NASA/TM–2015–218817

Intelligent Hardware-Enabled Sensor and Software Safety and Health Management for Autonomous UAS

PI: *Kristin Y. Rozier*
civil servant, NASA Ames Research Center, Moffett Field, CA 94035, USA

Co-I: *Johann Schumann*
SGT, Inc., NASA Ames Research Center, Moffett Field, CA 94035, USA

Co-I: *Corey Ippolito*
civil servant, NASA Ames Research Center, Moffett Field, CA 94035, USA

May 2015

NASA/TM–2015–218817

Intelligent Hardware-Enabled Sensor and Software Safety and Health Management for Autonomous UAS

PI: *Kristin Y. Rozier*
civil servant, NASA Ames Research Center, Moffett Field, CA 94035, USA

Co-I: *Johann Schumann*
SGT, Inc., NASA Ames Research Center, Moffett Field, CA 94035, USA

Co-I: *Corey Ippolito*
civil servant, NASA Ames Research Center, Moffett Field, CA 94035, USA

National Aeronautics and
Space Administration

Ames Research Center
Moffett Field, California 94035-1000

May 2015

Acknowledgments

We thank our student interns and collaborators for their invaluable contributions to this project. Thanks to: Johannes Geist (M.S. Research Intern, University of Applied Sciences Tecnikum Wien, Austria); Chetan S. Kulkarni (Research Engineer, SGT, Inc., NASA Ames Research Center, Moffett Field, CA); Eddy Mazmanian (civil servant, NASA Ames Research Center, Moffett Field, CA); Patrick Moosbrugger (M.S. Research Intern, University of Applied Sciences Tecnikum Wien, Austria); Quoc-Sang Phan (Ph.D. Research Intern, Queen Mary University of London, UK); Thomas Reinbacher (Ph.D. Research Intern, Vienna University of Technology, Austria); Indranil Roychoudhury (Computer Scientist, SGT, Inc., NASA Ames Research Center, Moffett Field, CA); Iyal Suresh (sophomore Undergraduate Intern, University of California, Los Angeles).

The project team worked closely with members of the airborne science group at NASA Ames Research Center in Code SG. We would like to give special thanks to Matt Fladeland (ARC/SG), Ric Kolyer (ARC/SG), Bruce Storms (ARC/AOX), and the rest of the Dragon Eye team who we collaborated with for modification and flight testing of the NASA Dragon Eye aircraft systems.

This report is available in electronic form at
http://research.kristinrozier.com/R2U2/NASA_TM_FinalReport.pdf

Abstract

Unmanned Aerial Systems (UAS) can only be deployed if they can effectively complete their mission and respond to failures and uncertain environmental conditions while maintaining safety with respect to other aircraft as well as humans and property on the ground. We propose to design a real-time, onboard system health management (SHM) capability to continuously monitor essential system components such as sensors, software, and hardware systems for detection and diagnosis of failures and violations of safety or performance rules during the flight of a UAS. Our approach to SHM is three-pronged, providing: (1) real-time monitoring of sensor and software signals; (2) signal analysis, preprocessing, and advanced on-the-fly temporal and Bayesian probabilistic fault diagnosis; (3) an unobtrusive, lightweight, read-only, low-power hardware realization using Field Programmable Gate Arrays (FPGAs) in order to avoid overburdening limited computing resources or costly re-certification of flight software due to instrumentation. No currently available SHM capabilities (or combinations of currently existing SHM capabilities) come anywhere close to satisfying these three criteria yet NASA will require such intelligent, hardwareenabled sensor and software safety and health management for introducing autonomous UAS into the National Airspace System (NAS). We propose a novel approach of creating modular building blocks for combining responsive runtime monitoring of temporal logic system safety requirements with model-based diagnosis and Bayesian network-based probabilistic analysis. Our proposed research program includes both developing this novel approach and demonstrating its capabilities using the NASA Swift UAS as a demonstration platform.

Contents

List of Figures

List of Tables

Chapter 1

Introduction

Unmanned Aerial Systems (UAS) can only be deployed if they can effectively complete their mission and respond to failures and uncertain environmental conditions while maintaining safety with respect to other aircraft as well as humans and property on the ground. In particular in situations, where there is no link to the ground or the UAS must fly autonomously, on-board software and hardware is responsible for controlling and guiding the UAS in a safe and effective manner. Failures and deviations must be detected and mitigation actions initiated.

In this project, we have designed a real-time, onboard system health management (SHM) capability to continuously monitor essential system components such as sensors, software, and hardware systems for detection and diagnosis of failures and violations of safety or performance rules during the flight of a UAS. To provide assurance that a UAS will not cause any harm during its missions, we designed a SHM framework that operates aboard a low-cost, dedicated, and separate FPGA (Field-Programmable Gate Array). 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 re-compilation and we use an intuitive, expressive specification language; we require real-time projections of Linear Temporal Logic (LTL) since operational concepts for UAS 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.

Our approach to modeling system and software health is three-pronged, based upon (1) real-time analysis of sensor and software signals, (2) advanced on-the-fly temporal processing, and (3) Bayesian probabilistic fault diagnosis. All the components of our framework have been integrated into an FPGA design; read-only data connections connect to the system buses and flight computer, assuring up-to-date data values while minimizing any interference to other components on-board the UAS.

Existing monitoring methods, like Runtime Verification (RV), assess the system status via software instrumentation and checking the current state against a formal specification. Because software instrumentation makes re-certification of the flight software onerous, alter the original timing behavior, or increase resource consumption, an RV approach is not feasible. In addition, RV only checks if a test against the specification has passed or failed.

In our approach developed within this projects, we go ways beyond RV by synergistically combining temporal logic and Bayesian diagnostic reasoning. UAS often need to adhere to timing-related rules like: R_1 :"*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). Mainly due to promising complexity results, restrictions of LTL to its past-time fragment have most often been used for RV. Though specifications including past time operators (e.g., "the IR sensor must have been powered up at least 5 minutes prior to takeoff") may be natural for some other domains, flight rules require future-time reasoning. We developed efficient temporal observer pairs to process LTL specifications containing past-time and future-time operators.

Often, a given failure situation might be attributed to different causes. For example, an erroneous altitude reading might have been caused by a faulty barometric altitude sensor (e.g., blocked Pitot tube), a malfunctioning laser altimeter, or some problem in the flight software. On-the-spot statistical diagnosis is important for root cause analysis: which component(s) most likely caused the current situation. For this kind of diagnostic reasoning, we are using Bayesian networks. We have developed a method for efficient reasoning inside the FPGA.

Each of the components of rt-R2U2 have been developed and put into an FPGA design. The FPGA hardware has been integrated—for testing and evaluation purposes—into a NASA DragonEye UAS.

This report is structured as follows: the next two chapters describe in detail the temporal processing (Chapter 2) and the Bayesian diagnostic reasoning (Chapter 3). Theoretical background, related work, implementation, and results of case studies on existing flight data of the NASA Swift UAS will be presented. These two chapters are pre-prints of publications [40] and [21]; thus the text contains some overlaps.

Chapter 4 covers the implementation of our framework onto a Parallella FPGA board and integration into the DragonEye UAS. This UAS has been equipped with the open-source ArduPlane flight software. For preparation of flight tests, we performed a detailed risk analysis, describe the hardware integration and initial flight tests.

Chapter 5 discusses future work and concludes. Several appendices contain correctness and complexity proofs for the temporal observers (Appendix A, B), detailed simulation results (Appendix C), details of the risk analysis (Appendix D), the list of monitored software variables (Appendix E), and a detailed list of publications and presentations of this project (Appendix F).

Chapter 2

Temporal-Logic Based Runtime Observer Pairs for System Health Management of Real-Time Systems [39]

2.1 Introduction

[1] Autonomous and automated systems, including Unmanned Aerial Systems (UAS), rovers, and satellites, have a large number of components, e.g., sensors, actuators, and software, that must function together reliably at mission time. System Health Management (SHM) [24] can detect, isolate, and diagnose faults and possibly initiate recovery activities on such real-time systems. Effective SHM requires assessing the status of the system with respect to its specifications and estimating system health during mission time. Johnson et al. [24, Ch.1] recently highlighted the need for new, formal-methods based capabilities for modeling complex relationships among different sensor data and reasoning about timing-related requirements; computational expense prevents the current best methods for SHM from meeting operational needs.

We need a new SHM framework for real-time systems like the Swift [23] electric UAS (see Fig. 2.1), developed at NASA Ames. SHM for such systems requires:

Responsiveness: the SHM framework must continuously monitor the system. Deviations from the monitored specifications must be detected within a tight and a priori known time bound, enabling mitigation or rescue measures, e.g., a controlled emergency landing to avoid damage on the ground. Reporting intermediate status and satisfaction of timed

[1] The material in this chapter is published in [39].

requirements as early as possible is required for enabling responsive decision-making.
UNOBTRUSIVENESS: the SHM framework must not alter crucial properties of the system including *functionality* (not change behavior), *certifiability* (avoid re-certification of flight software/hardware), *timing* (not interfere with timing guarantees), and *tolerances* (not violate size, weight, power, or telemetry bandwidth constraints). Utilizing commercial-off-the-shelf (COTS) and previously proven system components is absolutely required to meet today's tight time and budget constraints; adding the SHM framework to the system must not alter these components as changes that require them to be re-certified cancel out the benefits of their use. Our goal is to create the most effective SHM capability with the limitation of read-only access to the data from COTS components.
REALIZABILITY: the SHM framework must be usable in a plug-and-play manner by providing a generic interface to connect to a wide variety of systems. The specification language must be easily understood and expressive enough to encode e.g. temporal relationships and flight rules. The framework must adapt to new specifications without a lengthy re-compilation. We must be able to efficiently monitor different requirements during different mission stages, like takeoff, approach, measurement, and return.

2.1.1 Related Work

Existing methods for Runtime Verification (RV) [6] assess system status by automatically generating, mainly software-based, observers to check the state of the system against a formal specification. Observations in RV are usually made accessible via software instrumentation [22]; they report only when a specification has passed or failed. Such instrumentation violates our requirements as it may make re-certification of the system onerous, alter the original timing behavior, or increase resource consumption [36]. Also, reporting only the outcomes of specifications violates our responsiveness requirement.

