities $p(S|U, H\_S)$. In this example, the sensor reads "low" most of the time when in the "up" mode and "high" in the "down" mode. If the sensor is broken, and in state "bad," no such relation exists as indicated by the 0.5 probabilities.

Table 4. Conditional probability tables for node H_S (left) and S (right) of the Bayesian network in Figure 9.

H_S	Θ_{H_S}
healthy	0.99
bad	0.01

U	H_S	S	Θ_S
up	healthy	low	0.8
up	healthy	high	0.2
up	bad	low	0.5
up	bad	high	0.5
down	healthy	low	0.01
down	healthy	high	0.99
down	bad	low	0.5
down	bad	high	0.5

Once the nodes, edges, and CPT parameters of a BN have been specified, it can be used for reasoning, in other words for computing outputs from inputs. Each input to a BN is, in our rt-R2U2 setting, provided by connecting an input signal to the state of a C or S node, called "clamping" or "conditioning." Note that inputs to the BN can be outputs of any block in our rt-R2U2 framework, for example, a smoothed and discretized sensor reading, the result (binary or ternary) of a temporal observer, or the output of another reasoning block. The BN outputs are further discussed in Section 7.2, after describing how computation is in fact not performed using the BN directly. Instead, it is done using a data structure, an arithmetic circuit, that the BN in compiled to, off-line.

7.2. Compilation to Arithmetic Circuits

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 (AC) evaluation (Darwiche, 2003; Chavira & Darwiche, 2007).

We select AC evaluation as the rt-R2U2 inference algorithm; we therefore compile our BN into an arithmetic circuit. In real-time avionics systems, where there is a strong need to align the resource consumption of SHM computation with resource bounds (Musliner et al., 1995; Mengshoel, 2007), al-

gorithms based upon arithmetic circuit evaluation are powerful, as they provide predictable real-time performance (Chavira & Darwiche, 2005; Mengshoel et al., 2010).

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

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

$$p(X_1, X_2, \ldots) = \prod_{\lambda_x} \lambda_x \prod_{\theta_{x|\mathbf{u}}} \theta_{x|\mathbf{u}}, \tag{1}$$

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

$$f = \sum_{\mathbf{x}} \prod_{\lambda_x} \lambda_x \prod_{\theta_{x|\mathbf{u}}} \theta_{x|\mathbf{u}} \tag{2}$$

An arithmetic circuit is a compact representation of a network polynomial (Darwiche, 2009) which, in its non-compact form, is exponential in size and thus unrealistic in the general case. Hence, answers to our SHM probabilistic queries, including marginals and most probable explanations (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

$$\Pr(x \mid \mathbf{e}) = \frac{1}{\Pr(\mathbf{e})} \cdot \frac{\partial f}{\partial \lambda_x}(\mathbf{e}), \tag{3}$$

where $\Pr(\mathbf{e})$ is the probability of the evidence $\mathbf{e}$. 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).

8. HARDWARE REALIZATION

8.1. Architecture Overview

Figure 12 shows a detailed overview of the FPGA hardware. Sensor and software signals are fed into the RV-unit (runtime verification) for signal processing and temporal reasoning. Bayesian diagnostic reasoning is performed by the RR-unit on the FPGA. All processing units are connected via a memory interface and controlled by the control interface, which is also in charge of loading the compiled specifications. The detailed architecture of RV- and RR-units is described in (Geist et al., 2014); (Reinbacher et al., 2014) focuses on the temporal reasoning algorithms and their FPGA implementation. In this paper, we discuss how the reasoning with a diagnostic BN can be implemented efficiently within our rt-R2U2 framework on an FPGA. A detailed description of the temporal reasoning algorithms and their FPGA implementation can be found in (Reinbacher et al., 2014). The core of the RR-unit is a special-purpose processing unit for reasoning with Bayesian networks, μBayes. We designed the μBayes unit in the hardware description language VHDL and use the logic-synthesis tool ALTERA QUARTUS II[11] to synthesize the design onto an Altera Cyclone IV EP4CE115 FPGA.

8.2. Hardware Realization for Reasoning Component

In our rt-R2U2 framework the BN reasoning blocks are provided with values produced by other blocks, as inputs to C and S nodes. We use these evidence values to calculate posterior marginals for the health nodes H of the Bayesian SHM model in our BN hardware implementation. Posterior marginals for this kind of BN reasoning can be evaluated in the arithmetic circuit by traversing the nodes of the circuit in a bottom-up and a subsequent top-down manner. We also make the following observations regarding the structure of arithmetic circuits:

(i) The labels of inner nodes in the arithmetic circuit alternate between addition and multiplication. 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.

8.2.1. Hardware Architecture of μBayes

The above observations led us to a hardware architecture that is centered around parallel processing units called computing blocks. A computing block, as shown in Figure 10, 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 kinds of structures found in subtrees of an arithmetic circuit. By rearranging the arithmetic circuit using commutative and associative properties, we can tile the entire AC with instances of these three modes.

These computing blocks are the building blocks of our Bayesian SHM hardware unit, which we call μBayes. Figure 11 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 ("evidence") to the BN are provided as evidence indicators and stored in a separate evidence indicator memory. An offline

