Iowa State University

From the SelectedWorks of Kristin Yvonne Rozier

2014

Runtime Observer Pairs and Bayesian Network Reasoners On-board FPGAs: Flight-Certifiable System Health Management for Embedded Systems

Johannes Geist, *Research Institute for Advanced Computer Science (RIAC)*
Kristin Y. Rozier, *NASA Ames Research Center*
Johann Schumann, *SGT, Inc.*

Available at: https://works.bepress.com/kristin-yvonne-rozier/12/

Runtime Observer Pairs and Bayesian Network Reasoners On-board FPGAs: Flight-Certifiable System Health Management for Embedded Systems ⋆

Johannes Geist[1], Kristin Y. Rozier[2], and Johann Schumann[3]

[1] USRA/RIACS, Mountain View, CA, USA, jgeist@usra.edu
[2] NASA ARC, Moffett Field, CA, USA, Kristin.Y.Rozier@nasa.gov
[3] SGT, Inc., NASA Ames, Moffett Field, CA, USA, Johann.M.Schumann@nasa.gov

Abstract. Safety-critical systems, like Unmanned Aerial Systems (UAS) that must operate totally autonomously, e.g., to support ground-based emergency services, must also provide assurance they will not endanger human life or property in the air or on the ground. Previously, a theoretical construction for paired synchronous and asynchronous runtime observers with Bayesian reasoning was introduced that demonstrated the ability to handle runtime assurance within the strict operational constraints to which the system must adhere. In this paper, we show how to instantiate and implement temporal logic runtime observers and Bayesian network diagnostic reasoners that use the observers' outputs, on-board a field-standard Field Programmable Gate Array (FPGA) in a way that satisfies the strict flight operational standards of REALIZABILITY, RESPONSIVENESS, and UNOBTRUSIVENESS. With this type of compositionally constructed diagnostics framework we can develop compact, hierarchical, and highly expressive health management models for efficient, on-board fault detection and system monitoring. We describe an instantiation of our System Health Management (SHM) framework, rt-R2U2, on standard FPGA hardware, which is suitable to be deployed on-board a UAS. We run our system with a full set of real flight data from NASA's Swift UAS, and highlight a case where our runtime SHM framework would have been able to detect and diagnose a fault from subtle evidence that initially eluded traditional real-time diagnosis procedures.

1 Introduction

Totally autonomous systems operating in hazardous environments save human lives. In order to operate, they must both be able to intelligently react to unknown environments to carry out their missions and adhere to safety regulations to prevent causing harm. NASA's Swift Unmanned Aerial System (UAS) [6] is tasked with intelligently mapping California wildfires for maximally effective deployment of fire-fighting resources yet faces obstacles to deployment, i.e., from the FAA because it must also provably avoid harming any people or property in the air or on the ground in case of off-nominal conditions. Similar challenges are faced by NASA's Viking Sierra-class UAS, tasked with low-ceiling earthquake surveillance, as well as many other autonomous vehicles, UAS, rovers, and satellites. To provide assurance that these vehicles will not cause any

⋆ Additional artifacts to enable reproducibility are available at http://research.kristinrozier.com/RV14.html. This work was supported in part by ARMD 2014 Seedling Phase I and Universities Space Research Association under NASA Cooperative Agreement, International Research Initiative for Innovation in Aerospace Methods and Technologies (I3AMT), NNX12AK33A.

harm during their missions, we propose a framework designed to deliver runtime System Health Management (SHM) [7] while adhering to strict operational constraints, all aboard a low-cost, dedicated, and separate FPGA; FPGAs are standard components used in such vehicles. We name our framework **rt-R2U2** after these constraints:
r*eal*-**t***ime*: SHM must detect and diagnose faults in real time during any mission.
REALIZABLE: We must utilize existing on-board hardware (here an FPGA) providing a generic interface to connect a wide variety of systems to our plug-and-play framework that can efficiently monitor different requirements during different mission stages, e.g., deployment, measurement, and return. New specifications do not require lengthy recompilation and we use an intuitive, expressive specification language; we require real-time projections of Linear Temporal Logic (LTL) since operational concepts for UASs and other autonomous vehicles are most frequently mapped over timelines.
RESPONSIVE: We must continuously monitor the system, detecting any deviations from the specifications within a tight and a priori known time bound and enabling mitigation or rescue measures. This includes reporting intermediate status and satisfaction of timed requirements as early as possible and utilizing them for efficient decision making.
UNOBTRUSIVE: We must not alter any crucial properties of the system, use commercial-off-the-shelf (COTS) components to avoid altering cost, and above all not alter any hardware or software components in such a way as to lose flight-certifiability, which limits us to read-only access to the data from COTS components. In particular, we must not alter functionality, behavior, timing, time or budget constraints, or tolerances, e.g., for size, weight, power, or telemetry bandwidth.
U*nit*: The rt-R2U2 is a self-contained unit.

Previously, we defined a compositional design for combining building blocks consisting of paired temporal logic observers; Boolean functions; data filters, such as smoothing, Kalman, or FFT; and Bayesian reasoners for achieving these goals [17]. We require the temporal logic observer pairs for efficient temporal reasoning but since temporal monitors don't make decisions, Bayesian reasoning is required in conjunction with our temporal logic observer pairs in order to enable the decisions required by this safety-critical system. We designed and proved correct a method of synthesizing paired temporal logic observers to monitor, both synchronously and asynchronously, the system safety requirements and feed this output into Bayesian network (BN) reasoner back ends to enable intelligent handling and mitigation of any off-nominal operational conditions [15]. In this paper, we show how to create those BN back ends and how to efficiently encode the entire rt-R2U2 runtime monitoring framework on-board a standard FPGA to enable intelligent runtime SHM within our strict operational constraints. We demonstrate that our implementation can significantly outperform expert human operators by running it in a hardware-supported simulation with real flight data from a test flight of the Swift UAS during which a fluxgate magnetometer malfunction caused a hard-to-diagnose failure that grounded the flight test for 48 hours, a costly disturbance in terms of both time and money. Had rt-R2U2 been running on-board during the flight test it would have diagnosed this malfunction in real time and kept the UAS flying.