Systems in our applications domain often need to adhere to timing-related rules like: *after receiving the command 'takeoff' reach an altitude of* $600ft$ *within five minutes.* These flight rules can be easily expressed in temporal logics; often in some flavor of linear temporal logic (LTL), as studied in [9]. Mainly due to promising complexity results [8, 16], restrictions of LTL to its past-time fragment have most often been used for RV. Though specifications including past time operators may be natural for some other domains [26], flight rules require future-time reasoning. To enable more intuitive specifications, others have studied monitoring of future-time claims; see [30] for a survey and [7, 16, 20, 29, 45, 46] for algorithms and frameworks. Most of these observer algorithms, however, were designed with a software implementation in mind and require a powerful computer. There are many hardware alternatives, e.g. [18], however all either resynthesize monitors from scratch or exclude checking real-time properties [4]. Our unique approach runs the logic synthesis tool once to synthesize as many real-time observer blocks as we can fit on our platform, e.g., FPGA or ASIC; our Sec. 2.4.1 only interconnects these blocks. Others have proposed using Bayesian inference techniques [15] to estimate the health of a system. However, modeling timing-related behavior with dynamic Bayesian networks is very complex and quickly renders practical implementations infeasible.

2.1.2 Approach and Contributions

We propose a new paired-observer SHM framework allowing systems like the Swift UAS to assess their status against a temporal logic specification while enabling advanced health estimation, e.g., via discrete Bayesian networks (BN) [15] based reasoning. This novel combination of two approaches, often seen as orthogonal to each other, enables us to check timing-related aspects with our paired observers while keeping BN health models free of timing information, and thus computationally attractive. Essentially, we can enable better real-time SHM by utilizing paired temporal observers to optimize BN-based decision making. Following our requirements, we call our new SHM framework for real-time systems a rt-R2U2 (real-time, Realizable, Responsive, Unobtrusive Unit).

Our rt-R2U2 synthesizes a pair of observers for a real-time specification φ given in Metric Temporal Logic (MTL) [2] or a specialization of LTL for mission-time bounded characteristics, which we define in Sec. 2.2. To ensure RESPONSIVENESS of our rt-R2U2, we design two kinds of observer algorithms in Sec. 2.3 that verify whether φ holds at a discrete time and run them in parallel. *Synchronous* observers have small hardware footprints (max. eleven two-input gates per operator; see Theorem 3 in Sec. 2.4) and return an instant, three-valued abstraction $\{\mathbf{true}, \mathbf{false}, \mathbf{maybe}\}$) of the satisfaction check of φ with every new tick of the Real Time Clock (RTC) while their *asynchronous* counterparts concretize this abstraction at a later, a priori known time. This unique approach allows us to signal early failure *and acceptance* of every specification whenever possible via the asynchronous observer. Note that previous approaches to runtime monitoring signal only specification failures; signaling *acceptance*, and particularly *early acceptance* is unique to our approach and required for supporting other system components such as prognostics engines or decision making units. Meanwhile, our synchronous observer's three-valued output gives intermediate information that a specification has not yet passed/failed, enabling probabilistic decision making via a Bayesian Network as described in [43].

We implement the rt-R2U2 in hardware as a self-contained unit, which runs externally to the system, to support UNOBTRUSIVENESS; see Sec. 2.4. Safety-critical embedded systems often use industrial, vehicle bus systems, such as CAN and PCI, interconnecting hardware and software components,

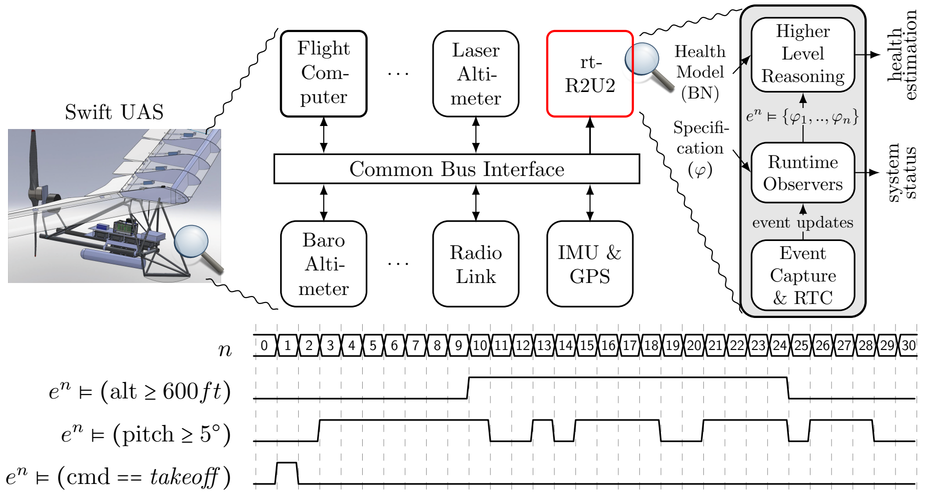


Figure 2.1: rt-R2U2: An instance of our SHM framework rt-R2U2 for the NASA Swift UAS. Swift subsystems (top): The laser altimeter maps terrain and determines elevation above ground by measuring the time for a laser pulse to echo back to the UAS. The barometric altimeter determines altitude above sea level via atmospheric pressure. The inertial measurement unit (IMU) reports velocity, orientation (yaw, pitch, and roll), and gravitational forces using accelerometers, gyroscopes, and magnetometers. Running example (bottom): predicates over Swift UAS sensor data on execution e; ranging over the readings of the barometric altimeter, the pitch sensor, and the takeoff command received from the ground station; n is the time stamp as issued by the Real-Time-Clock.

see Fig 2.1. Our rt-R2U2 provides generic read-only interfaces to these bus systems supporting our Unobtrusiveness requirement and sidestepping instrumentation. Events collected on these interfaces are time stamped by a RTC; progress of time is derived from the observed clock signal, resulting in a discrete time base $\mathbb{N}_0$. Events are then processed by our runtime observer pairs that check whether a specification holds on a sequence of collected events. Other RV approaches for on-the-fly observers exhibit high overhead [19, 27, 38] or use powerful database systems [5], thus, violate our requirements.

To meet our Realizability requirement, we design an efficient, highly parallel hardware architecture, yet keep it programmable to adapt to changes in the specification. Unlike existing approaches, our observers are designed with an efficient hardware implementation in mind, therefore, avoid recursion and expensive search through memory and aim at maximizing the benefits of the parallel nature of hardware. We synthesize rt-R2U2 *once* and generate a configuration, similar to machine code, to interconnect and configure the static hardware observer blocks of rt-R2U2, adapting to new specifications without running CAD or compilation tools like previous approaches. UAS have very limited bandwidth constraints; transferring a lightweight configuration is preferable to transferring a new image for the whole hardware design. The checks computed by these runtime observers represent the system's status and can be utilized by a higher level reasoner, such as a human operator, Bayesian network, or otherwise, to compute a health estimation, i.e., a conditional probability expressing the belief that a certain subsystem is healthy, given the status of the system. In this chapter, we compute these health estimations by adapting the BN-based inference algorithms of [15] in hardware. Our contributions include synthesis and integration of the synchronous/asynchronous observer pairs, a modular hardware implementation, and execution of a proof-of-concept rt-R2U2 running on a self-contained Field Programmable Gate Array (FPGA) (Sec. 2.5).

2.2 Real-time projections of LTL

MTL replaces the temporal operators of LTL with operators that respect time bounds [2].

Discrete-Time MTL 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 use the functions $\min, \max, \mathtt{dur}$, to extract the lower time bound (t), the upper time bound (t'), and the duration ($t' - t$) of J. We define the satisfaction relation of an MTL formula as follows: an execution $e = (s_n)$ for $n \geq 0$ is an infinite sequence of states. For an MTL formula φ, time $n \in \mathbb{N}_0$ and execution e, we define *φ holds at time n of execution e*, denoted $e^n \models \varphi$, inductively as follows:

$$\begin{array}{lll}
e^n \models \mathit{true} & & \text{is } \textbf{true}, \quad e^n \models \sigma \quad \text{iff} \quad \sigma \text{ holds in } s_n, \\
e^n \models \neg\varphi & \text{iff} & e^n \not\models \varphi, \quad e^n \models \mathcal{X}\,\varphi \quad \text{iff} \quad e^{n+1} \models \varphi, \\
e^n \models \varphi \wedge \psi & \text{iff} & e^n \models \varphi \text{ and } e^n \models \psi, \\
e^n \models \varphi\, \mathcal{U}_J\, \psi & \text{iff} & \exists i (i \geq n) : (i - n \in J \wedge e^i \models \psi \wedge \\
 & & \forall j (n \leq j < i) : \; e^j \models \varphi).
\end{array}$$

With the dualities $\Diamond_J\varphi \equiv \textbf{true}\, \mathcal{U}_J\, \varphi$ and $\neg\Diamond_J\neg\varphi \equiv \Box_J\, \varphi$ we arrive at two additional operators: $\Box_J\, \varphi$ (*φ is an invariant within the future interval J*) and $\Diamond_J\varphi$ (*φ holds eventually within the future interval J*). In order to efficiently encode specifications in practice, we introduce two special cases of $\Box_J\, \varphi$ and $\Diamond_J\varphi$: $\blacksquare_\tau\varphi \equiv \Box_{[0,\tau]}\, \varphi$ (*ϕ is an invariant within the next τ time units*) and $\blacklozenge_\tau\varphi \equiv \Diamond_{[0,\tau]}\varphi$ (*ϕ holds eventually within the next τ time units*). For example, the flight rule from Sec. 2.1, "After receiving the takeoff command reach an altitude of $600ft$ within five minutes," is efficiently captured in MTL by $(\text{cmd} == \mathit{takeoff}) \rightarrow \blacklozenge_5(\text{alt} \geq$

$600ft$), assuming a time-base of one minute and the atomic propositions (alt $\geq 600ft$) and (cmd $==$ *takeoff*) as in Fig. 2.1.

Systems in our application domain are usually bounded to a certain mission time. For example, the Swift UAS has a limited air-time, depending on the available battery capacity and predefined waypoints. We capitalize on this property to intuitively monitor standard LTL requirements using a mission-time bounded projection of LTL.