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

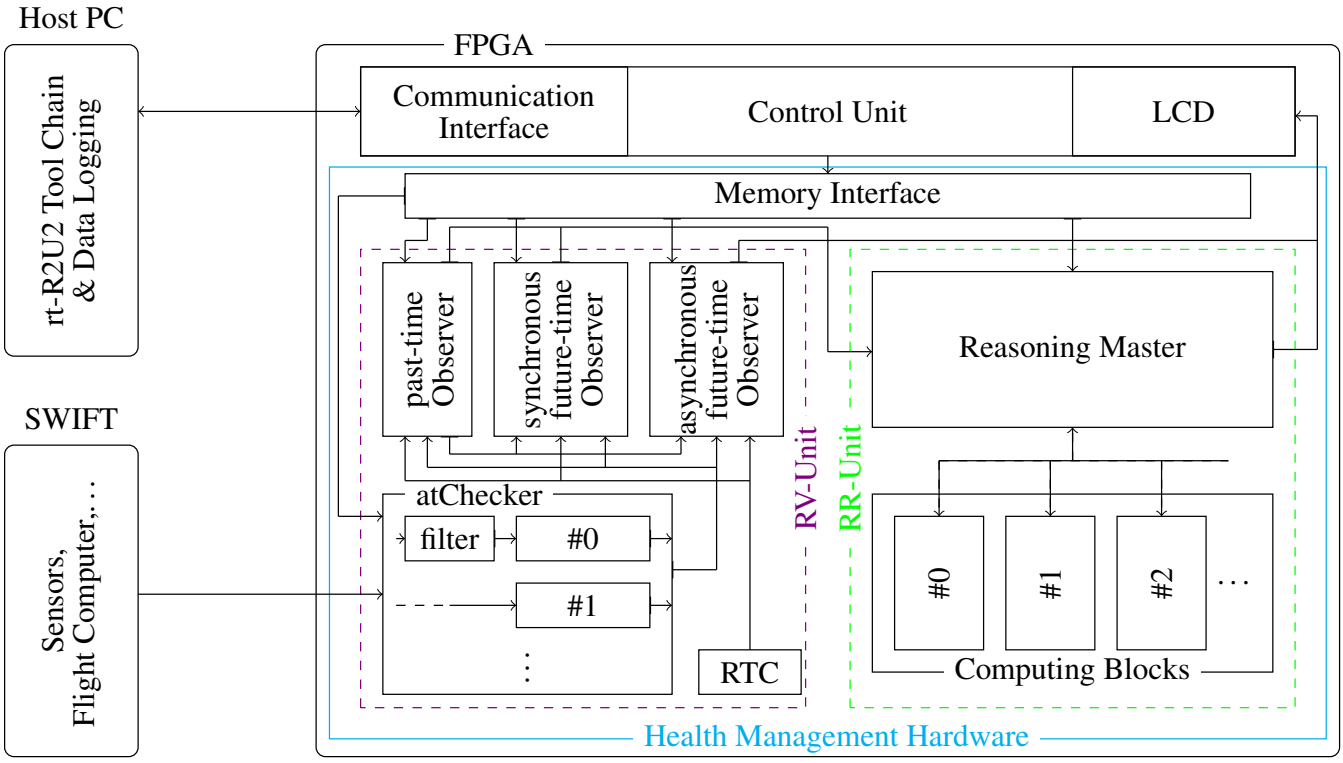


Figure 12. Overview of the rt-R2U2 hardware architecture (see Geist et al., 2014).

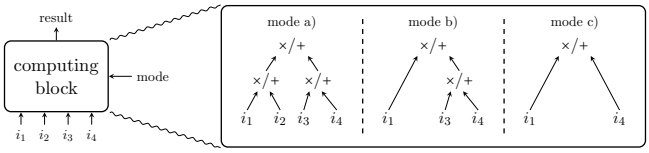


Figure 10. A computing block and its three modes of operation. Inputs to the computing block are denoted by $i_1, \ldots, i_4$.

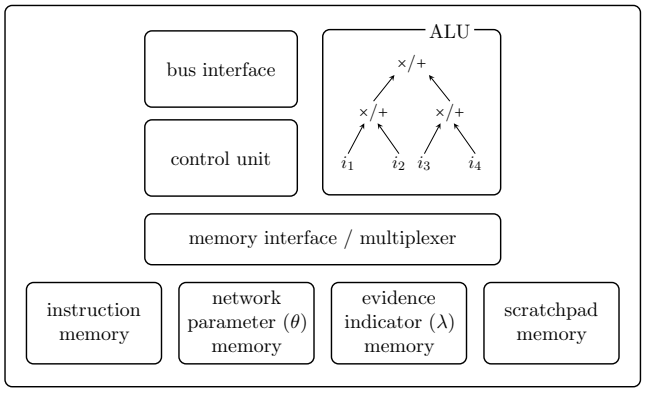


Figure 11. Internal architecture of a computing block.

compiler (see Section 8.2.2) translates the structure of the arithmetic circuit into native instructions for the μBayes unit. The control unit executes these instructions, which 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. 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.

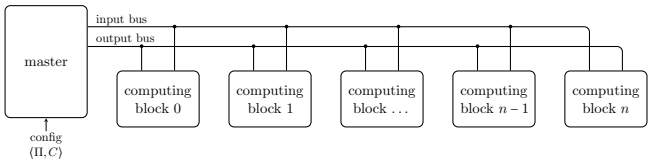


Figure 13. Architecture of the μBayes unit with parallel computing blocks. The configuration consists of the native program Π for μBayes and network parameters C.

Figure 13 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 and computes posterior marginals according to Equation 3. The inverse of the probability of the evidence, $1/\Pr(\mathbf{e})$, in this equation can be com-

puted by the master unit in parallel to the top-down traversal of the arithmetic circuit once the bottom-up traversal has been 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 calculated value of $\frac{1}{\Pr(\mathbf{e})}$.

In our implementation, we chose to represent fractional values in a fixed-point representation. This substantially reduces the hardware requirements compared to using 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 FPGAs provide several hundred of such multiplier units. Our selected FPGA provides an 18-bit hardware multiplier, yielding a resolution of 2^{-18} or approximately 10^{-6} for the representation of probabilities.

8.2.2. Synthesizing an Arithmetic Circuit into a μBayes Program

A GUI-based application (Reinbacher, 2013) (see Figure 14) on a host computer compiles an arithmetic circuit into a tuple $\langle \Pi, P \rangle$, where Π is a native program for the μBayes unit and P is a configuration for the network parameter memory, including the number of required computing blocks. The synthesis of $\langle \Pi, P \rangle$ from an arithmetic circuit involves the following steps:

(1) Parse the circuit into a DAG and use compile-time information from the Ace package[12] to relate nodes in the DAG to evidence indicators and network parameters. Assemble network parameter values according to the CPTs and add them to P. 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, generate an instruction π for each node in the arithmetic circuit that is computed by that computing block and 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.

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

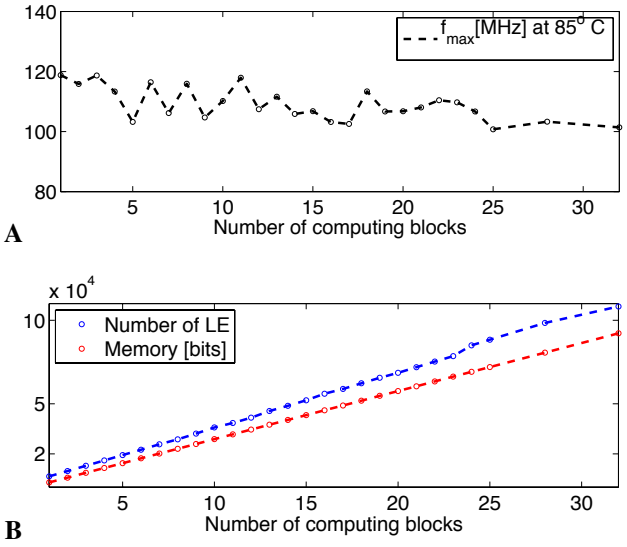


Figure 15. Logic synthesis results of our μBayes unit for an Altera Cyclone IV EP4CE115 FPGA. A: maximum operating frequency f_{max}. B: number of logic elements (LE), and required memory bits versus number of parallel computing blocks.

8.2.3. Hardware Resource Consumption

We synthesized the hardware design of the μBayes unit for various FPGAs using industrial logic synthesis tools from Altera and Xilinx. For the experiments in this paper, the tool ALTERA QUARTUS II was used. 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 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 15 shows the influence of the number of computing blocks on the maximum operating frequency, the number of logic elements, and the number of memory bits.