1.1 Related Work

While there has been promising work in Bayesian reasoning for probabilistic diagnosis via efficient data structures in software [16, 18], this does not meet our UNOBTRU-

SIVENESS requirement to avoid altering software or our REALIZABILITY requirement because it does not allow efficient reasoning over temporal traces. For that, we need dynamic Bayes Nets, which are much more complex and necessarily cannot be RESPONSIVE in real time.

There is a wealth of promising temporal-logic runtime monitoring techniques in software, including automata-based, low-overhead techniques, i.e., [5, 19]. The success of these techniques inspires our research question: how do we achieve the same efficient, low-overhead runtime monitoring results, but in hardware since we cannot modify system software without losing flight certifiability? Perhaps the most pertinent is Copilot [14], which generates constant-time and constant-space C programs implementing hard real-time monitors, satisfying our RESPONSIVENESS requirement. Copilot is unobtrusive in that it does not alter functionality, schedulability, certifiability, size, weight, or power, but the software implementation still violates our strict UNOBTRUSIVENESS requirement by executing software. Copilot provides only sampling-based runtime monitoring whereas rt-R2U2 provides complete SHM including BN reasoning.

BusMOP [13, 10] is perhaps most similar to our rt-R2U2 framework. Exactly like rt-R2U2, BusMOP achieves zero runtime overhead via a bus-interface and an implementation on a reconfigurable FPGA and monitors COTS peripherals. However, BusMOP only reports property failure and (at least at present) does not handle future-time logic, whereas we require early-as-possible reporting of future-time temporal properties passing and intermediate status updates. The time elapsed from any event that triggers a property resolution to executing the corresponding handler is up to 4 clock cycles for BusMOP whereas rt-R2U2 always reports in 1 clock cycle. Most importantly, although BusMOP can monitor multiple properties at once, it handles diagnosis on a single-property-monitoring basis, executing arbitrary user-supplied code on the occurrence of any property violation whereas rt-R2U2 performs SHM on a system level, synthesizing BN reasoners that utilize the passage, failure, and intermediate status of multiple properties to assess overall system health and reason about conditions that require many properties to diagnose. Also rt-R2U2 never allows execution of arbitrary code as that would violate UNOBTRUSIVENESS, particularly flight certifiability requirements.

The gNOSIS [8] framework also utilizes FPGAs, but assesses FPGA implementations, mines assertions either from simulation or hardware traces, and synthesizes LTL into, sometimes very large, Finite State Machines that take time to be re-synthesized between missions, violating our REALIZABILITY requirement. Its high bandwidth, automated probe insertion, ability to change timing properties of the system, and low sample-rate violate our UNOBTRUSIVENESS and RESPONSIVENESS requirements, though gNOSIS may be valuable for design-time checking of rt-R2U2 in the future.

1.2 Contributions

We define hardware, FPGA encodings for both the temporal logic runtime observer pairs proposed in [15] and the special BN reasoning units required to process their three-valued output for diagnostics and decision-making. We detail novel FPGA implementations within a specific architecture to exhibit the strengths of an FPGA implementation in hardware in order to fulfill our strict operational requirements; this construction incurs zero runtime overhead. We provide a specialized construction rather than

the standard "algorithm-rewrite-in-VHDL" that may be acceptable for less-constrained systems. We provide timing and performance data showing reproducible evidence that our new rt-R2U2 implementation performs within our required parameters of REALIZABILITY, RESPONSIVENESS, and UNOBTRUSIVENESS in real time. Finally, we highlight implementation challenges to provide instructive value for others looking to reproduce our work, i.e., implementing theoretically proven temporal logic observer constructions on a real-world UAS. Using full-scale, real flight test data streams from NASA's Swift UAS, we demonstrate this real-time execution and prove that rt-R2U2 would have pinpointed in real time a subtle buffer overflow issue that grounded the flight test and stumped human experts for two days in real life.

This paper is organized as follows: Section 2 provides the reader with theoretical principles of our approach. Section 3 provides an overview of the various parts and Sections 4 and 5 give more details about the hardware implementation. A real-world test case of NASA's Swift UAS is evaluated in Section 6. Section 7 concludes this paper with a summary of our findings.

2 Preliminaries

Our system health models are comprised of paired temporal observers, sensor filters, and Bayesian network probabilistic reasoners, all encoded on-board an FPGA; see [17] for a detailed system-level overview.

2.1 Temporal-Logic Based Runtime Observer Pairs [15]

We encode system specifications in real-time projections of LTL. Specifically, we use Metric Temporal Logic (MTL), which replaces the temporal operators of LTL with operators that respect time bounds [1] and mission-time LTL [15], which reduces to MTL with all operator bounds being between now (i.e., time 0) and the mission termination time.

Definition 1 (Discrete-Time MTL [15]). *For atomic proposition $\sigma \in \Sigma$, σ is a formula. Let time bound $J = [t, t']$ with $t, t' \in \mathbb{N}_0$. If φ and ψ are formulas, then so are:*

$$\neg\varphi \mid \varphi \wedge \psi \mid \varphi \vee \psi \mid \varphi \rightarrow \psi \mid \mathcal{X}\varphi \mid \varphi\, \mathcal{U}_J\, \psi \mid \Box_J \varphi \mid \Diamond_J \varphi.$$

Time bounds are specified as intervals: for $t, t' \in \mathbb{N}_0$, we write $[t, t']$ for the set $\{i \in \mathbb{N}_0 \mid t \leq i \leq t'\}$. We interpret MTL formulas over executions of the form $e : \omega \rightarrow 2^{Prop}$; we define φ *holds at time* n *of execution* e, denoted $e^n \models \varphi$, inductively as follows:

$e^n \models true$ is **true**, $e^n \models \sigma$ iff σ holds in s_n,
$e^n \models \neg\varphi$ iff $e^n \not\models \varphi$, $e^n \models \varphi \wedge \psi$ iff $e^n \models \varphi$ and $e^n \models \psi$,
$e^n \models \mathcal{X}\varphi$ iff $e^{n+1} \models \varphi$, $e^n \models \varphi \vee \psi$ iff $e^n \models \varphi$ or $e^n \models \psi$,
$e^n \models \varphi\, \mathcal{U}_J\, \psi$ iff $\exists i (i \geq n) : (i - n \in J \wedge e^i \models \psi \wedge \forall j (n \leq j < i) :\ e^j \models \varphi)$.