Mission-Time LTL For a given LTL formula ξ and a mission time $t_m \in \mathbb{N}_0$, we denote by ξ_m the mission-time bounded equivalent of ξ, where ξ_m is obtained by replacing every $\Box\varphi$, $\Diamond\varphi$, and $\varphi\,\mathcal{U}\,\psi$ operator in ξ by the $\Box_\tau\varphi$, $\Diamond_\tau\varphi$, and $\varphi\,\mathcal{U}_J\,\psi$ operators of MTL, where $J = [0, t_m]$ and $\tau = t_m$.

Inputs to rt-R2U2 are time-stamped events, collected incrementally from the system.

Execution Sequence 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 $v \in \{\mathbf{true}, \mathbf{false}, \mathbf{maybe}\}$ is a verdict.

We use a superscript integer to access a particular element in $\langle T_\varphi \rangle$, e.g., $\langle T_\varphi^0 \rangle$ is the first element in execution sequence $\langle T_\varphi \rangle$. We write $T_\varphi.\tau_e$ to access τ_e and $T_\varphi.v$ to access v of such an element. We say T_φ holds if $T_\varphi.v$ is **true** and T_φ does not hold if $T_\varphi.v$ is **false**. For a given execution sequence $\langle T_\varphi \rangle = \langle T_\varphi^0 \rangle, \langle T_\varphi^1 \rangle, \langle T_\varphi^2 \rangle, \langle T_\varphi^3 \rangle, \ldots$, the tuple accessed by $\langle T_\varphi^i \rangle$ corresponds to a section of an execution e as follows: for all times $n \in [\langle T_\varphi^{i-1} \rangle.\tau_e + 1, \langle T_\varphi^i \rangle.\tau_e]$, $e^n \models \varphi$ in case $\langle T_\varphi^i \rangle.v$ is **true** and $e^n \not\models \varphi$ in case $\langle T_\varphi^i \rangle.v$ is **false**. In case $\langle T_\varphi^i \rangle$ is **maybe**, neither $e^n \models \varphi$ nor $e^n \not\models \varphi$ is defined.

In the remainder of this report, we will frequently refer to execution sequences collected from the Swift UAS as shown in Fig. 2.1. The predicates shown are atomic propositions over sensor data in our specifications and are sampled with every new time stamp n issued by the RTC. For example, $\langle T_{\text{pitch} \geq 5^\circ} \rangle = ((\mathbf{false}, 0), (\mathbf{false}, 1), (\mathbf{false}, 2), (\mathbf{true}, 3), \ldots, (\mathbf{true}, 17), (\mathbf{true}, 18))$ describes $e^n \models (\text{pitch} \geq 5^\circ)$ sampled over $n \in [0, 18]$ and $\langle T_{\text{pitch} \geq 5^\circ} \rangle$ holds 19 elements.

2.3 Asynchronous and Synchronous Observers

The problem of monitoring a real-time specification has been studied extensively in the past; see [10, 30] for an overview. Solutions include: (a) translating the temporal formula into a finite-state automaton that accepts all the models of the specification [16, 18, 20, 46], (b) restricting MTL to its *safety* fragment and waiting until the operators' time bounds have elapsed to decide the truth value afterwards [7, 29], and (c) restricting LTL to its past-time fragment [8, 16, 38]. Compiling new observers to automata as in (a) requires re-running the logic synthesis tool to yield a new hardware observer, in automaton or autogenerated VHDL code format as described in [18], which may take dozens of minutes to complete, violating the Realizability requirement. Observers generated by (b) are in conflict with the Responsiveness requirement and (c) do not natively support flight rules. Our observers provide Unobtrusiveness via a self-contained hardware implementation. To enable such an implementation, our design needs to refrain from dynamic memory, linked lists, and recursion – commonly used in existing software-based observers, however, not natively available in hardware.

Our two types of runtime observers differ in the times when new outputs are generated and in the resource footprints required to implement them. A *synchronous* (time-triggered) observer is trimmed towards a minimalistic hardware footprint and computes a three-valued abstraction of the satisfaction check for the specification with each tick of the RTC, without considering events happening after the current time. An *asynchronous* (event-triggered) observer concretizes this abstraction at a later, a priori known, time and makes use of synchronization queues to take events into account that occur after the current time.[1] Our novel parallel composition of these two observers updates the status of the system at every tick of the RTC, yielding great responsiveness. An inconclusive answer when we can't yet know **true**/**false** is still beneficial as the higher-level reasoning part of our rt-R2U2 supports reasoning with inconclusive inputs. This allows us to derive an intermediate estimation of system health with the option to initiate fault mitigation actions even without explicitly knowing all inputs. If exact reasoning is required, we can re-evaluate system health when the *asynchronous* observer provides exact answers.

In the remainder of this section, we discuss[2] both *asynchronous* and *synchronous* observers for the operators $\neg\,\varphi$, $\varphi \wedge \psi$, ◧$_\tau\,\varphi$, $\Box_J\ \varphi$, and $\varphi\ \mathcal{U}_J\ \psi$. Informally, an MTL observer is an algorithm that takes execution sequences as input and produces another execution sequence as output. For a given unary operator $\bullet$, we say that an observer algorithm implements $e^n \models \bullet\,\varphi$, iff for all execution sequences $\langle T_\varphi\rangle$ as input, it produces an execution sequence as output that evaluates $e^n \models \bullet\,\varphi$ (analogous for binary operators).

2.3.1 Asynchronous Observers

The main characteristic of our asynchronous observers is that they are *evaluated with every new input tuple* and that for every generated output tuple T we have that $T.v \in \{\mathbf{true}, \mathbf{false}\}$ and $T.\tau_e \in [0, n]$. Since verdicts are exact evaluations of a future-time specification φ for each clock tick

[1]Similar terms have been used by others [12] to refer to monitoring with pairs of observers that do not update with the RTC, incur delays dangerous to a UAS, and require system interaction that violates our requirements (Sec. 2.1).

[2]Proofs of correctness for every observer algorithm appear in the Appendix.

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.

Our observers distinguish two types of transitions of the signals described by execution sequences. We say transition _/‾ of execution sequence $\langle T_\varphi \rangle$ occurs at time $n = \langle T_\varphi^i \rangle.\tau_e + 1$ iff $(\langle T_\varphi^i \rangle.v \oplus \langle T_\varphi^{i+1} \rangle.v) \wedge \langle T_\varphi^{i+1} \rangle.v$ holds. Similarly, we say transition ‾_ of execution sequence $\langle T_\varphi \rangle$ occurs at time $n = \langle T_\varphi^i \rangle.\tau_e + 1$ iff $(\langle T_\varphi^i \rangle.v \oplus \langle T_\varphi^{i+1} \rangle.v) \wedge \langle T_\varphi^i \rangle.v$ holds ($\oplus$ denotes the Boolean exclusive-or). For example, transitions _/‾ and ‾_ of $\langle T_{\text{pitch} \geq 5^\circ} \rangle$ in Fig. 2.1 occur at times 3 and 11, respectively.

2.3.1.1 Negation ($\neg\, \varphi$)

The observer for $\neg\, \varphi$, as stated in Alg. 7, is straightforward: for every input T_φ we negate the truth value of $T_\varphi.v$. The observer generates $(\ldots, (\textbf{true}, 2), (\textbf{false}, 3), \ldots)$.

2.3.1.2 Invariant within the Next τ Time Stamps (◧$_\tau\, \varphi$)

An observer for ◧$_\tau\, \varphi$ requires registers $m_{\uparrow\varphi}$ and m_{τ_s} with domain $\mathbb{N}_0$: $m_{\uparrow\varphi}$ holds the time stamp of the latest _/‾ transition of $\langle T_\varphi \rangle$ whereas m_{τ_s} holds the start time of the next tuple in $\langle T_\varphi \rangle$. For the observer in Alg. 8, the check $m \leq (T_\varphi.\tau_e - \tau)$ in line 8 tests whether φ held for at least the previous τ time stamps. To illustrate the algorithm, consider an observer for ◧$_5$ (pitch $\geq 5^\circ$) and the execution in Fig. 2.1. At time $n = 0$, we have $m_{\uparrow\varphi} = 0$ and since $\langle T^0_{\text{pitch} \geq 5^\circ} \rangle$ does not hold the output is $(\textbf{false}, 0)$. Similarly, the outputs for $n \in [1, 2]$ are $(\textbf{false}, 1)$ and $(\textbf{false}, 2)$. At time $n = 3$, a _/‾ transition of $\langle T_{\text{pitch} \geq 5^\circ} \rangle$ occurs, thus $m_{\uparrow\varphi} = 3$. Since the check in line 8 does not hold, the algorithm does not generate a new output, i.e., returns $(\llcorner\!\lrcorner, \llcorner\!\lrcorner)$ designating output is delayed until a later time, which repeats at times $n \in [4, 7]$. At $n = 8$, the check in line 8 holds and the algorithm returns $(\textbf{true}, 3)$. Likewise, the outputs for $n \in [9, 10]$ are $(\textbf{true}, 4)$ and $(\textbf{true}, 5)$. At $n = 11$, $\langle T^{11}_{\text{pitch} \geq 5^\circ} \rangle$ does not hold and the algorithm outputs $(\textbf{false}, 11)$. We note the ability of the observer to *re-synchronize* its output with respect to its inputs and the RTC. For $n \in [8, 10]$, outputs are given for a time prior to n, however, at $n = 11$ the observer re-synchronizes: the output $(\textbf{false}, 11)$ signifies that $e^n \not\models$ ◧$_5$ (pitch $\geq 5^\circ$) for $n \in [6, 11]$. By the equivalence ⬖$_\tau\, \varphi \equiv \neg$◧$_\tau \neg\varphi$, we immediately arrive at an observer for ⬖$_\tau\, \varphi$ from Alg. 8 by negating both the input and the output tuple.