9. 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 observers as presented in Section 5. For all examples, actual

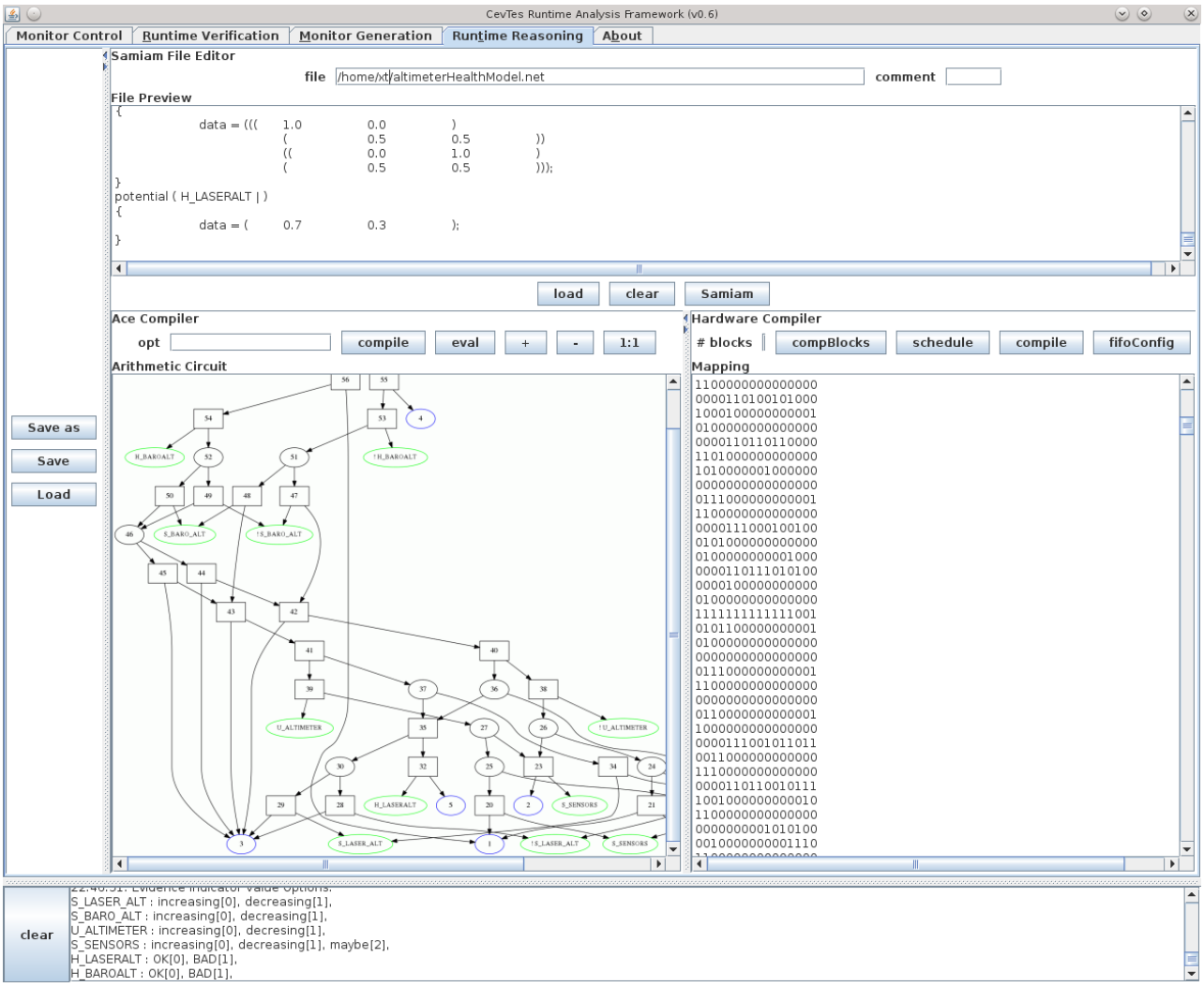


Figure 14. Screen-shot of our GUI-based synthesis tool (Reinbacher, 2013) for μBayes configurations. There is a textual description of the altimeter health model Bayesian network (top), a compiled arithmetic circuit of the network (bottom left), and a binary configuration for our μBayes unit (bottom right).

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 (Section 9.2) 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 rt-R2U2 framework can be used for reasoning about software (Section 9.3).

9.1. Monitoring of Requirements

Recall from Section 3.2 that our rt-R2U2 framework operates on a set of requirements, which are interpreted via paths through a network of building blocks to achieve our fault detection and diagnostic 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 observers in hardware, on-board FPGAs, from our temporal logic translations of the requirements (Section 5). In this way, requirements form the backbone of our rt-R2U2 framework.

Here, we exemplify the monitoring process for our temporal logic-based runtime observers, including how they take input from and pass input to other blocks in our rt-R2U2 framework (Figure 16). We demonstrate the power of generating observers from temporal logic requirements.

Consider our requirement **F1** from Section 3.2, instantiated with data from the barometric altimeter:

$$\square((\text{s\_cmd} == \text{takeoff}) \rightarrow \diamondsuit_{[0,40s]}(\text{s\_baroAlt} \geq 600\ \text{ft}))$$

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 observers that yield both a synchronous observer that updates with each tick of the system clock and an asynchronous observer that determines the satisfaction of the requirement as soon as there is enough information to do so.

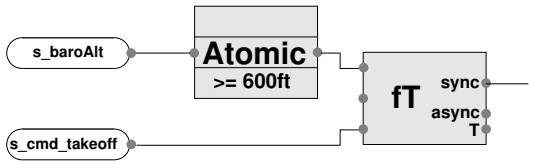


Figure 16. An rt-R2U2 configuration block diagram for monitoring the requirement $\square((\text{s\_cmd} == \text{takeoff}) \rightarrow \diamondsuit_{[0,40s]}(\text{s\_baroAlt} \geq 600\text{ ft}))$.

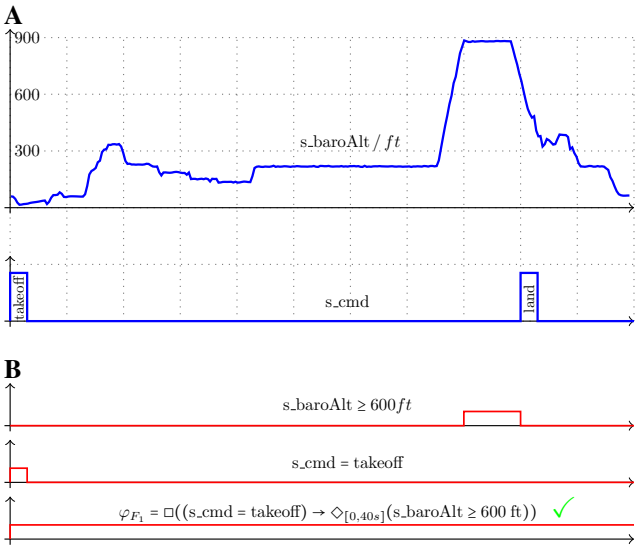


Figure 17. A: flight data recorded on-board the Swift UAS: barometric altitude (top) and commanded mode s_cmd. B: temporal traces of subformulas and results of $\square((\text{s\_cmd} == \text{takeoff}) \rightarrow \diamondsuit_{[0,40s]}(\text{s\_baroAlt} \geq 600\text{ ft}))$. The UAS reaches the top of the climb at about 30s after takeoff.