Since systems in our application domain are usually bounded to a certain mission time $\tau \in \mathbb{N}_0$, we also encode *mission-time LTL* [15]. For a formula φ in LTL, we create mission-bounded formula φ_m by replacing every $\Box$, $\Diamond$, and $\mathcal{U}$ operator in φ with its

bounded MTL equivalent using the bounds $J = [0, \tau]$. An execution sequence for an MTL formula φ, denoted by $\langle T_\varphi \rangle$, is a sequence of tuples $T_\varphi = (v, \tau_e)$ where $\tau_e \in \mathbb{N}_0$ is a time stamp and $v \in \{\textbf{true}, \textbf{false}, \textbf{maybe}\}$ is a verdict.

For every temporal logic system specification, we synthesize a pair of runtime observers, one asynchronous and one synchronous, using the construction defined and proved correct in [15]. *Asynchronous observers* are *evaluated with every new input*, in this case with every tick of the system clock. For every generated output tuple T we have that $T.v \in \{\textbf{true}, \textbf{false}\}$ and $T.\tau_e \in [0, n]$. Since verdicts are exact evaluations of a future-time specification φ, for each clock tick they may resolve φ for clock ticks prior to the current time n if the information required for this resolution was not available until n. *Synchronous observers* are *evaluated at every tick of the system clock* and their output tuples T are guaranteed to be synchronous to the current time stamp n. Thus, for each time n, a synchronous observer outputs a tuple T with $T.\tau_e = n$. This eliminates the need for synchronization queues. Outputs of these observers are three-valued verdicts: $T.v \in \{\textbf{true}, \textbf{false}, \textbf{maybe}\}$ depending on whether we can concretely valuate that the observed formula holds at this time point (**true**), does not hold (**false**), or cannot be evaluated due to insufficient information (**maybe**). Verdicts of **maybe** are later resolved concretely by the matching asynchronous observers in the first clock tick when sufficient information is available.

2.2 Bayesian Networks for Health Models

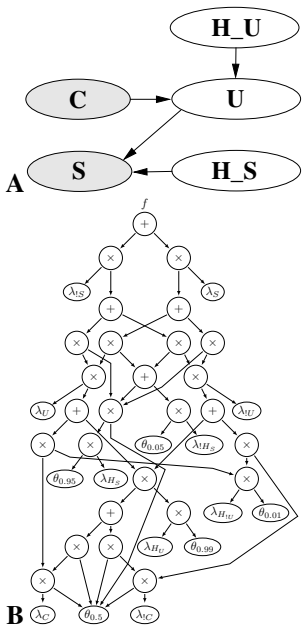


Fig. 1. A: BN for Health management. **B:** Arithmetic circuit

In order to maximize the reasoning power of our health management system, we use Bayesian networks (BN). BNs have been well established in the area of diagnostic and health management (e.g., [12, 9]) as they can cope with conflicting sensor signals and priors. BNs are directed acyclic graphs, where each node represents a statistical variable. Directed edges between nodes correspond to (local) conditional dependencies. For our health models, we are using BNs of a general structure as shown in Figure 1A. We do not use dynamic BNs, because all temporal aspects are being dealt with by the temporal observers described above. Discrete sensor signals or outputs of the synchronous temporal observers (**true**, **false**, **maybe**) are clamped to the "sensor" and "command" nodes of the BN as observable. Since sensors can fail, they have (unobservable) health nodes attached. As priors, these health nodes can contain information on how reliable the component is, e.g., by using a Mean Time To Failure (MTTF) metric.

Unobservable nodes U may describe the behavior of the system or component as it is defined and influenced by the sensor or software information. Often, such nodes are used to define a mode or state of the

system. For example, it is likely that the UAS is climbing if the altimeter sensor says "altitude increasing." Such (desired) behavior can also be affected by faults, so behavior nodes have health nodes attached. For details of modeling see [16]. The local conditional dependencies are stored in the Conditional Probability Table (CPT) of each node. For example, the CPT of the sensor node S defines its probabilities given its dependencies: $P(S|U, H\_S)$.

In our health management system, we, at each time stamp, calculate the posterior probabilities of the BN's health nodes, given the sensor and command values $\mathbf{e}$ as evidence. The probability $Pr(H\_S = good|\mathbf{e})$ gives an indication of the status of the sensor or component. Reasoning in real-time avionics applications requires aligning resource consumption of diagnostic computations with tight resource bounds [11]. We are therefore using a representation of BNs that is based upon arithmetic circuits (AC), which are directed acyclic graphs where leaf nodes represent indicators (λ in Fig. 1) and parameters (θ) while all other nodes represent addition and multiplication operators. AC based reasoning algorithms are powerful, as they provide predictable real-time performance [2, 9].

The AC is factually a compact encoding of the joint distribution into a network polynomial [3]. The marginal probability (see Corollary 1 in [3]) for a variable x given evidence $\mathbf{e}$ can then be calculated as $Pr(x \,|\, \mathbf{e}) = \frac{1}{\Pr(\mathbf{e})} \cdot \frac{\partial f}{\partial \lambda_x}(\mathbf{e})$ where $Pr(\mathbf{e})$ is the probability of the evidence. In a first, bottom-up pass, the λ indicators are clamped according to the evidence and the probability of this particular evidence setting is evaluated. A subsequent top-down pass over the circuit computes the partial derivatives $\frac{\partial f}{\partial \lambda_x}$. Based upon the structure of the AC, this algorithm only requires —except for the final division by $Pr(\mathbf{e})$— only additions and multiplications. Since the structure of the AC is determined at compile time, a fixed, reproducible timing behavior can be guaranteed.

2.3 Digital Design 101 and FPGAs

Integrated circuits (ICs) have come a long way from the first analog, vacuum tube-based switching circuits, over discrete semiconductors to sub-micron feature size for modern ICs. Our ability to implement rt-R2U2 in hardware is strongly based upon high-level hardware definition languages and tools to describe the functionality of the hardware design, and FPGAs, which make it possible to "instantiate" the hardware on-the-fly without having to go through costly silicon wafer production.

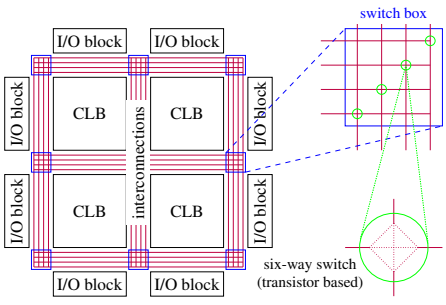


Fig. 2. Simplified representation of a modern FPGA architecture.