2.3.1.3 Invariant within Future Interval ($\Box_J\, \varphi$)

The observer for $\Box_J\, \varphi$, as stated in Alg. 9, builds on an observer for ◧$_\tau\, \varphi$ and makes use of the equivalence ◧$_\tau \varphi \equiv \Box_{[0,\tau]}\, \varphi$. Intuitively, the observer for ◧$_\tau\, \varphi$ returns true iff φ holds for at least the next τ time units. We can thus construct an observer for $\Box_J\, \varphi$ by reusing the algorithm for ◧$_\tau\, \varphi$, assigning $\tau = \texttt{dur}(J)$ and shifting the obtained output by $\min(J)$ time stamps into the past. From the equivalence $\Diamond_J \varphi \equiv \neg \Box_J\, \neg\varphi$, we can immediately derive an observer for $\Diamond_J\, \varphi$ from the observer for $\Box_J\, \varphi$. To illustrate the algorithm, consider an observer for $\Box_{5,10}$ (alt $\geq 600ft$) over the execution in Fig. 2.1. For $n \in [0,4]$ the algorithm returns $(\llcorner\!\lrcorner, \llcorner\!\lrcorner)$, since $(\langle T^{0\ldots4}_{\mathrm{alt}\geq 600ft}\rangle.\tau_e - 5) \geq 0$ (line 3 of Alg. 9) does not hold. At $n = 5$ the underlying observer for $\blacksquare_5$ (alt $\geq 600ft$) returns $(\mathbf{false}, 5)$, which is transformed (by line 4) into the output $(\mathbf{false}, 0)$. For similar arguments, the outputs for $n \in [6,9]$ are $(\mathbf{false}, 1)$, $(\mathbf{false}, 2)$, $(\mathbf{false}, 3)$, and $(\mathbf{false}, 4)$. At $n \in [10,14]$, the observer for $\blacksquare_5$ (alt $\geq 600ft$) returns $(\llcorner\!\lrcorner, \llcorner\!\lrcorner)$. At $n = 15$, $\blacksquare_5$ (alt $\geq 600ft$) yields $(\mathbf{true}, 10)$, which is transformed (by line 4) into the output is $(\mathbf{true}, 5)$. Note also that $\mathcal{X}\varphi \equiv \Box_{[1,1]}\, \varphi$.

The remaining observers for the binary operators $\varphi \wedge \psi$ and $\varphi\, \mathcal{U}_J\, \psi$ take tuples (T_φ, T_ψ) as inputs, where T_φ is from $\langle T_\varphi\rangle$ and T_ψ is from $\langle T_\psi\rangle$. Since $\langle T_\varphi\rangle$ and $\langle T_\psi\rangle$ are execution sequences produced by two different observers, the two elements of the input tuple (T_φ, T_ψ) are not necessarily generated at the same time. Our observers for binary MTL operators thus use two FIFO-organized *synchronization queues* to buffer parts of $\langle T_\varphi\rangle$ and $\langle T_\psi\rangle$, respectively. For a synchronization queue q we denote by $q = ()$ its emptiness and by $|q|$ its size.

Algorithm 1 Observer for $\neg\,\varphi$.

```
1: At each new input $T_\varphi$:
2: $T_\xi \leftarrow (\neg\, T_\varphi.v, T_\varphi.\tau_e)$
3: return $T_\xi$
```

Algorithm 2 Observer for $◧_\tau\,\varphi$. Initially, $m_{\uparrow\varphi} = m_{\tau_s} = 0$.

```
1: At each new input $T_\varphi$:
2: $T_\xi \leftarrow T_\varphi$
3: if _|‾ transition of $T_\xi$ occurs then
4:     $m_{\uparrow\varphi} \leftarrow m_{\tau_s}$
5: end if
6: $m_{\tau_s} \leftarrow T_\varphi.\tau_e + 1$
7: if $T_\xi$ holds then
8:     if $m_{\uparrow\varphi} \leq (T_\xi.\tau_e - \tau)$ holds then
9:         $T_\xi.\tau_e \leftarrow T_\xi.\tau_e - \tau$
10:    else
11:        $T_\xi \leftarrow (␣, ␣)$
12:    end if
13: end if
14: return $T_\xi$
```

Algorithm 3 Observer for $\varphi \wedge \psi$.

```
1: At each new input $(T_\varphi, T_\psi)$:
2: if $T_\varphi$ holds and $T_\psi$ holds and $q_\varphi \neq ()$
   holds and $q_\psi \neq ()$ holds then
3:     $T_\xi \leftarrow (\textbf{true}, \min(T_\varphi.\tau_e, T_\psi.\tau_e))$
4: else if $\neg T_\varphi$ holds and $\neg T_\psi$ holds and
   $q_\varphi \neq ()$ holds and $q_\psi \neq ()$ holds then
5:     $T_\xi \leftarrow (\textbf{false}, \max(T_\varphi.\tau_e, T_\psi.\tau_e))$
6: else if $\neg T_\varphi$ holds and $q_\varphi \neq ()$ holds
   then
7:     $T_\xi \leftarrow (\textbf{false}, T_\varphi.\tau_e)$
8: else if $\neg T_\psi$ holds and $q_\psi \neq ()$ holds
   then
9:     $T_\xi \leftarrow (\textbf{false}, T_\psi.\tau_e)$
10: else
11:    $T_\xi \leftarrow (␣, ␣)$
12: end if
13: dequeue$(q_\varphi, q_\psi, T_\xi.\tau_e)$
14: return $T_\xi$
```

Algorithm 4 Observer for $\Box_J\,\varphi$.

```
1: At each new input $T_\varphi$:
2: $T_\xi \leftarrow ◧_{\texttt{dur}(J)}\, T_\varphi$
3: if $(T_\xi.\tau_e - \min(J) \geq 0)$ then
4:     $T_\xi.\tau_e \leftarrow T_\xi.\tau_e - \min(J)$
5: else
6:     $T_\xi \leftarrow (␣, ␣)$
7: end if
8: return $T_\xi$
```

Algorithm 5 Observer for $\varphi\,\mathcal{U}_J\,\psi$. Initially, $m_{pre} = m_{\uparrow\varphi} = 0$, $m_{\downarrow\varphi} = -\infty$, and $p = \textbf{false}$.

```
1: At each new input $(T_\varphi, T_\psi)$ in lockstep
   mode:
2: if _|‾ transition of $T_\varphi$ occurs then
3:     $m_{\uparrow\varphi} \leftarrow \tau_e - 1$
4:     $m_{pre} \leftarrow -\infty$
5: end if
6: if ‾|_ transition of $T_\varphi$ occurs and $T_\psi$
   holds then
7:     $T_\varphi.v$, $p \leftarrow \textbf{true}, \textbf{true}$
8:     $m_{\downarrow\varphi} \leftarrow \tau_e$
9: end if
10: if $T_\varphi$ holds then
11:     if $T_\psi$ holds then
12:         if $(m_{\uparrow\varphi} + \min(J) < \tau_e)$ holds
   then
13:             $m_{pre} \leftarrow \tau_e$
14:             return $(\textbf{true}, \tau_e - \min(J))$
15:         else if $p$ holds then
16:             return $(\textbf{false}, m_{\downarrow\varphi})$
17:         end if
18:     else if $(m_{pre} + \texttt{dur}(J) \leq \tau_e)$ holds
   then
19:         return $(\textbf{false}, \max(m_{\uparrow\varphi}, \tau_e -
   \max(J)))$
20:     end if
21: else
22:     $p \leftarrow \textbf{false}$
23:     if $(\min(J) = 0)$ holds then
24:         return $(T_\psi.v, \tau_e)$
25:     end if
26:     return $(\textbf{false}, \tau_e)$
27: end if
28: return $(␣, ␣)$
```

2.3.1.4 Conjunction ($\varphi \wedge \psi$)

The observer for $\varphi \wedge \psi$, as stated in Alg. 10, reads inputs (T_φ, T_ψ) from two synchronization queues, q_φ and q_ψ. Intuitively, the algorithm follows the rules for conjunction in Boolean logic with additional emptiness checks on q_φ and q_ψ. The procedure **dequeue**$(q_\varphi, q_\psi, T_\xi.\tau_e)$ drops all entries T_φ in q_φ for which the following holds: $T_\varphi.\tau_e \leq T_\xi.\tau_e$ (analogous for q_ψ). To illustrate the algorithm, consider an observer for ◧$_5$ (alt $\geq 600ft$)$\wedge$(pitch $\geq 5°$) and the execution in Fig. 2.1. For $n \in [0, 9]$ the two observers for the involved subformulas immediately output $(\textbf{false}, n)$. For $n \in [10, 14]$, the observer for ◧$_5$ (alt $\geq 600ft$) returns $(␣, ␣)$, while in the meantime, the atomic proposition (pitch $\geq 5°$) toggles its truth value several times, i.e., $(\textbf{true}, 10)$, $(\textbf{false}, 11)$, $(\textbf{false}, 12)$, $(\textbf{true}, 13)$, $(\textbf{false}, 14)$. These tuples need to be buffered in queue $q_{\text{pitch}\geq 5°}$ until the observer for ◧$_5$ (alt $\geq 600ft$) generates its next output, i.e., $(\textbf{true}, 10)$ at $n = 15$. We apply the function **aggregate**$(\langle T_\varphi \rangle)$, which repeatedly replaces two consecutive elements $\langle T^i_\varphi \rangle, \langle T^{i+1}_\varphi \rangle$ in $\langle T_\varphi \rangle$ by $\langle T^{i+1}_\varphi \rangle$ iff $\langle T^i_\varphi \rangle.v = \langle T^{i+1}_\varphi \rangle.v$, to the content of $q_{\text{pitch}\geq 5°}$ once every time an element is added to $q_{\text{pitch}\geq 5°}$. Therefore, at $n = 15$: $q_{\text{pitch}\geq 5°} = ((\textbf{true}, 10), (\textbf{false}, 12), (\textbf{true}, 13), (\textbf{false}, 14), (\textbf{true}, 15))$ and $q_{◧_5\ (\text{alt}\geq 600ft)} = ((\textbf{true}, 10))$. The observer returns $(\textbf{true}, 10)$ (line 3) and **dequeue**$(q_\varphi, q_\psi, 10)$ yields: $q_{\text{pitch}\geq 5°} = ((\textbf{false}, 12), (\textbf{true}, 13), (\textbf{false}, 14), (\textbf{true}, 15))$ and $q_{◧_5\ (\text{alt}\geq 600ft)} = ()$.