Figure 16 breaks down how we monitor this requirement. First, the raw data from the barometric altimeter are passed through one of our atomic filter blocks, as described in Section 4 and compared to the threshold of 600ft. The resulting stream of Boolean values and the command data stream are the two inputs to our pair of temporal logic observers for this requirement. Figure 17B shows the corresponding temporal traces. 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 $(\text{s\_cmd} == \text{takeoff})$. The straight red line at the bottom shows that the requirement holds at every time point during the flight. This bottom line is the output from our asynchronous observer and can be used as the input to another block in our rt-R2U2 framework, such as a Bayesian reasoning block. During UAS flight, this stream of output data will be delayed until there is enough data to determine if the temporal requirement is true. In order to better support real-time reasoning, we use the output of the paired synchronous observer, which differentiates, in real time, when we know that the flight rule holds from when we do not have enough information, at the present time, to make that decision.

Now consider requirement **R3** from Section 5.2

$$\square(\square_{[0,5s]}(V_z > 0) \rightarrow \diamondsuit_{[0,2s]}(\Delta_{\text{s\_baroAlt}} > 300\text{ft/min}))$$

stating that a significant positive vertical velocity (at least 5 consecutive seconds) needs to be followed by an increase in altitude. Figure 18 shows the rt-R2U2 configuration block diagram for this requirement.

Again, we take the raw measurement data from the barometric altimeter (s_baroAlt), pass it through one of our smoothing filter blocks to reduce the sensor noise, take the difference and compare that against the threshold of 300ft/min; the output of this atomic block corresponds to the subformula $\Delta_{\text{s\_baroAlt}} > 300\text{ft/min}$. This stream is fed as an input to our temporal observer. In Figure 19, the barometric altitude appears in the top panel. The inertial navigation unit supplies the vertical velocity V_z reading; its data stream is the second panel of Figure 19. We feed this sensor data stream through a moving average filter or smoothing; the result is shown in blue in the third panel. These data streams are then processed by components of our asynchronous runtime observer; results are shown in the bottom three panels of Figure 19. The red curve at the top, our vertical velocity observer, 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 curve in the middle, our barometric altimeter observer, flags time points that fall within a two second time interval when the change in altitude is above the given threshold. These components comprise our runtime observer, 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 19.

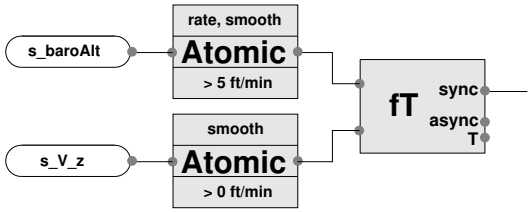


Figure 18. The rt-R2U2 configuration block diagram for monitoring requirement **R3**.

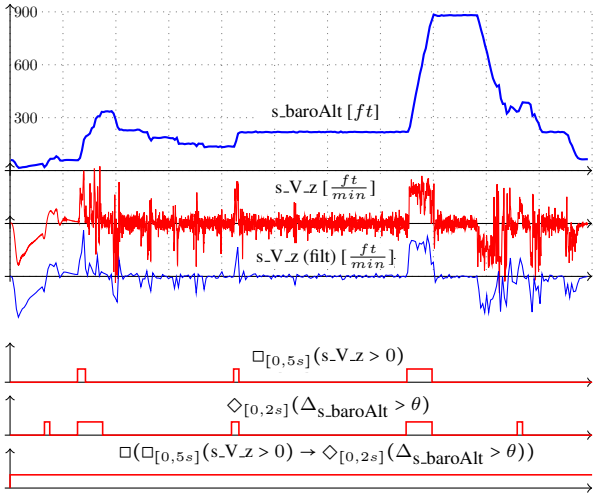


Figure 19. Raw input signals (top two panels), intermediate signals (panels 3–5), and outputs of temporal logic observers (bottom panels).

9.2. Sensor Health Management

The continuous monitoring of a UAS's flight-critical sensors is very important. Faulty, iced, or clogged Pitot tubes for measuring speed of the aircraft has, for instance, 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.[13] 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. Unfortunately, these other sources of information are also not independently reliable, thus creating a non-trivial SHM problem.

In this experiment, we integrate 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 5 lists the signals and their intended meanings. Our corresponding rt-R2U2 configuration block diagram is shown in Figure 20.

The measurements of the laser and barometric altimeter are smoothed and applied to a rate filter to obtain velocities. Then, the atomic block discretizes these rates into into increasing ($rate \geq 0$) and decreasing ($rate < 0$), before the information is fed into the reasoning component. Here, the atomic block produces two signals: *inc* and *dec*.

Figure 21 shows the BN model for reasoning about altimeter failures. Sensor nodes (inputs) for each of the different sensor types are at the bottom. The unobservable state U_A,

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

Table 5. Signals and their intended meanings.

Signal	Unit	Description
s_baroAlt	ft	altitude reading from barometric altimeter
s_laserAlt	ft	altitude reading from laser altimeter
s_V_z	ft/min	vertical velocity reading from IMU
s_pitch	rad	Euler pitch reading from IMU

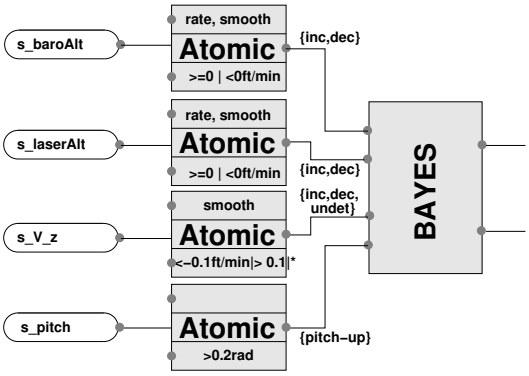
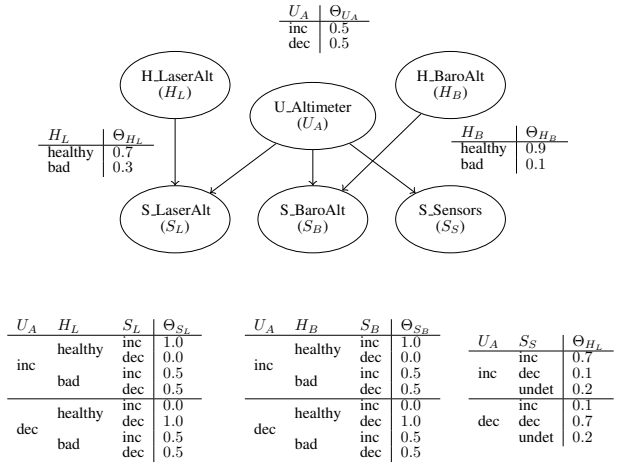


Figure 20. rt-R2U2 configuration block diagram: model for altimeter health.

describing whether the altitude of the UAS is increasing or decreasing, influences the sensor readings, hence there are edges from U_A to sensor nodes S_L, S_B, and S_S. The laser altimeter can fail. Therefore, the sensor node S_L is influenced by the node H_L, reflecting the health of the laser altimeter. A similar structure can be found for the barometric altimeter (nodes S_B and H_B, respectively). For simplicity, we did not model the health of the IMU sensors S_S, so there is no separate health node. Since no further knowledge is available on how often the UAS climbs or descends, the prior probabilities of the status node U_A are set to $p(inc) = p(dec) = 0.5$. This information is encoded in the CPTs of the health node H_U. Because the laser altimeter is prone to errors, its probability of being healthy is only 0.7, i.e., $p(\text{healthy}) = 0.7$ and $p(\text{bad}) = 1 - p(\text{healthy}) = 0.3$. This information is present in the CPT of H_L. Barometric altitude, which is measured by a more reliable sensor is defined in H_B with a prior probability of being healthy of 0.9. For details on the CPTs see also Section 7.1 above.