VHDL - Very High Speed Integrated Circuit Hardware Definition Language. This type-safe programming language allows the concise description of concurrent systems, supporting the inherent nature of any IC. Therefore, programming paradigms are substantially different from software programming languages, e.g., memory usage and mapping has to be considered explicitly and algorithms with loops have to be rewritten into finite state machines. In general, a lot more time and effort has to be put into system design.

FPGA - Field Programmable Gate Array is a fast, cheap, and efficient way to produce a custom-designed digital system or prototype. Basically an FPGA consists of logic cells (Figure 2), that can be programmed according to its intended use. A modern FPGA is composed of three main parts *Configurable Logic Blocks* (CLBs), long and short *interconnections* with six-way programmable switches, and *I/O blocks*. The CLBs are elementary Look Up Tables (LUTs) where, depending on the input values, a certain output value is presented to the next cell. Hence, every possible combination of unary operations can be programmed. Complex functionality can be achieved by connecting different CLBs using short (between neighboring cells) and long interconnections. These interconnections need the most space on an FPGA, because in general every cell can be connected to every other cell. The I/O cells are also connected to this interconnection grid. To be able to route the signals in all directions there is a "switch box" on every intersection. This six-way switch is based on 6 transistors that can be programmed to route the interconnection accordingly. In order to achieve higher performance modern FPGAs have hardwired blocks for certain generic or complex operations (adder, memory, multiplier, I/O transceiver, etc.).

3 System Overview

Our system health models are constructed based upon information extracted from system requirements, sensor schematics, and specifications of expected behaviors, which are usually written in natural language. In a manual process (Figure 3) we develop the health model in our framework, which is comprised of temporal components (LTL and MTL specifications), Bayesian networks (BNs), and signal processing. Our tool chain compiles the individual parts and produces binary files, which, after linking, are downloaded to the FPGA. The actual hardware architecture, which is defined in VHDL, is compiled using a commercial tool chain[4] and used to configure the FPGA. This lengthy process, which can take more than 1 hour on a high-performance workstation needs to be carried out only once, since it is independent of the actual health model.

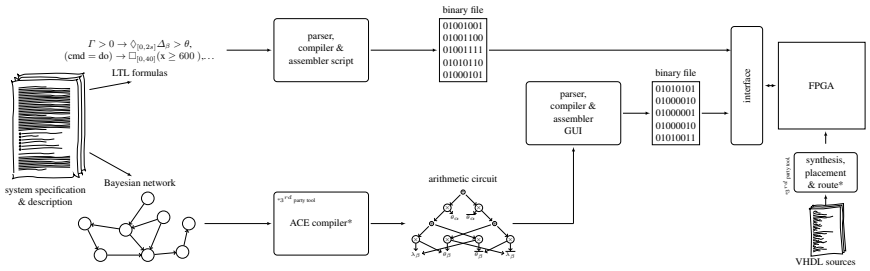


Fig. 3. rt-R2U2 software tool chain

[4] http://www.xilinx.com/products/design-tools/ise-design-suite/index.htm

3.1 Software

The software tool chain for creating the code for the temporal logic specifications is straightforward and only translates the given formulas to a binary representation with mapping information. Significantly more effort goes into preparing a BN for our system. First, the given network is translated into an optimized arithmetic circuit (AC) using the Ace[5] tool. Then, the resulting AC must be compiled and mapped for efficient execution on the FPGA. This process, which will be described in more detail in Section 5, is controlled with a Java GUI.

3.2 Hardware

The hardware architecture (Figure 4A) of our implementation is built out of three components: the *control subsystem*, the *runtime verification* (RV) unit, and the *runtime reasoning* (RR) unit. Whereas the control subsystem establishes the communication link to the external world (e.g., to load health models and to receive health results), the RV and RR units comprise the proper health management hardware, which we will discuss in detail in the subsequent sections. Any sensor and software data passed along the Swift UAS bus can be directly fed into the signals' filters and pre-processing modules of the *atChecker*, which are a part of the RV unit, where they are converted into streams of Boolean values.

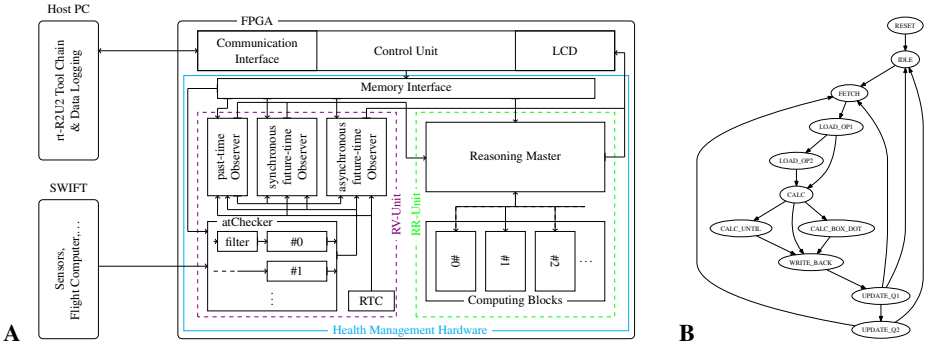


Fig. 4. A: Overview of the rt-R2U2 architecture. **B:** FSM for the ftObserver.

Our architecture is designed in such a way that its requirements with respect to gates and look-up tables only depend on the number of signals we monitor, not on the temporal logic formulas or the Bayesian networks. In the configuration used for our case study (with 12 signals), the monitoring device synthesized for the Xilinx Virtex 5 XC5VFX130T FPGA needed 28849 registers, 24450 look-up tables, 63 blocks of RAM, and 25 digital signal processing units. These numbers clearly strongly depend on the architecture of the FPGA, and, in our case used 35% of the registers, 29% of the LUTs, 21% of the RAM, and 7% of the DSP blocks.

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

The runtime verification subsystem evaluates the compiled temporal logic formulas over the Boolean signals prepared by the atChecker. Since evaluations of the past-time variations of our logics (MTL and mission-time LTL) are naturally synchronous, we can essentially duplicate the synchronous observer construction, but with past-time evaluation, to add support for past-time formulas should they prove useful in the context of the system specifications. Depending on the type of logic encoding each individual formula (past or future time), it is either evaluated by the past-time or future-time subsystem. As the algorithms are fundamentally different for the two time domains we use two separate entities in the FPGA. A real time clock (RTC) establishes a global time domain and provides a time base for evaluating the temporal logic formulas.