2.3.1.5 Until within Future Interval ($\varphi\, \mathcal{U}_J\, \psi$)

The observer for $\varphi\, \mathcal{U}_J\, \psi$, as stated in Alg. 12, reads inputs (T_φ, T_ψ) from two synchronization queues and makes use of a Boolean flag p and three registers $m_{\uparrow\varphi}$, $m_{\downarrow\varphi}$, and m_{pre} with domain $\mathbb{N}_0 \cup \{-\infty\}$: $m_{\uparrow\varphi}$ ($m_{\downarrow\varphi}$) holds the time stamp of the latest _/‾ transition (‾_ transition) of $\langle T_\varphi \rangle$ and m_{pre} holds the latest time stamp where the observer detected $\varphi\, \mathcal{U}_J\, \psi$ to hold. Input tuples (T_φ, T_ψ) for the observer are read from synchronization queues in a *lockstep* mode: (T_φ, T_ψ) is split into (T'_φ, T'_ψ), where $T'_\varphi.\tau_e = T'_\psi.\tau_e$ and the time stamp $T''_\varphi.\tau_e$ of the next tuple (T''_φ, T''_ψ) is $T'_\varphi.\tau_e{+}1$. This ensures that the observer outputs only a single tuple at each run and avoids output buffers, which would account for additional hardware resources (see correctness proof in the Appendix for a discussion). Intuitively, if T_φ does not hold (lines 22-26) the observer is synchronous to its input and immediately outputs $(\textbf{false}, T_\varphi.\tau_e)$. If T_φ holds (lines 11-20) the time stamp n' of the output tuple is not necessarily *synchronous* to the time stamp $T_\varphi.\tau_e$ of the input anymore, however, bounded by $(T_\varphi.\tau_e - \max(J)) \leq n' \leq T_\varphi.\tau_e$ (see Lemma "*unrolling*" in the Appendix). To illustrate the algorithm, consider an observer for (pitch $\geq 5°$) $\mathcal{U}_{[5,10]}$ (alt $\geq 600ft$) over the execution in Fig. 2.1. At time $n = 0$, we have $m_{pre} = 0$, $m_{\uparrow\varphi} = 0$, and $m_{\downarrow\varphi} = -\infty$ and since $\langle T^0_{\text{pitch}\geq 5°} \rangle$ does not hold, the observer outputs $(\textbf{false}, 0)$ in line 26. The outputs for $n \in [1, 2]$ are $(\textbf{false}, 1)$ and $(\textbf{false}, 2)$. At time $n = 3$, a _/‾ transition of $\langle T_{\text{pitch}\geq 5°} \rangle$ occurs, thus we assign $m_{\uparrow\varphi} = 2$ and $m_{pre} = -\infty$ (lines 3 and 4). Since $\langle T^3_{\text{pitch}\geq 5°} \rangle$ holds and $\langle T^{3}_{\text{alt}\geq 600ft}\rangle$ does not hold, the predicate in line 18 is evaluated, which holds and the algorithm returns $\langle \textbf{false}, \max(2, 3-10)\rangle = (\textbf{false}, 2)$. Thus, the observer does not yield a new output in this case, which repeats for times $n \in [4, 9]$. At time $n = 10$, a $\_\!/\!\overline{\;}$ transition of $\langle T_{\text{alt}\geq 600ft}\rangle$ occurs and the predicate in line 12 is evaluated. Since $(2+5) < 10$ holds, the algorithm returns $(\textbf{true}, 5)$, revealing that $e^n \models (\text{pitch} \geq 5^\circ)\ \mathcal{U}_{[5,10]}\ (\text{alt} \geq 600ft)$ for $n \in [3, 5]$. At time $n = 11$, a $\overline{\;}\!\backslash\!\_$ transition of $\langle T_{\text{pitch}\geq 5^\circ}\rangle$ occurs and since $\langle T^{11}_{\text{alt}\geq 600ft}\rangle$ holds, p and the truth value of the current input $\langle T^{11}_{\text{pitch}\geq 5^\circ}\rangle.v$ are set **true** and $m_{\downarrow\varphi} = 11$. Again, line 12 is evaluated and the algorithm returns $(\textbf{true}, 6)$. At time $n = 12$, since $\langle T^{12}_{\text{pitch}\geq 5^\circ}\rangle$ does not hold, we clear p in line 22 and the algorithm returns $(\textbf{false}, 12)$ in line 26, i.e., $e^n \not\models (\text{pitch} \geq 5^\circ)\ \mathcal{U}_{[5,10]}\ (\text{alt} \geq 600ft)$ for $n \in [7, 12]$. At time $n = 13$, a $\_\!/\!\overline{\;}$ transition of $\langle T_{\text{pitch}\geq 5^\circ}\rangle$ occurs, thus $m_{\uparrow\varphi} = 12$ and $m_{pre} = -\infty$. The predicates in line 12 and 15 do not hold, the algorithm returns no new output in line 28. At time $n = 14$, a $\overline{\;}\!\backslash\!\_$ transition of $\langle T_{\text{pitch}\geq 5^\circ}\rangle$ occurs, thus p and $\langle T^{14}_{\text{pitch}\geq 5^\circ}\rangle.v$ are set **true** and $m_{\downarrow\varphi} = 14$. The predicate in line 15 holds, and the algorithm outputs $(\textbf{false}, 14)$, revealing that $e^n \not\models (\text{pitch} \geq 5^\circ)\ \mathcal{U}_{[5,10]}\ (\text{alt} \geq 600ft)$ for $n \in [13, 14]$.

2.3.2 Synchronous Observers

The main characteristic of our synchronous observers is that they are evaluated at every tick of the RTC and that 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. Inputs and outputs of these observers are execution sequences with three-valued verdicts. The underlying abstraction is given by $\widehat{\textbf{eval}} : \boxast \to \{\textbf{true}, \textbf{false}, \textbf{maybe}\}$, where $\boxast \in \{\neg\,\varphi, \varphi \wedge \psi, ◧_\tau\,\varphi, \Box_J\ \varphi, \varphi\ \mathcal{U}_J\ \psi\}$. The implementation of $\widehat{\textbf{eval}}\,(\neg\,\varphi)$ and $\widehat{\textbf{eval}}\,(\varphi \wedge \psi)$ follows the rules for Kleene logic [25]. For the remaining operators we define the verdict $T_\xi.v$ of the output tuple $(T_\xi.v, n)$, generated for inputs $(T_\varphi.v, n)$ (respectively $(T_\psi.v, n)$ for $\varphi\,\mathcal{U}_J\ \psi$), as:

$$\widehat{\textbf{eval}}\,(◧_\tau\,\varphi) = \begin{cases} \textbf{true} & \text{if} \quad T_\varphi.v \text{ holds and } \tau = 0, \\ \textbf{false} & \text{if} \quad T_\varphi.v \text{ does not hold,} \\ \textbf{maybe} & \qquad \text{otherwise.} \end{cases}$$

$$\widehat{\textbf{eval}}\,(\Box_J\ \varphi) = \textbf{maybe}.$$

$$\widehat{\textbf{eval}}\,(\varphi\,\mathcal{U}_J\ \psi) = \begin{cases} \textbf{true} & \text{if} \quad \begin{array}{l} T_\varphi.v \text{ and } T_\psi.v \text{ holds} \\ \text{and } \min(J) = 0, \end{array} \\ \textbf{false} & \text{if} \quad T_\varphi.v \text{ does not hold,} \\ \textbf{maybe} & \qquad \text{otherwise.} \end{cases}$$

To illustrate our synchronous observer algorithms, consider the previously discussed formula $◧_5\ (\text{alt} \geq 600ft) \wedge (\text{pitch} \geq 5^\circ)$, which we want to evaluate using the synchronous observer:

$$\xi = \widehat{\textbf{eval}}\,(\widehat{\textbf{eval}}\,(◧_5\ (\text{alt} \geq 600ft)) \wedge (\text{pitch} \geq 5^\circ))$$

For $n \in [0, 9]$, as in the case of the *asynchronous* observer, we can immediately output $(\mathbf{false}, n)$. At $n = 10$, $\widehat{\mathbf{eval}}\,(\blacksquare_5 \text{ (alt} \geq 600ft))$ yields $(\mathbf{maybe}, n)$, thus, the observer is inconclusive about the truth value of $e^{10} \models \xi$. At $n \in [11, 12]$ since (pitch $\geq 5°$) does not hold, the outputs are $(\mathbf{false}, n)$. For analogous arguments, the output at $n = 13$ is $(\mathbf{maybe}, 13)$, at $n = 14$ $(\mathbf{false}, 14)$, and at $n = 15$ $(\mathbf{maybe}, 15)$. In this way, at times $n \in \{11, 12, 14\}$ the synchronous observer completes early evaluation of ξ, producing output that would, without the abstraction, be guaranteed by the exact asynchronous observer with a delay of 5 time units, i.e., at times $n \in \{16, 17, 19\}$.

2.4 Mapping Observers into Efficient Hardware

We introduce a mapping of the observer pairs into efficient hardware blocks and a synthesis procedure to generate a configuration for these blocks from an arbitrary MTL specification. This configuration is loaded into the control unit of our rt-R2U2, where it changes the interconnections between a pool of (static) hardware observer blocks and assigns memory regions for synchronization queues. This approach enables us to quickly change the monitored specification (within resource limitations) without re-compiling the rt-R2U2's hardware, supporting our REALIZABILITY requirement.

Asynchronous observers require arithmetic operations on time stamps. Registers and flags as required by the observer algorithm are mapped to circuits that can store information, such as flip-flops. For the synchronization queues we turn to block RAMs (abundant on FPGAs), organized as ring buffers. Time stamps are internally stored in registers of width $w = \lceil log_2(n) \rceil + 2$, to indicate $-\infty$ and to allow overflows when performing arithmetical operations on time stamps. Subtraction and relational operators as required by the observer for $\blacksquare_\tau\, \varphi$ (Fig. 2.2) can be built around adders. For example, the check in line 8 of Alg. 8 is implemented using two w-bit wide adders: one for $q = T_\varphi.\tau_e - \tau$ and one to decide whether $m_{\uparrow\varphi} \geq q$. A third adder runs in parallel and assigns a new value to m_{τ_s} (line 6 of Alg. 8). Detecting a _/‾ transition on $\langle T_\varphi \rangle$ maps to an `XOR` gate and an `AND` gate, implementing the circuit $(T_\varphi^{i-1}.v \oplus T_\varphi^{i}.v) \wedge T_\varphi^{i}.v$, where $T_\varphi^{i-1}.v$ is the truth value of the previous input, stored in a flip-flop. The multiplexer either writes a new output or sets a flag to indicate (␣, ␣).