The CPTs for the sensor nodes shown in Figure 21 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 meaningful 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



U_A	Θ_{U_A}
inc	0.5
dec	0.5

H_L	Θ_{H_L}
healthy	0.7
bad	0.3

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	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

U_A	S_S	Θ_{H_L}
inc	inc	0.7
	dec	0.1
	undet	0.2
dec	inc	0.1
	dec	0.7
	undet	0.2

Figure 21. Bayesian network and CPT tables for reasoning about altimeter failure.

likely ($p = 0.7$) that this has been caused by an upward movement of the UAS ($U_A = inc$). Due to high noise of the IMU sensors, we introduce a dead-band such that small sensor values will not be considered for this reasoning. Those sensor values, which are below a given threshold, cause S_s = undet. Figure 22 breaks down how we evaluate this BN and how our architecture is able to detect a temporary outage of the laser altimeter. The data shown here are based upon actual Swift data, recorded during a test flight where the laser altimeter in fact failed.

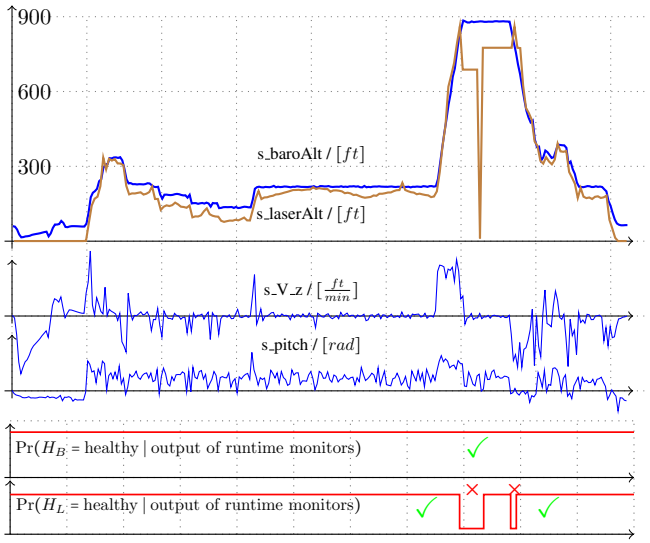


Figure 22. Flight data collected from the Swift UAS (top three panels) and output of our Bayesian SHM model, given as probabilities (bottom two panels). For this figure, a ground level of 0ft is assumed resulting in $s\_baroAlt \approx s\_laserAlt$ for healthy sensors.

With our current implementation of the μBayes unit and a configuration as shown in Figure 20, running at a system clock frequency of 115 MHz, the unit is able to evaluate the altimeter health model displayed in Figure 21 at 660Hz. We believe this is a strong result, given that an update rate of 50–200Hz is typical for control loops in aircraft.

9.3. Reasoning about Software Health

In principle, SHM models for software components are structured in a similar way to those for sensor monitoring. Signals are extracted from communication links between components, the operating system, or from specific memory locations using shared variables. If the required data are available on external buses, like on the Swift, 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 an often substantial code complexity and the multitude of operational modes. Furthermore, substantial reasoning can be required because individual failures, e.g., caused by 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 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 23 shows a mockup of a flawed architecture for a flight-control computer. This system consists of the aircraft guidance, navigation, and control (GN&C) system, the drivers for the aircraft sensors and actuators, a science camera, and the transmitter for the video stream. All components communicate via a global message queue. This message queue is, under certain conditions detailed below, fast enough to push through all messages at the required speed. For debugging and logging purposes, all message headers are written in blocking mode into an on-board file system. A corresponding requirement appears as example flight rule **F4** in Section 5:

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

This architecture works perfectly when the system is started and the file system is empty or near empty. After some time of operation, as the file system becomes increasingly populated but writes can still occur, suddenly oscillations in the alti-

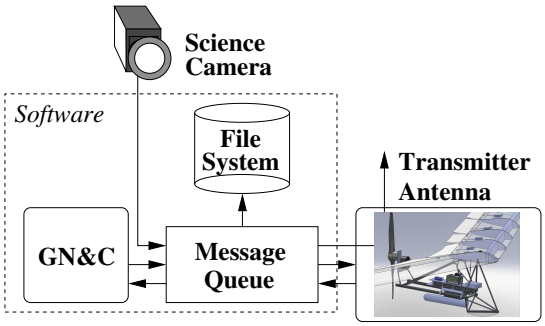


Figure 23. Flawed system architecture for file system-related scenario.

tude of the aircraft occur. These are similar to pilot-induced-oscillations (PIO). However, no software error whatsoever 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 longer searches for free blocks. This situation accounts for longer delays in the message queue, which cause delays in the seemingly unrelated inner control loop, ultimately causing oscillations of the entire UAS. For a software health model, therefore, non-trivial reasoning is important, because these kinds of failures can manifest themselves in seemingly unrelated components of the aircraft.

Table 6. Discrete signals and their intended meanings.

Signal	Unit	Description
s_FS_Error	$\mathbb{B}$	error in file system (FS)
s_W_FS	$\mathbb{B}$	writing into file system
s_FS	%	available space in FS
s_Queue_lng	$\mathbb{N}$	length of message queue
s_baroAlt	ft	barometric altitude
s_delta_q	$1/s$	dynamic queue behavior (derived)
s_osc	$\mathbb{B}$	UAS oscillation (derived)

Table 6 and Figure 24 show details of our model. All signals, except the barometric altitude signal, are extracted from the operating system running on the flight computer. In the diagram in Figure 24, discrete signals are fed directly into the Bayesian network; continuous signals like the length of the message queue or the amount of data in the file system are discretized into categories using thresholds, 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 spectrum. Here, we use a standard FFT of size 128 and pick the low-frequency band 2. A threshold of 3.5 is used to determine if low-frequency oscillations occur. This band selection also allows the health model to distinguish between oscillations with a low frequency and high-frequency vibrations.

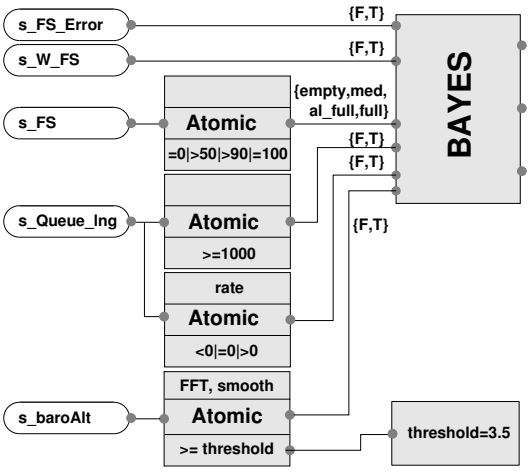


Figure 24. The rt-R2U2 configuration block diagram for the file system scenario.