After the temporal logic formulas have been evaluated, the results are transferred to the runtime reasoning (RR) subsystem, where the compiled Bayesian network is evaluated to yield the posterior marginals of the health model. For easier debugging and evaluation, a memory dump of the past and future time results as well as of the posterior marginals has been implemented. After each execution cycle, the evaluation is paused and the memory dump is transferred to the host PC for further analysis.

4 FPGA implementation of MTL/mission-time LTL

As shown in Figure 4A, incoming sensor and software signals, which consist of vectors of binary fixed-point numbers, are first processed and discretized by the atChecker unit. This hardware component can contain filters to smooth the signal, Fast Fourier Transforms, or Kalman Filters, and performs scaling and comparison operations to yield a Boolean value. 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 constants p_1, p_2, and c, filters F_j^i, and a comparison operator $\bowtie \in \{=, <, \leq, \geq, >, \neq\}$. For example, the discrete signal "UAS is at least 400ft above ground" would be specified by: $(mvg\_avg(alt_{UAS}) - alt_{gnd}) > 400$, where the altitude measurements of the UAS would be smoothed out by a moving average filter before the altitude of the ground is subtracted. Note that several blocks can be necessary for thresholding, e.g., to determine if the UAS is above 400ft, 1000ft, or 5000ft.

Each temporal logic processing unit (ptObserver, ftObserver) is implemented as a processor, which executes the compiled formulas instruction by instruction. It contains its own program and data memory, and finite-state-machine (FSM) based execution unit (Figure 4B[6]). Individual instructions process Boolean operators and temporal logic operators using the stages of FETCH (fetch instruction word) followed by loading the appropriate operand(s). Calculation of the result can be accomplished in one step (CALC) or might require an additional state for the more complex temporal operations like $\mathcal{U}$ or $\Box_{[.,.]}$. During calculation, values for the synchronous and asynchronous operators are updated according to the logic's formal algorithm (see [15]). Finally, results are written back into memory (WRITE) and the queues are updated during states (UPDATE_Q1, UPDATE_Q2), before the execution engine goes back to its IDLE state. Asynchronous temporal observers usually need local memory for keeping information like the time

[6] The architecture and FSM for processing the past time fragment is similar to this unit and thus will not be discussed here.

stamps for the last rising transition or the start time of the next tuple in the queues, which are implemented using a ring buffer. Internal functions *feasible* and *aggregate* put information (timestamps) into the ring buffer, whereas a highly specialized garbage collecting function removes time stamps that can no longer contribute to the validity of the formula, thus keeping memory requirements low. These updates to the queues happen during the UPDATE states of the processor ([15]).

In contrast to asynchronous observers, which require additional memory for keeping internal history information, *synchronous* observers are realized as memoryless Boolean networks. Their three-valued logic {**false**, **true**, **maybe**} is encoded in two binary signals as $\langle 0,0\rangle$, $\langle 0,1\rangle$, and $\langle 1,0\rangle$, respectively.

Let us consider the following specification, which expresses that the UAS, after receiving the takeoff command must reach an altitude alt above ground of at least 600ft within 40 seconds: cmd $=$ takeoff $\rightarrow \Diamond_{[0,40s]}(alt \geq 600)$. Obviously, synchronous and asynchronous observers report **true** before the takeoff. After takeoff, the synchronous observer immediately returns **maybe** until the 40-second time window has expired or the altitude exceeds 600ft, whichever comes first. Then the formula can be decided to yield **true** or **false**. In contrast, the asynchronous observer always yields the concrete valuation of the formula, **true** or **false**, for every time stamp; however this result (which is always tagged with a time stamp) might retroactively resolve an earlier point in time.

For rt-R2U2, both types of observers are important. Whereas asynchronous observers guarantee the concrete result but might refer to an earlier system state, synchronous observers immediately yield some information, which can be used by the Bayesian network to disambiguate failures. In our example, this information can be used to express that, with a certain (albeit unknown) probability, the UAS still can reach the desired target in time, but hasn't done so yet. Our Bayesian health models can reflect that fact by using three-valued sensor and command nodes.

5 FPGA implementation of Bayesian Networks

The BN reasoning has been implemented on the FPGA as a Multiple Instruction, Multiple Data (MIMD) architecture. This means that every processing unit calculates a part of the AC using its individual data and program memory. That way, a high degree of parallelism can be exploited and we can obtain a high performance and low latency evaluation unit. Therefore, our architectural design process led to a simple, tightly coupled hardware architecture, which relies on optimized instructions provided by the BN compiler (Figure 3). The underlying idea of this architecture is to partition the entire arithmetic circuit into small parts of constant size, which in turn are processed by a number of parallel execution units with the goal of minimizing inter-processor data exchanges and synchronization delays. We will first describe the hardware architecture and then focus on the partitioning algorithm in the BN compiler.

BN Computing Block. We designed an elementary BN processor (BN computing block) that can process three different kinds of small "elementary" arithmetic circuits. A number of identical copies (the number depends on the size of the FPGA) of these computing blocks work as slaves in a master-slave configuration. Figure 5A shows the three different patterns. Each pattern consists of up to three arithmetic operators (addi-

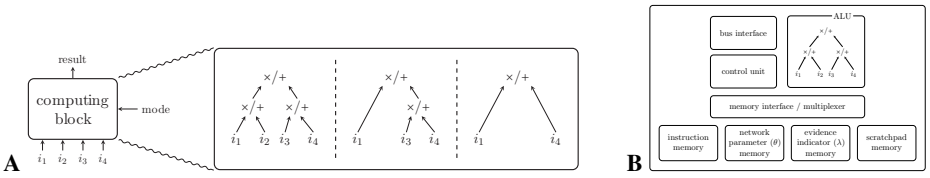


Fig. 5. A: A computing block and its three modes of operation. **B:** Internals of a computing block.

tion or multiplication) and can have 2, 3, or 4 inputs. Such a small pattern can be efficiently executed by a BN computing block. Figure 5B shows a BN computing block, which is built from several separate hardware units (bus interface, local memory, instruction decoder, ALU, etc.). On an abstract level the calculation is based on a generic four-stage pipeline execution (FETCH, DECODE, CALCULATE, and WRITE-BACK). To achieve this performance-focused behavior, each subsystem runs independently. Therefore, a handshake synchronizing protocol between each internal component is used. As a MIMD processor, each BN computing block keeps its own instruction memory as well as local storage for network parameters and evidence indicators. A local scratchpad memory is used to store intermediate results.