Synchronous observers do not require calculations on time stamps and directly map to basic digital logic gates. Fig. 2.2 shows a circuit representing an $\widehat{\mathbf{eval}}\,(\blacksquare_\tau\, \varphi)$ observer that accounts for one two-input `AND` gate, one two-input `OR` gate, and two Inverter gates. Inputs (i_1, i_2) and outputs (y_1, y_2) are encoded (to project the three-valued logic into Boolean logic) such as: $\mathbf{true}\,(0, 0)$, $\mathbf{false}\,(0, 1)$, and $\mathbf{maybe}\,(1, 0)$. Input j is set if $\tau_e = 0$ and cleared otherwise.

Algorithm 6 Assigning synchronization queue sizes for $\mathrm{AST}(\xi')$. Let S be a set of nodes; Initially: $w = 0$, add all Σ nodes of $\mathrm{AST}(\xi')$ to S; The function $\mathrm{wcd} : \boxast \to \mathbb{N}_0$ calculates the *worst-case-delay* an asynchronous observer may introduce by: $\mathrm{wcd}(\neg\,\varphi) = \mathrm{wcd}(\varphi \wedge \psi) = 0$, $\mathrm{wcd}(\text{◧}_\tau\,\varphi) = \tau$, $\mathrm{wcd}(\Box_J\ \varphi) = \mathrm{wcd}(\varphi\,\mathcal{U}_J\ \psi) = \max(J)$.

```
1: while S is not empty do
2:     $s, w \leftarrow$ get next node from $S$, 0
3:     if $s$ is type $\varphi\,\mathcal{U}_J\ \psi$ or $\varphi \wedge \psi$ then
4:         $w \leftarrow \max(|q_\varphi|, |q_\psi|) + \mathrm{wcd}(s)$
5:     end if
6:     while $s$ is not a synchronization queue do
7:         $s, w \leftarrow$ get predecessor of $s$ in $\mathrm{AST}(\xi')$, $w + \mathrm{wcd}(s)$
8:     end while
9:     Set $|q| = w$; ($q$ is opposite synchronization queue of $s$)
10:    Add all $\varphi\,\mathcal{U}_J\ \psi$ and $\varphi \wedge \psi$ nodes that have unassigned synchro-
       nization queue sizes to $S$
11: end while
```

2.4.1 Synthesizing a Configuration for the rt-R2U2

The synthesis procedure to translate an MTL specification ξ into a configuration such that the rt-R2U2 instantiates observers for both ξ and $\widehat{\mathbf{eval}}\,(\xi)$, works as follows:

- Preprocessing. By the equivalences given in Sect. 2.2 rewrite ξ to ξ', such that operators in ξ' are from $\{\neg\,\varphi, \varphi \wedge \psi, \text{◧}_\tau\,\varphi, \Box_J\ \varphi, \varphi\,\mathcal{U}_J\ \psi\}$ (**SA1**).

- Parsing. Parse ξ' to obtain an Abstract Syntax Tree (AST), denoted by $\mathrm{AST}(\xi')$. The leaves of this tree are the atomic propositions Σ of ξ' (**SA2**).

- Allocating observers. For all nodes q in $\mathrm{AST}(\xi')$ allocate both the corresponding synchronous and the asynchronous hardware observer block (**SA3**).

- Adding synchronization queues. $\forall q \in \mathrm{AST}(\xi')$: If q is of type $\varphi \wedge \psi$ or $\varphi\ \mathcal{U}_J\ \psi$ add queues q_φ and q_ψ to the inputs of the respective asynchronous observer (**MA1**).

- Interconnect and dimensioning. Connect observers and queues according to $\mathrm{AST}(\xi')$. Execute Alg. 6 (**MA2**).

Let $\{\sigma_1, \sigma_2, \sigma_3\} \in \Sigma$ and $\xi = \sigma_1 \to (\text{⬖}_{10}\ (\sigma_2) \vee \text{⬖}_{100}(\sigma_3))$ be an MTL formula we want to synthesize a configuration for. SA1 yields $\xi' = \neg(\sigma_1 \wedge \neg(\neg\text{◧}_{10}\ (\neg\sigma_2)) \wedge \neg(\neg\text{◧}_{100}\ (\neg\sigma_3)))$ which simplifies to $\xi = \neg(\sigma_1 \wedge \text{◧}_{10}\ (\neg\sigma_2) \wedge \text{◧}_{100}\ (\neg\sigma_3))$. SA2 yields $\mathrm{AST}(\xi')$. SA3 instantiates two $\varphi \wedge \psi$, three $\neg\,\varphi$, one $\text{◧}_{10}\ T_\varphi$ and one $\text{◧}_{100}\ T_\varphi$ observers, both synchronous and asynchronous. MA1, introduces queues $q_{\sigma_1}, q_{\xi_2}, q_{\xi_3}, q_{\xi_4}$

and MA2 interconnects observers and queues and assigns $|q_{\sigma_1}| = 100$, $|q_{\xi_2}| = 100$, $|q_{\xi_3}| = 10$, and $|q_{\xi_4}| = 0$, see Fig. 2.2.

2.4.2 Circuit Size and Depth Complexity Results

Having discussed how to determine the size of the synchronization queues for our asynchronous MTL observers, we are now in the position to prove space and time complexity bounds.

Theorem 2.41 (Space Complexity of Asynchronous Observers)

The respective asynchronous observer for a given MTL specification φ has a space complexity, in terms of memory bits, bounded by $(2 + \lceil \log_2(n) \rceil) \cdot (2 \cdot m \cdot p)$, where m is the number of binary observers (i.e., $\varphi \wedge \psi$ or $\varphi \mathcal{U}_J \psi$) in φ, p is the worst-case delay of a single predecessor chain in $\mathrm{AST}(\varphi)$, and $n \in \mathbb{N}_0$ is the time stamp it is executed.

Theorem 2.42 (Time Complexity of Asynchronous Observers)

The respective asynchronous observer for a given MTL specification φ has an asymptotic time complexity of $\mathcal{O}\Big(\log_2 \log_2 \max(p, n) \cdot d\Big)$, where p is the maximum worst-case-delay of any observer in $\mathrm{AST}(\varphi)$, d the depth of $AST(\varphi)$, and $n \in \mathbb{N}_0$ the time stamp it is executed.

For our synchronous observers, we prove upper bounds in terms of two-input gates on the size of resulting circuits. Actual implementations may yield significant better results on circuit size, depending on the performance of the logic synthesis tool.

Theorem 2.43 (Circuit-Size Complexity of Synchronous Observers)
For a given MTL formula φ, the circuit to monitor $\widehat{\mathbf{eval}}(\varphi)$ has a circuit-size complexity bounded by $11 \cdot m$, where m is the number of observers in $AST(\varphi)$.

Theorem 2.44 (Circuit-Depth Complexity of Synchronous Observers)
For a given MTL formula φ, the circuit to monitor $\widehat{\mathbf{eval}}(\varphi)$ has a circuit-depth complexity of $4 \cdot d$.

2.5 Applying the rt-R2U2 to NASA's Swift UAS

We implemented our rt-R2U2 as a register-transfer-level VHDL hardware design, which we simulated in Mentor Graphics ModelSim and synthesized for different FPGAs using the industrial logic synthesis tool Altera Quartus II.[3] With our rt-R2U2, we analyzed raw flight data from NASA's Swift UAS collected during test flights. The higher-level

[3]Simulation traces are available in the Appendix; tools can be downloaded at `http://www.mentor.com` and `http://www.altera.com`.

reasoning is performed by a *health model*, modeled as a Bayesian network (BN) where the nodes correspond to discrete random variables. Fig. 2.3 shows the relevant excerpt for reasoning about altitude. Directed edges encode conditional dependencies between variables, e.g., the sensor reading S_L depends on the health of the laser altimeter sensor H_L. Conditional probability tables at each node define the local dependencies. During health estimation, verdicts computed by our observer algorithms are provided as virtual sensor values to the observable nodes S_L, S_B, S_S; e.g., the laser altimeter measuring an altitude increase would result in setting S_L to state *inc*. Then, the posteriors of the multivariate probability distribution encoded in the BN are calculated [15]; for details of modeling and reasoning see [21, 42].