Figure 25 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 side of the network, shaded in gray. File system sensors include: a sensor detecting file system errors; a sensor detecting writes; and a sensor providing information on storage capacity in the file system (with states: *empty*, *medium*, *almost full*, and *full*). Message queue sensors include: *S_Queue_length* providing information about the length of the message queue and *S_Delta_queue* sensing whether the length of the message queue is increasing or decreasing. The oscillation sensor node *S_Oscillation* connects to the output of a FFT block to detect regularly repeated variations of the vertical acceleration. 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 and status nodes for system oscillation and delay build the foundation for reasoning about this and similar scenarios.

Figure 26 shows the temporal traces of a file system-induced fault scenario in simulation. For the purpose of this experiment, we start out with a file system that is considerably full. At time stamp $t = 20$, the file system status is set to almost full. Figure 26 (top) illustrates how the oscillation in altitude ramps up. The output of the 128-element FFT filter, band 2, indicates the presence of oscillation at around $t = 100$ time stamps (Panel 2). The black line indicates the threshold of 3.5. Panel 3 shows the values of relevant discrete signals (from top to bottom): pitch-up and pitch-down commands, oscillation, file system almost full, and file system full. Note that the latter signal never becomes true.

The bottom panel shows the marginal posteriors of the health nodes *H_SW* (blue), *H_pitch* (red) and *H_accelerometer* (magenta). A clear drop of *H_SW* at the time when the oscillation is being detected indicates a problem in the on-board

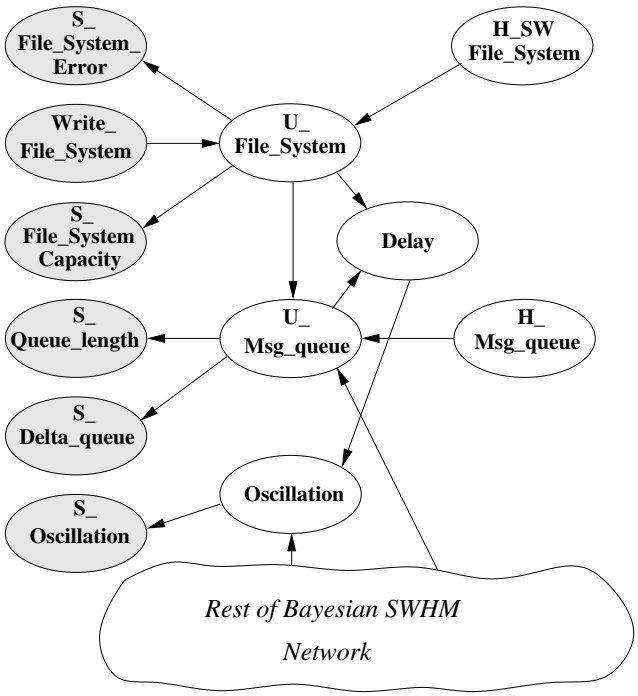


Figure 25. Relevant nodes from Bayesian system health model for oscillation detection (from Schumann, Mbaya, et al., 2013).

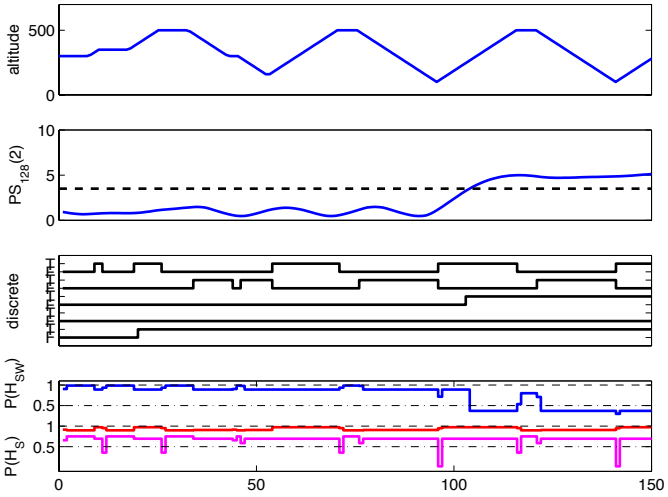


Figure 26. Traces of simulation experiment with a file-system related failure scenario (based upon (Schumann, Mbaya, et al., 2013)). Top panel: altitude profile with oscillation. Panel 2: frequency band 2 of FFT power spectrum. Panel 3: values of discrete signals (see text). Panel 4: marginal posteriors for selected health nodes.

software, whereas the aircraft sensors seem to be healthy. 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. This experiment clearly shows the capability of rt-R2U2 to isolate non-trivial software faults.

10. CONCLUSIONS AND FUTURE WORK

We have in this article presented a coordinated, extensible, 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 for temporal logic runtime observers, model-based analysis and signal processing, and powerful probabilistic reasoning using Bayesian networks. This design adheres to our overarching design requirements of UNOBTRUSIVENESS, RESPONSIVENESS, and REALIZABILITY, and we can automatically transform the resulting rt-R2U2 block diagrams of health models into efficient FPGA hardware designs. We demonstrated the capabilities of this approach on a set of requirements and flight rules, both for sensor and software health management. We presented experimental results for this approach using actual data recorded on-board the NASA Swift UAS.

However, the results shown here are only the first steps toward a real-time on-board sensor and software health management system. For the proof of concept demonstration in this article, we analyzed recorded data streams from test flights of the Swift UAS. For our analysis, we played back these data streams, simulating real-time processing. There are two clear options for processing the data on-board instead: reading sensor data passed on the common bus or having sensor data sent to our rt-R2U2 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 rt-R2U2 framework disturb the bus or any other UAS component. This is a major requirement for obtaining flight certification and carrying out actual flight tests running rt-R2U2 on the Swift UAS, which are our next goals.

Other directions for future work include specializations of rt-R2U2 for specific UAS that would maintain flight certifiability but enable us to use the outputs in real time rather than maintaining a read-only interface with the UAS. Uses for the output range from simple, conservative mitigation recommendations in the case of failed components to full-fledged autonomous decision making.

Our approach enables a designer to embed efficient SHM components that are capable of accurately capturing system complexity. The rt-R2U2 design is extensible because the building blocks comprising it can be connected in any number of ways and additional blocks could be added seamlessly if required. For example, the output of a prognostics model for monitoring on-board battery life could be used to improve accuracy of the system health model and augment diagnostic

reasoning (Schumann, Roychoudhury, & Kulkarni, 2015).

We are currently investigating possibilities for increasing automation in the process of generating each of the rt-R2U2 configuration blocks. Our tool chain for the generation of the FPGA configuration (Geist et al., 2014) can be extended to handle modular and hierarchical health models. Model-based blocks using a Kalman filter can be generated automatically from a high-level, domain-oriented specification using the AutoFilter tool (Whittle & Schumann, 2004).