Although probabilities are best represented using floating-point numbers according to IEEE 754, we chose to use an 18-bit fixed-point representation, because floating-point ALUs are resource-intensive in terms of both number of logic gates used and power, and would drastically reduce the number of available parallel BN computing blocks. Our chosen resolution is based on the 18-bit hardware multiplier that is available on our Xilinx Virtex 5 FPGA. We achieve a resolution of $2^{-18} = 3.8 \cdot 10^{-6}$, which is sufficient for our purposes to represent probability values.

All slave processors are connected via a bus to the BN master processor. Besides programming, data handling, and controlling their execution, the master also calculates the final result $Pr(x \mid \mathbf{e}) = \frac{1}{\Pr(\mathbf{e})} \cdot \frac{\partial f}{\partial \lambda_x}(\mathbf{e})$, because the resources needed to perform the division are comparatively high and therefore not replicated over the slave processors.

Mapping of AC to BN computing units. Our software tool chain tries to achieve an optimal mapping of the AC to the different BN computation units during compile time, using a pattern-matching-based algorithm. We "tile" the entire AC with the three small patterns (Figure 5A) in such a way that the individual BN processing units operate as parallel as possible and communication and data transfer is reduced to a minimum. For this task, we use a Bellman-Ford algorithm to obtain the optimal placement. Furthermore, all scheduling information (internal reloads and communication on the hardware bus to exchange data with other computing blocks) as well as the configuration for the master and probability values for the Conditional Probability Table (CPT) are prepared for the framework.

6 Case Study: Fluxgate Magnetometer Buffer Overflow

In 2012, a NASA flight test of the Swift UAS was grounded for 48 hours as system engineers worked to diagnose an unexpected problem with the UAS that ceased vital data transmissions to the ground. All data of the scientific sensors on the UAS (e.g.,

laser altimeter, magnetometer, etc.) were collected by the Common Payload System (CPS). The fluxgate magnetometer (FG), which measures strength and direction of the Earth's magnetic field, had previously failed and was replaced before the flight test. System engineers eventually determined that the replacement was not configured correctly; firmware on-board the fluxgate magnetometer was sending data to its internal transmit buffer at high speed although the intended speed of communication with the CPS was 9600 baud. As the rate was set to a higher value and the software in the magnetometer did not catch this error, internal buffer overflows started to occur, resulting in an increasing number of corrupted packets sent to the CPS. This misconfiguration in the data flow was very difficult to deduce by engineers on the ground because they had to investigate the vast number of possible scenarios that could halt data transmission.

Signal	description	Source
N_g	number of good FG packets since start of mission	CPS
N_b	number of bad FG packets since start of mission	CPS
E^{log}	logging event	CPS
$FG_{x,y,z}$	directional fluxgate magnetometer reading	CPS
$Hd_{x,y}$	aircraft heading	FC
p, q, r	pitch, roll, and yaw rate	FC

Table 1: Signals and sources used in this health model, sampled with a 1Hz sampling rate

In this case study, we use the original data as recorded by the Swift Flight Computer (FC) and the CPS. At this time, no publicly available report on this test flight has been published; the tests and their resulting data are identified within NASA by the date and location, Surprise Valley, California on May 8, 2012, starting at 7:50 am. With our rt-R2U2 architecture, which continuously monitors our standard set of rates, ranges, and relationships for the on-board sensors, we have been able to diagnose this problem in real-time, and could have avoided the costly delay in the flight tests.

The available recorded data are time series of continuous and discrete sensor and status data for navigational, sensor, and system components. From the multitude of signals, we selected, for the purpose of this case study, the signals shown in Table 1. We denote the total number of packets from the FG with $N_{tot} = N_g + N_b$; $X^R = X^t - X^{t-1}$ is the rate of signal X, and X^N denotes the normalized vector X.

6.1 The Bayesian Health Model

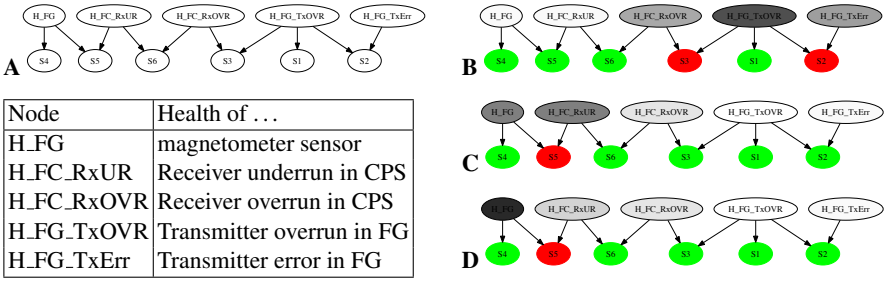
The results of the temporal specifications $S_1, \ldots, S_6$ alone are not sufficient to disambiguate the different failure modes. We are using the Bayesian network as shown in Figure 6A, which receives, as evidence, the results of each specification S_i and produces posterior marginals of the health nodes for the various failure modes. All health nodes are shown in Figure 6A. H_FG indicates the health of the FG sensor itself. It is obviously related to evidence that the measurements are valid (S_4) and that the measurements are changing over time (S_5). The two causal links from these health nodes indicate that relationship. Failure modes H_FG_TXERROR and H_FG_TXOVR indicate an