Our temporal specifications are evaluated by our runtime observers and describe flight rules (φ_1, φ_2) and virtual sensors:

$$\begin{aligned} \varphi_1 &= (\text{cmd} == \textit{takeoff}) \rightarrow \Diamond_{10}\,(\text{alt}_B \geq 600ft) \\ \varphi_2 &= (\text{cmd} == \textit{takeoff}) \rightarrow \Diamond_{*}\,(\text{cmd} == \textit{land}) \end{aligned}$$

φ_1 encodes our running example flight rule; φ_2 is a mission-bounded LTL property requiring that the command *land* is received after *takeoff*, within the projected mission time, indicated by $*$. Fig. 2.3 shows the execution sequences produced by both the asynchronous ($e^n \models \varphi_1$) and the synchronous ($e^n \models \widehat{\mathbf{eval}}\,(\varphi_1)$) observers for flight rule φ_1. To keep the presentation accessible we scaled the timeline to just 24 time stamps; the actual implementation uses a resolution of 2^{32} time stamps. The synchronous observer is able to prove the validity of φ_1 immediately at all time stamps but one ($n = 1$), where the output is $(\mathbf{maybe}, 1)$, indicated by 〰. The asynchronous observer will resolve this inconclusive output at time $n = 11$, by generating the tuple $(\mathbf{false}, 1)$, revealing a violation of φ at time $n = 1$. The verdicts of $\sigma_{S_{L\uparrow}}$,$\sigma_{S_{L\downarrow}}$, $\sigma_{S_{B\uparrow}}$, $\sigma_{S_{B\downarrow}}$, $\varphi_{S_{S\uparrow}}$, and $\varphi_{S_{S\downarrow}}$ are mapped to inputs S_L, S_B, S_S of the health model:

$$\begin{aligned} \sigma_{S_{L\uparrow}} &= (\text{alt}_L - \text{alt}'_L) > 0 & \sigma_{S_{L\downarrow}} &= (\text{alt}_L - \text{alt}'_L) < 0 \\ \sigma_{S_{B\uparrow}} &= (\text{alt}_B - \text{alt}'_B) > 0 & \sigma_{S_{B\downarrow}} &= (\text{alt}_B - \text{alt}'_B) < 0 \end{aligned}$$

$\sigma_{S_{B\uparrow}}$ observes if the first derivation of the barometric altimeter reading is positive, thus, holds if the sensors values indicate that the UAS is ascending. We set S_B to *inc* if $\sigma_{S_{B\uparrow}}$ holds and to *dec* if $\sigma_{S_{B\downarrow}}$ holds. The specifications $\varphi_{S_{S\uparrow}}$ and $\varphi_{S_{S\downarrow}}$ subsume the pitch and the velocity readings to an additional, indirect altitude sensor. Due to sensor noise, simple threshold properties on the IMU signals would yield a large number of false positives. Instead $\varphi_{S_{S\uparrow}}$ and $\varphi_{S_{S\downarrow}}$ use $\Box_\tau\,\varphi$ observers as filters, by requiring that the pitch and the velocity signals exceed a threshold for multiple time steps.

$$\begin{aligned} \varphi_{S_{S\uparrow}} &= \Box_{10}\,(\text{pitch} \geq 5^\circ) \;\wedge\; \Box_5\,(\text{vel\_up} \geq 2\tfrac{m}{s}) \\ \varphi_{S_{S\downarrow}} &= \Box_{10}\,(\text{pitch} < 2^\circ) \;\wedge\; \Box_5\,(\text{vel\_up} \leq -2\tfrac{m}{s}) \end{aligned}$$

Our real-time SHM analysis matched post-flight analysis by test engineers, including successfully pinpointing a laser altimeter failure, see Fig 2.3: the barometric altimeter, pitch, and the velocity readings indicated an *increase* in altitude ($\sigma_{S_{B\uparrow}}$ and $\varphi_{S_{S\uparrow}}$ held) while the laser altimeter indicated a *decrease* ($\sigma_{S_{L\downarrow}}$ held). The posterior marginal $\Pr(H_L = \text{healthy} \mid e^n \models \{\sigma_{S_L}, \sigma_{S_B}, \varphi_{S_S}\})$ of the node H_L, inferred from the BN, dropped from 70% to 8%, indicating a low degree of trust in the laser altimeter reading during the outage; engineers attribute the failure to the UAS exceeding its operational altitude.

2.6 Conclusion

We presented a novel SHM technique that enables both real-time assessment of the system status of an embedded system with respect to temporal-logic-based specifications and also supports statistical reasoning to estimate its health at runtime. To ensure Realizability, we observe specifications given in two real-time projections of LTL that naturally encode future-time requirements such as flight rules. Real-time health modeling, e.g., using Bayesian networks allows mitigative reactions inferred from complex relationships between observations. To ensure Responsiveness, we run both an over-approximative, but *synchronous* to the real-time clock (RTC), and an exact, but *asynchronous* to the RTC, observer in parallel for every specification. To ensure Unobtrusiveness to flight-certified systems, we designed our observer algorithms with a light-weight, FPGA-based implementation in mind and showed how to map them into efficient, but reconfigurable circuits. Following on our success using rt-R2U2 to analyze real flight data recorded by NASA's Swift UAS, we plan to analyze future missions of the Swift or small satellites with the goal of deploying rt-R2U2 onboard.

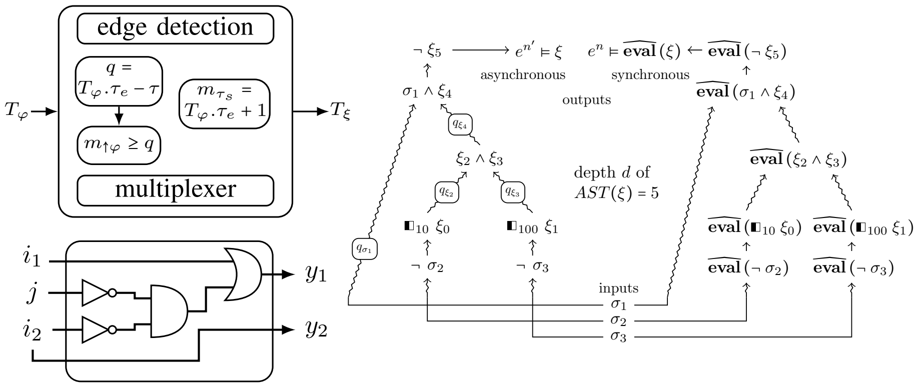
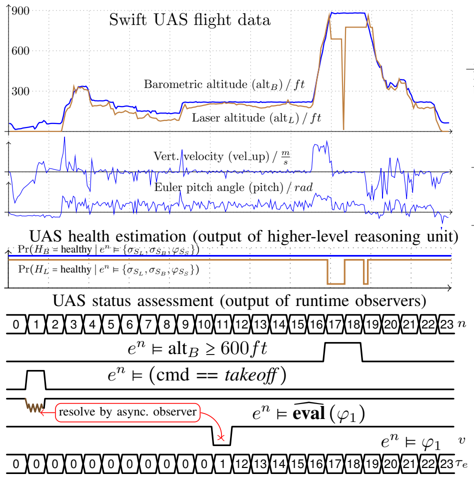
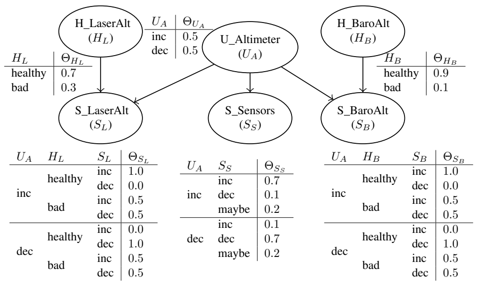

edge detection
$q = T_\varphi.\tau_e - \tau$
$m_{\tau_s} = T_\varphi.\tau_e + 1$
$m_{\uparrow\varphi} \geq q$
multiplexer
T_φ
T_ξ
i_1
j
i_2
y_1
y_2
$\neg\,\xi_5 \longrightarrow e^{n'} \vDash \xi$
asynchronous
$e^n \vDash \widehat{\textbf{eval}}(\xi) \leftarrow \widehat{\textbf{eval}}(\neg\,\xi_5)$
synchronous
outputs
$\sigma_1 \wedge \xi_4$
$\widehat{\textbf{eval}}(\sigma_1 \wedge \xi_4)$
q_{ξ_4}
$\xi_2 \wedge \xi_3$
$\widehat{\textbf{eval}}(\xi_2 \wedge \xi_3)$
depth d of $AST(\xi) = 5$
q_{ξ_2}
q_{ξ_3}
q_{σ_1}
$\blacksquare_{10}\ \xi_0$
$\blacksquare_{100}\ \xi_1$
$\widehat{\textbf{eval}}(\blacksquare_{10}\ \xi_0)$
$\widehat{\textbf{eval}}(\blacksquare_{100}\ \xi_1)$
$\neg\,\sigma_2$
$\neg\,\sigma_3$
$\widehat{\textbf{eval}}(\neg\,\sigma_2)$
$\widehat{\textbf{eval}}(\neg\,\sigma_3)$
inputs
σ_1
σ_2
σ_3

Figure 2.2: Left: hardware implementations for $\blacksquare_\tau\ \varphi$ (top) and $\widehat{\textbf{eval}}\,(\blacksquare_\tau\ \varphi)$ (bottom). Right: subformulas of AST(ξ), observers, and queues synthesized for ξ. Mapping the observers to hardware yields two levels of parallelism: (i) asynchronous (left) and the synchronous observers (right) run in parallel and (ii) observers for subformulas run in parallel, e.g., $\blacksquare_{10}\ \xi_0$ and $\blacksquare_{100}\ \xi_1$.





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

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

Inputs to our rt-R2U2 are flight data, sampled in real time; a health model as BN, right; and an MTL specification φ. Outputs: health estimation (posterior marginals of H_L and H_B, quantifying the health of the laser and barometric altimeter) and the status of the UAS.

Figure 2.3: Adding SHM to the Swift UAS

Chapter 3

Runtime Observer Pairs and Bayesian Network Reasoners On-board FPGAs: Flight-Certifiable System Health Management for Embedded Systems [21]

3.1 Introduction

[1]

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) [23] 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 harm during their missions, we propose a framework designed to deliver runtime System Health Management (SHM) [24] while adhering to strict operational constraints, all aboard a low-cost, dedicated, and separate

[1]The material in this chapter is published in [21].

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 re-compilation and we use an intuitive, expressive specification language; we require real-time projections of Linear Temporal Logic (LTL) since operational concepts for UAS 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 [43]. 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 [39]. In this chapter, 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.

3.1.1 Related Work

While there has been promising work in Bayesian reasoning for probabilistic diagnosis via efficient data structures in software [41,44], this does not meet our Unobtrusiveness 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., [17,45]. 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 [37], 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 [32,35] 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 [28] 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.

3.1.2 Contributions

We define hardware, FPGA encodings for both the temporal logic runtime observer pairs proposed in [39] 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 chapter is organized as follows: Section 3.2 provides the reader with theoretical principles of our approach. Section 3.3 provides an overview of the various parts and Sections 3.4 and 3.5 give more details about the hardware implementation. A real-world test case of NASA's Swift UAS is evaluated in Section 3.6. Section 3.7 concludes this chapter with a summary of our findings.

3.2 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 [43] for a detailed system-level overview.

3.2.1 Temporal-Logic Based Runtime Observer Pairs [39]

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 [2] and mission-time LTL [39], which reduces to MTL with all operator bounds being between now (i.e., time 0) and the mission termination time.