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 managing the health of a complex and large software systems, 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 the NASA NRA grant "ISWHM: Tools and Techniques for Software and System Health Management" (NNX08AY50A) and by NASA ARMD 2014 Seedling Phase I, International Research Initiative for Innovation in Aerospace Methods and Technologies (I3AMT), NNX12AK33A. We would like to thank the anonymous reviewers for their detailed feedback to improve this paper.

REFERENCES

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

Austin, T. M. (1999). DIVA: A reliable substrate for deep submicron microarchitecture design. In *Micro* (pp. 196–207). IEEE Computer Society Press.

Barr, M. (2013). *Bookout vs. Toyota, 2005 Camry L4 Software Analysis.* (redacted) Retrieved from `http://www.safetyresearch.net/Library/BarrSlides_FINAL_SCRUBBED.pdf`

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 *Proc. 11th International Conference on Runtime Verification (RV'11)* (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 *Proc. NASA Formal Methods Symposium* (Vol. 7871, pp. 42–68). Springer Verlag.

Bonakdarpour, B., & Smolka, S. A. (Eds.). (2014). *Proc. Runtime Verification, Fifth International Conference, RV'14* (Vol. 8734). Springer Verlag.

Brörkens, M., & Möller, M. (2002). Dynamic event generation for runtime checking using the JDI. *Electronic Notes in Theoretical Computer Science*, *70*(4), 21 - 35.

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 *Proc. 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)* (pp. 2443–2449).

Chenard, J.-S. (2011). Hardware-based temporal logic checkers for the debugging of digital integrated circuits [Dissertation]. *McGill University* Montreal, Canada.

Choi, A., Darwiche, A., Zheng, L., & Mengshoel, O. J. (2011). A tutorial on Bayesian networks for system health management. In A. Srivastava & J. Han (Eds.), *Data mining in systems health management: Detection, diagnostics, and prognostics.* Chapman and Hall/CRC Press.

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.

Drusinsky, D. (2003). Monitoring temporal rules combined with time series. In *CAV* (Vol. 2725, pp. 114–118). Springer Verlag.

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. *Electronic Communications of the EASST*, *46*, 1–15.

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

Geist, J., Rozier, K. Y., & Schumann, J. (2014). Runtime Observer Pairs and Bayesian Network Reasoners On-board FPGAs: Flight-Certifiable System Health Management for Embedded Systems. In *Proc. 14th International Conference on Runtime Verification (RV'14)* (Vol. 8734, pp. 215–230). 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. 21st National Conference on Artificial Intelligence* (pp. 143–148).

Huang, J., Erdogan, C., Zhang, Y., Moore, B., Luo, Q., Sundaresan, A., & Rosu, G. (2014). ROSRV: Runtime Verification for Robots. In *Proc. 14th International Conference on Runtime Verification (RV'14)* (Vol. 8734, pp. 247–254). Springer Verlag.

Ippolito, C., Espinosa, P., & Weston, A. (2010). 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 heterogeneous System. In *Proc. of the 39th International Conference on Parallel Processing* (pp. 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)* (pp. 268–276).

Koller, D., & Friedman, N. (2009). *Probabilistic graphical methods: Principles and techniques*. MIT Press.

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.

Legay, A., & Bensalem, S. (Eds.). (2013). *Proc. Runtime Verification, Fourth International Conference, RV'13* (Vol. 8174). Springer Verlag.

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* (pp. 95–104).

Lindsey, A. E., & Pecheur, C. (2004). Simulation-based verification of autonomous controllers via Livingstone Pathfinder. In *Proc. TACAS 2004* (Vol. 2988, pp. 357–371). Springer Verlag.

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)* (p. 340-349).

Luo, Q., Zhang, Y., Lee, C., Jin, D., Meredith, P. O., Serbanuta, T. F., & Rosu, G. (2014). RV-Monitor: Efficient parametric runtime verification with simultaneous properties. In *Proc. 14th International Conference on Runtime Verification (RV'14).* (Vol. 8734, pp. 285–300). Springer Verlag.

Maler, O., Nickovic, D., & Pnueli, A. (2005). Real Time Temporal Logic: Past, Present, Future. In *FORMATS* (Vol. 3829, pp. 2–16). Springer Verlag.

Maler, O., Nickovic, D., & Pnueli, A. (2007). On synthesizing controllers from bounded-response properties. In *Proc. CAV* (Vol. 4590, pp. 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* (pp. 475–505). Springer Verlag.

Mengshoel, O. J. (2007). Designing resource-bounded reasoners using Bayesian networks: System health monitoring and diagnosis. In *Proc. 18th International Workshop on Principles of Diagnosis (DX)* (pp. 330–337).

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)* (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*, *23*(2), 235–247.

Musliner, D., Hendler, J., Agrawala, A. K., Durfee, E., Strosnider, J. K., & Paul, C. J. (1995). The challenges of real-time AI. *IEEE Computer*, *28*, 58–66.

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 Systems* (pp. 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*. Morgan Kaufmann.

Pearl, J. (2000). *Causality : models, reasoning, and inference*. Cambridge University Press.

Pellizzoni, R., Meredith, P., Caccamo, M., & Rosu, G. (2008). Hardware Runtime Monitoring for Dependable COTS-Based Real-Time Embedded Systems. *RTSS*, 481-491.

Pike, L., Goodloe, A., Morisset, R., & Niller, S. (2010). Copilot: A Hard Real-Time Runtime Monitor. In *Proc. 10th International Conference on Runtime Verification (RV'10)* (Vol. 6418, pp. 345–359). Springer Verlag.

Pike, L., Niller, S., & Wegmann, N. (2011). Runtime verification for ultra-critical systems. In *Proc. 14th International Conference on Runtime Verification (RV'14)* (Vol. 7186, pp. 310–324). Springer Verlag.

Pnueli, A. (1977). The temporal Logic of Programs. In *Proc. 18th Annual Conference on Foundations of Computer Science (FOCS'77)* (pp. 46–57). IEEE Computer Society Press.

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)* (pp. 178–185).

Qadeer, S., & Tasiran, S. (Eds.). (2012). *Proc. Runtime Verification, Third International Conference, RV'12* (Vol. 7687). Springer Verlag.

Reinbacher, T. (2013). *Analysis of embedded real-time systems at runtime* Dissertation. *Vienna University of Technology* Vienna, Austria.

Reinbacher, T., Rozier, K. Y., & Schumann, J. (2014). Temporal-logic based runtime observer pairs for system health management of real-time systems. In *Proc. 20th International Conference on Tools and Algorithms for the Construction and Analysis of Systems (TACAS)* (Vol. 8413, pp. 357–372). Springer-Verlag.

Ricks, B., & Mengshoel, O. J. (2014). Diagnosis for uncertain, dynamic and hybrid domains using Bayesian Networks and Arithmetic Circuits. *International Journal of Approximate Reasoning*, *55*(5), 1207–1234.

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)* (pp. 415–422).

Ricks, B. W., & Mengshoel, O. J. (2009b). Methods for probabilistic fault diagnosis: An electrical power system case study. In *Proc. Annual Conference of the PHM Society (PHM'2009).*

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).*

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 Journal*, *5*(2), 163–203.