Description	Formula
S_1: The FG packet transmission rate N^R_{tot} is appropriate: about 64 per second.	$63 \leq N^R_{tot} \leq 66$
S_2: The number of bad packets N^R_b is low, no more than one bad packet every 30 seconds.	$\square_{[0,30]}(N^R_b = 0 \vee (N^R_b \geq 1\, \mathcal{U}_{[0,30]} N^R_b = 0))$
S_3: The bad packet rate N^R_b does not appear to be increasing; we do not see a pattern of three bad packets within a short period of time.	$\neg(\Diamond_{[0,30]} N^R_b \geq 2 \wedge \Diamond_{[0,100]} N^R_b \geq 3)$
S_4: The FG sensor is working, i.e., the data appears good. Here, we use a simple, albeit noisy sanity check by monitoring if the aircraft heading vector with respect to the x and y coordinates (Hd_x, Hd_y) calculated by the flight computer using the magnetic compass and inertial measurements roughly points in the same direction (same quadrant) as the normalized fluxgate magnetometer reading (FG^N_x, FG^N_y). To avoid any false positive evaluations due to a noisy sensor, we filter the input signal.	$((Hd_x \geq 0 \rightarrow FG^N_x \geq 0) \wedge$ $(Hd_x < 0 \rightarrow FG^N_x < 0)) \vee$ $((Hd_y \geq 0 \rightarrow FG^N_y \geq 0) \wedge$ $(Hd_y < 0 \rightarrow FG^N_y < 0))$
S_5: We have a subformula Eul that states if the UAS is moving (Euler rates of pitch p, roll q, and yaw r are above the tolerance thresholds $\theta = 0.05$) then the fluxgate magnetometer should also register movement above its threshold $\theta_{FG} = 0.005$. The formula states that this should not fail more than three times within 100 seconds of each other.	$Eul := (\lvert p\rvert > \theta \vee \lvert q\rvert > \theta \vee \lvert r\rvert > \theta) \rightarrow$ $(\lvert FG_x\rvert > \theta_{FG} \vee \lvert FG_y\rvert > \theta_{FG} \vee$ $\lvert FG_z\rvert > \theta_{FG})$ $\neg(\neg Eul \wedge (\Diamond_{[2,100]}(\neg Eul \wedge \Diamond_{[2,100]} \neg Eul)))$
S_6: Whenever a logging event occurs, the CPS has received a good or a bad packet. S_6 needs a sampling rate of at least 64Hz.	$E^{log} \rightarrow ((E^{log}_g \wedge \neg E^{log}_b) \vee (E^{log}_b \wedge \neg E^{log}_g)$
S'_6: This case study uses a 1Hz sampling rate. We are losing precision and S_6 becomes $N^R_g + N^R_b = N^R_{tot} = 64$.	$N^R_{tot} = 64$

Table 2: Temporal formula specifications that are translated into paired runtime observers for the fluxgate magnetometer (FG) health model

error in the transmission circuit/software and overflow of the transmission buffer of the fluxgate magnetometer, respectively. The final two failure modes H_FC_RXOVR and H_FC_RXUR concern the receiver side of the CPS and denote problems with receiver buffer overflow and receiver buffer underrun, respectively.

Figure 6B shows the reasoning results of this case study, where the wrong configuration setting of the fluxgate magnetometer produces an increasing number of bad packets. The posterior of the node H_FG_TXOVR is substantially lower, compared to the other health nodes, indicating that a problem in the fluxgate magnetometer's transmitter component is most likely. So, debugging and repair attempts or on-board mitigation can be focused on this specific component, thus our SHM could have potentially avoided the extended ground time of the Swift UAS. This situation also indicates that, with a

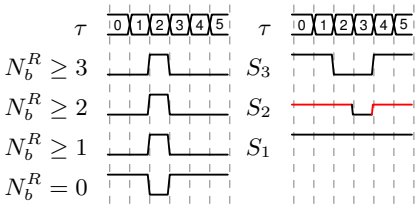


Node	Health of ...
H_FG	magnetometer sensor
H_FC_RxUR	Receiver underrun in CPS
H_FC_RxOVR	Receiver overrun in CPS
H_FG_TxOVR	Transmitter overrun in FG
H_FG_TxErr	Transmitter error in FG

Fig. 6. A: Bayesian network for our example with legend of health nodes. **B, C, D**: posterior probabilities (lighter shading corresponds to values closer to 1.0) for different input conditions.

smaller likelihood, this failure might have been caused by some kind of overrun of the receiver circuit in the flight computer, or specific errors during transmission.

Figures 6C, D show the use of prior information to help disambiguate failures. Assume that we detected that the FG data are not changing, i.e., S_5 = **false**, despite the fact that the aircraft is moving. This could have two causes: the sensor itself is broken, or something in the software is wrong and no packets are reaching the receiver, causing an underrun there. When this evidence is applied (red indicates **false**, green indicates **true**), the posterior of all nodes is close to 1 (white); only H_FG and H_FC_RXUR show values around 0.5 (gray), indicating that these two failures cannot be properly distinguished. This is not surprising, since we set the priors to $P(H_{sensor} = ok) = P(H\_FC\_RxUR) = 0.99$. Making the sensor less reliable, i.e., $P(H_{sensor} = ok) = 0.95$, now enables the BN to clearly disambiguate both failure modes. Further disambiguation information is provided by S_5, which indicates that we actually receive valid (i.e., UAS is moving) packets.



		$\tau = 1$	$\tau = 2$	$\tau = 3$
H_FC_RxOVR	ok	99.47%	17.27%	65.52%
	bad	0.53%	82.73%	34.48%
H_FG_TxOVR	ok	99.88%	81.82%	31.03%
	bad	0.12%	18.18%	68.97%
H_FG_TxErr	ok	90.00%	90.00%	62.07%
	bad	10.00%	10.00%	37.93%

Fig. 7. Recorded traces: sensor signals (left), trace of $S_1 \ldots S_3$ (middle). Data of health nodes (right) reflecting the buffer overflow situation shown in 6B.

As the case study is based on a real event, we ran it on our hardware and extracted a trace of the sensor signals and specifications. Figure 7 shows a small snippet from this trace. The results of the atChecker evaluation of certain sensor signals can be seen on the left. On the right we show the results of S_1 to S_3. The system model delivers different health estimations during this trace. While at $\tau = 1$ the system is perfectly healthy, at $\tau = 2$ the rate of bad packets drastically increases. More than 3 bad packets have been received within 30 seconds. While the violation of S_3 would suggest a receiver overrun at this time, the indication for a buffer overflow becomes concrete at $\tau = 3$. This is

indicated in the table on the right in Figure 7. The high probability of a transmitter overrun at the fluxgate magnetometer side with the reduced confidence of an error-free transition, leads to determining a root cause at the fluxgate magnetometer buffer.

7 Conclusion

We have presented an FPGA-based implementation for our health management framework called rt-R2U2 for the runtime monitoring and analysis of important safety and performance properties of a complex unmanned aircraft, or other autonomous systems. A combination of temporal logic observer pairs and Bayesian networks makes it possible to define expressive, yet compact health models. Our hardware implementation of this health management framework using efficient special-purpose processors allows us to execute our health models in real time. Furthermore, new or updated health models can be loaded onto the FPGA quickly between missions without having to re-synthesize its entire configuration in a time-consuming process.

We have demonstrated modeling and analysis capabilities on a health model, which monitors the serial communication between the payload computer and sensors (e.g., an on-board fluxgate magnetometer) on NASA's Swift UAS. Using data from an actual test flight, we demonstrated that our health management system could have quickly detected a configuration problem of the fluxgate magnetometer as the cause for a buffer overflow—the original problem grounded the aircraft for two days until the root cause could be determined.

Our rt-R2U2 system health management framework is applicable to a wide range of embedded systems, including CubeSats and rovers. Our independent hardware implementation allows us to monitor the system without interfering with the previously-certified software. This makes rt-R2U2 amenable both for black-box systems, where only the external connections/buses are available (like the Swift UAS), and monitoring white-box systems, where potentially each variable of the flight software could be monitored.

There is of course a question of trade-offs in any compositional SHM framework like the one we have detailed here: for any combination of data stream and off-nominal behavior, where is the most efficient place to check for and handle that off-nominal behavior? Should a small wobble in a data value be filtered out via a standard analog filter, accepted by a reasonably lenient temporal logic observer, or flagged by the BN diagnostic reasoner? In the future, it would be advantageous to complete a study of efficient design patterns for compositional temporal logic/BN SHM and map the types of checks we need to perform and the natural variances in sensor readings that we need to allow for their most efficient implementations.

Future work will also address the challenges of automatically generating health models from requirements and design documents, and carrying out flight tests with our FPGA-based rt-R2U2 on-board. In a next step, the output of rt-R2U2 could be connected to an on-board decision-making component, which could issue commands to loiter, curtail the mission, execute an emergency landing, etc.. Here, probabilistic information and confidence intervals calculated by the Bayesian networks of our approach can play an important role in providing solid justifications for decisions made.

References

1. Alur, R., Henzinger, T.A.: Real-time Logics: Complexity and Expressiveness. In: LICS. pp. 390–401. IEEE Computer Society Press (1990)
2. Chavira, M., Darwiche, A.: Compiling Bayesian networks with local structure. In: Proceedings of the 19th International Joint Conference on Artificial Intelligence (IJCAI). pp. 1306–1312 (2005)
3. Darwiche, A.: A differential approach to inference in Bayesian networks. Journal of the ACM 50(3), 280–305 (2003)
4. Darwiche, A.: Modeling and reasoning with Bayesian networks. In: Modeling and Reasoning with Bayesian Networks (2009)
5. Drusinsky, D.: The temporal rover and the ATG rover. In: SPIN. Lecture Notes in Computer Science, Vol. 1885, pp. 323–330. Springer Verlag (2000)
6. Ippolito, C., Espinosa, P., Weston, A.: Swift UAS: An electric UAS research platform for green aviation at NASA Ames Research Center. In: CAFE EAS IV (April 2010)
7. Johnson, S., Gormley, T., Kessler, S., Mott, C., Patterson-Hine, A., Reichard, K., Philip Scandura, J.: System Health Management: with Aerospace Applications. Wiley & Sons (2011)
8. Majzoobi, M., Pittman, R.N., Forin, A.: gNOSIS: Mining FPGAs for verification (2011)
9. Mengshoel, O.J., Chavira, M., Cascio, K., Poll, S., Darwiche, A., Uckun, S.: Probabilistic model-based diagnosis: An electrical power system case study. IEEE Trans. on Systems, Man and Cybernetics, Part A: Systems and Humans 40(5), 874–885 (2010)
10. Meredith, P.O., Jin, D., Griffith, D., Chen, F., Roşu, G.: An overview of the mop runtime verification framework. International Journal on Software Tools for Technology Transfer 14(3), 249–289 (2012)
11. Musliner, D., Hendler, J., Agrawala, A.K., Durfee, E., Strosnider, J.K., Paul, C.J.: The challenges of real-time AI. IEEE Computer 28, 58–66 (January 1995), citeseer.comp.nus.edu.sg/article/musliner95challenges.html
12. Pearl, J.: A constraint propagation approach to probabilistic reasoning. In: UAI. pp. 31–42. AUAI Press (1985)
13. Pellizzoni, R., Meredith, P., Caccamo, M., Rosu, G.: Hardware runtime monitoring for dependable COTS-based real-time embedded systems. RTSS pp. 481–491 (2008)
14. Pike, L., Wegmann, N., Niller, S., Goodloe, A.: Copilot: monitoring embedded systems. Innovations in Systems and Software Engineering 9(4), 235–255 (2013)
15. Reinbacher, T., Rozier, K.Y., Schumann, J.: Temporal-logic based runtime observer pairs for system health management of real-time systems. In: Proceedings of the 20th International Conference on Tools and Algorithms for the Construction and Analysis of Systems (TACAS). Lecture Notes in Computer Science (LNCS), Vol. 8413, pp. 357–372. Springer-Verlag (April 2014)
16. Schumann, J., Mbaya, T., Mengshoel, O.J., Pipatsrisawat, K., Srivastava, A., Choi, A., Darwiche, A.: Software health management with Bayesian networks. Innovations in Systems and Software Engineering 9(2), 1–22 (2013)
17. Schumann, J., Rozier, K.Y., Reinbacher, T., Mengshoel, O.J., Mbaya, T., Ippolito, C.: Towards real-time, on-board, hardware-supported sensor and software health management for unmanned aerial systems. In: Proceedings of the 2013 Annual Conference of the Prognostics and Health Management Society (PHM2013). pp. 381–401 (October 2013)
18. Srivastava, A.N., Schumann, J.: Software health management: a necessity for safety critical systems. Innovations in Systems and Software Engineering 9(4), 219–233 (2013)
19. Tabakov, D., Rozier, K.Y., Vardi, M.Y.: Optimized temporal monitors for SystemC. Formal Methods in System Design 41(3), 236–268 (2012)

View publication stats