Rozier, K. Y., & Vardi, M. Y. (2010). LTL satisfiability checking. *International Journal on Software Tools for Technology Transfer (STTT)*, *12*(2), 123 - 137.

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., Roychoudhury, I., & Kulkarni, C. (2015). Diagnostic reasoning using prognostic information for unmanned aerial systems. In *Proc. Annual Conference of the PHM Society (PHM'2015) PHM (under review).*

Schumann, J., Rozier, K. Y., Reinbacher, T., Mengshoel, O. J., Mbaya, T., & Ippolito, C. (2013). Towards Real-time, On-board, Hardware-supported Sensor and Software Health Management for Unmanned Aerial Systems. In *Proc. Annual Conference of the Prognostics and Health Management Society (PHM'2013).*

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).

Srivastava, A. N., & Schumann, J. (2013). Software Health Management: a necessity for safety critical systems. *Innovations in Systems and Software Engineering*, *9*(1), 219–233.

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

Tsai, J. J., Fang, K. Y., Chen, H. Y., & Bi, Y. (1990). A noninterference monitoring and replay mechanism for real-time software testing and debugging. *Transactions on Software Engineering*, *16*, 897–916.

Watterson, C., & Heffernan, D. (2007). Runtime verification and monitoring of embedded systems. *IET Software*, *1*(5), 172-179.

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. 19th International Symposium on Computer Architecture and High Performance Computing* (pp. 221–228).

Ye, B., & Zhang, Y. (2009). Improved FPGA implementation of particle filter for radar tracking applications. In *2nd Asian-Pacific Conference on Synthetic Aperture Radar (APSAR)* (pp. 943–946).

Zhang, N. L., & Poole, D. (1996). Exploiting causal independence in Bayesian network inference. *Journal of Artificial Intelligence Research*, *5*, 301-328.

Zhao, Y., & Rozier, K. Y. (2012). Formal specification and verification of a coordination protocol for an automated air traffic control system. In *Proc. 12th International Workshop on automated Verification of critical Systems (AVoCS 2012)* (Vol. 53). European Association of Software Science and Technology.

Zhao, Y., & Rozier, K. Y. (2014a). Formal specification and verification of a coordination protocol for an automated air traffic control system. *Science of Computer Programming Journal*, *96*(3), 337-353.

Zhao, Y., & Rozier, K. Y. (2014b). Probabilistic model checking for comparative analysis of automated air traffic control systems. In *Proc. 33rd IEEE/ACM International Conference on Computer-Aided Design (ICCAD 2014)* (pp. 690–695). IEEE/ACM.

Zheng, L., & Mengshoel, O. J. (2013). Optimizing parallel belief propagation in junction trees using regression. In *Proc. of 19th ACM SIGKDD Conference on Knowledge Discovery and Data Mining (KDD-13).*

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).*

BIOGRAPHIES

Dr. Johann Schumann is Chief Scientist for Computational Sciences with SGT, Inc. and working at the NASA Ames Research Center. He received his PhD (1991) and German habilitation degree (2000) in Computer Science from the Technische University Munich in Germany. His general research interests focus on the application of formal and statistical methods to improve design and reliability of advanced safety-critical software. Dr. Johann Schumann is engaged in research on software health management, verification and validation of IVHM algorithms, analysis and V&V of advanced air traffic control algorithms, and the automatic generation of reliable code. He is author of a book on theorem proving in software engineering and has published numerous articles on automated deduction and its applications, automatic program generation, V&V of safety-critical systems, and neural network oriented topics.

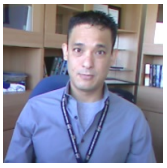
Dr. Kristin Yvonne Rozier is Assistant Professor of Aerospace Engineering and Engineering Mechanics at the University of Cincinnati and currently heads the Laboratory for Temporal Logic; previously she spent 14 years as a Research Scientist at NASA. She earned her PhD (2012) from Rice University. Her research focuses on formal methods, temporal logic, model checking, and automated reasoning. Her advances in computation for the aerospace domain earned her the American Helicopter Society's Howard Hughes Award, the American Institute of Aeronautics and Astronautics Intelligent Systems Distinguished

Service Award, and the Women in Aerospace Inaugural Initiative-Inspiration-Impact Award. She has also earned the Lockheed Martin Space Operations Lightning Award, two NASA Group Achievement Awards, and the SWE Above and Beyond Award. She is an Associate Fellow of AIAA and a Senior Member of IEEE and SWE.

Dr. Thomas Reinbacher obtained his PhD "sub auspiciis praesidentis rei publicae" in Computer Engineering from Vienna University of Technology in 2013. He held various visiting research positions and collaborated with RWTH Aachen University (Germany) and the NASA Ames Research Center (USA). His research interest focus on runtime analysis and formal verification techniques for embedded real time systems. Application wise, he is interested in safety critical, high reliability systems found in aeronautics, aviation and the automotive domain. Dr. Reinbacher is currently working for a professional services company based in Munich, Germany.

Dr. Ole J. Mengshoel is an Associate Research Professor of Electrical and Computer Engineering at CMU Silicon Valley. His current research focuses on reasoning, diagnosis, decision support, and machine learning under uncertainty often using Bayesian networks. Dr. Mengshoel has published over 50 articles and papers in journals and conferences, and holds 4 U.S. patents. He has a Ph.D. in Computer Science from the University of Illinois, Urbana-Champaign. Prior to joining CMU, he was a research scientist in the Knowledge-Based Systems Group at SINTEF (Scandinavia's largest independent research organization), in the Decision Sciences Group at Rockwell Scientific (now Teledyne Scientific and Imaging), and a senior scientist/research area lead at USRA/RIACS (a research organization that supports NASA).

Timmy Mbaya is an avionics engineer at Boeing Defense & Space. His work includes avionics real-time systems for NASA Space Launch System (SLS) and other experimental space systems. Also he is currently pursuing a thesis-based M.S. with a focus on Artificial Intelligence and Intelligent robotics, and emphasis on probabilistic reasoning, from the University of Southern California. He holds a B.S. in Computer Science Summa Cum Laude and distinction from the University of Massachusetts. Mr. Mbaya has been engaged in published research in the field of probabilistic reasoning/Bayesian networks with applications to Integrated Vehicle Health Management/Integrated Software Health Management for aerospace systems for the past five years; including research work at NASA Ames Research Center through RIACS.



Corey Ippolito is a Research Scientist at NASA Ames Research Center and is currently pursuing a PhD at Carnegie Mellon University. He has an M.S. and B.S. in Aerospace Engineering from the Georgia Institute of Technology, and a graduate certificate in Space Systems Engineering from the Stevens Institute of Technology. Mr. Ippolito heads the Exploration Aerial Vehicles and Green Aviation (EAV/GA) laboratory. He has also lead several aircraft development projects, including the Swift UAV, the EAV, and the eXperimental Sensor Controlled Aerial Vehicle (X-SCAV). He has received several awards including NASA Group Achievement Awards for leading a robotic research team in the Atacama Desert and the Surprise Valley UAS project, a NASA Award of Excellence, and a NASA Award for Superior Accomplishment for successful flight test of collaborative UAV/UGV autolanding.