Discrete-Time MTL [39] 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 *$\varphi$ holds at time n of execution e*, denoted $e^n \models \varphi$, inductively as follows:

$$\begin{array}{llllll}
e^n \models true & & \text{is } \mathbf{true}, & e^n \models \sigma & \text{iff} & \sigma \text{ holds in } s_n, \\
e^n \models \neg\varphi & \text{iff} & e^n \not\models \varphi, & e^n \models \varphi \wedge \psi & \text{iff} & e^n \models \varphi \text{ and } e^n \models \psi, \\
e^n \models \mathcal{X}\varphi & \text{iff} & e^{n+1} \models \varphi, & e^n \models \varphi \vee \psi & \text{iff} & e^n \models \varphi \text{ or } e^n \models \psi, \\
e^n \models \varphi\, \mathcal{U}_J\, \psi & \text{iff} & & & & \\
& \multicolumn{5}{l}{\exists i (i \geq n) : (i - n \in J \wedge e^i \models \psi \wedge \forall j (n \leq j < i) :\ e^j \models \varphi).}
\end{array}$$

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* [39]. 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 $v \in \{\mathbf{true}, \mathbf{false}, \mathbf{maybe}\}$ 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 [39]. *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 $T.v \in \{\mathbf{true}, \mathbf{false}\}$ 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: $T.v \in \{\mathbf{true}, \mathbf{false}, \mathbf{maybe}\}$ 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.

3.2.2 Bayesian Networks for Health Models

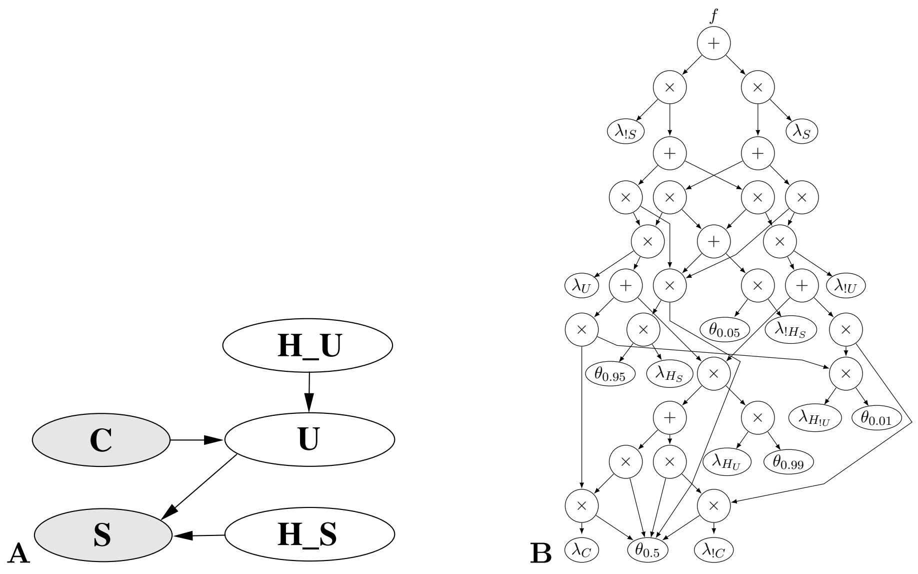


Figure 3.1: **A:** BN for Health management. **B:** Arithmetic circuit

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., [31, 34]) 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 3.1A. 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 [41]. 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 $Pr(H\_S = \mathit{good}|\mathbf{e})$ 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 [33]. 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. 3.1) 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 [11, 31].

The AC is factually a compact encoding of the joint distribution into a network polynomial [13]. The marginal probability (see Corollary 1 in [13]) for a variable x given evidence $\mathbf{e}$ can then be calculated as $Pr(x \,|\, \mathbf{e}) = \frac{1}{\mathrm{Pr}(\mathbf{e})} \cdot \frac{\partial f}{\partial \lambda_x}(\mathbf{e})$ 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.

3.2.3 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.

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 3.2), 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*.

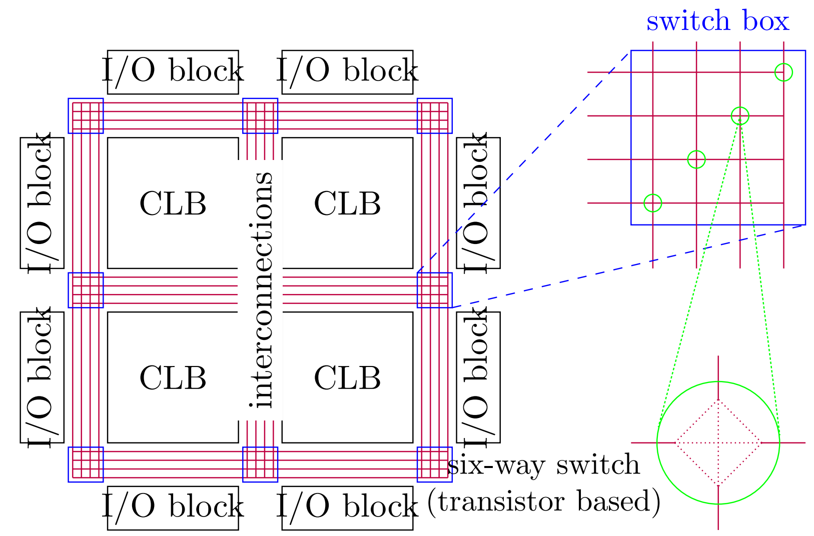


Figure 3.2: Simplified representation of a modern FPGA architecture.

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.).

3.3 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 3.3) 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[2]

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

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.

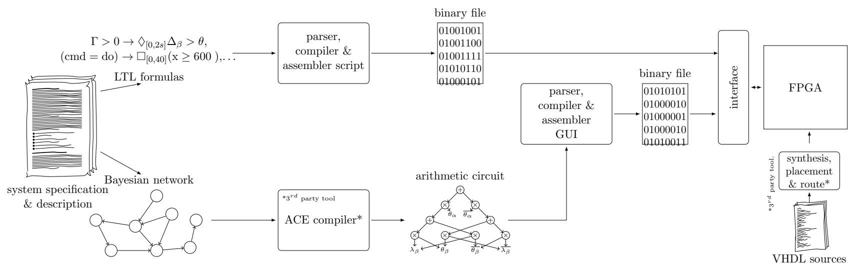

$\Gamma > 0 \rightarrow \Diamond_{[0,2s]}\Delta_\beta > \theta$,
(cmd = do) $\rightarrow \Box_{[0,40]}$(x $\geq$ 600),...
LTL formulas
system specification & description
Bayesian network
parser, compiler & assembler script
binary file
01001001 01001100 01001111 01010110 01000101
parser, compiler & assembler GUI
binary file
01010101 01000010 01000001 01000010 01010011
interface
FPGA
*3rd party tool
ACE compiler*
arithmetic circuit
synthesis, placement & route*
*3rd party tool.
VHDL sources

Figure 3.3: rt-R2U2 software tool chain

3.3.1 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[3] 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 3.5, is controlled with a Java GUI.

3.3.2 Hardware

The hardware architecture (Figure 3.4A) 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.

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

[3]`http://reasoning.cs.ucla.edu/ace/`

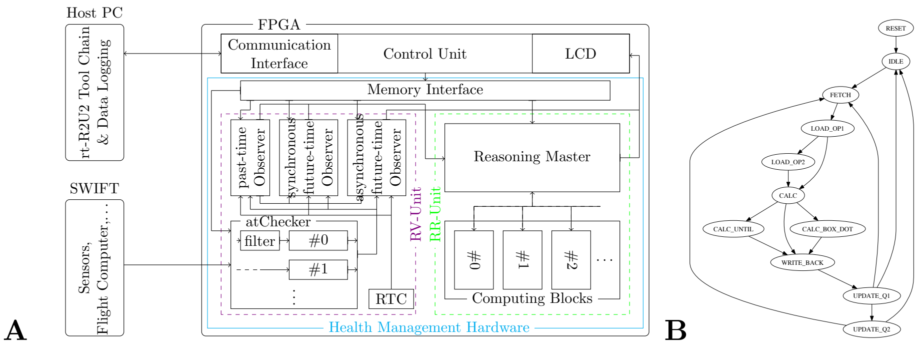


Figure 3.4: **A:** Overview of the rt-R2U2 architecture. **B:** FSM for the ftObserver.

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.

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.

3.4 FPGA implementation of MTL/mission-time LTL

As shown in Figure 3.4A, 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 3.4B[4]). 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 [39]). 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 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 ([39]).

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

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

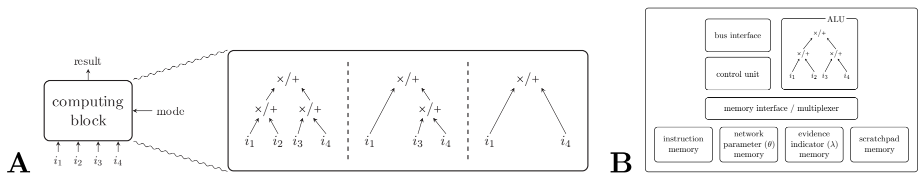


Figure 3.5: **A:** A computing block and its three modes of operation. **B:** Internals of a computing block.

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.

3.5 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 3.3). 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 3.5A shows the three different patterns. Each pattern consists of up to three arithmetic operators (addition or multiplication) and can have 2, 3, or 4 inputs. Such a small pattern can be efficiently executed by a BN computing block. Figure 3.5B 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 3.5A) 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.

3.6 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

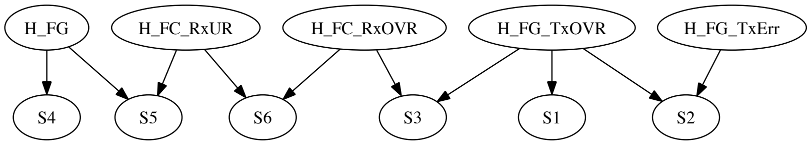
Table 3.1: Signals and sources used in this health model, sampled with a 1Hz sampling rate

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 3.1. 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.

3.6.1 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 3.6A, 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 3.6A. 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 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.



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

Figure 3.6: Bayesian Network for our example with legend of health nodes.

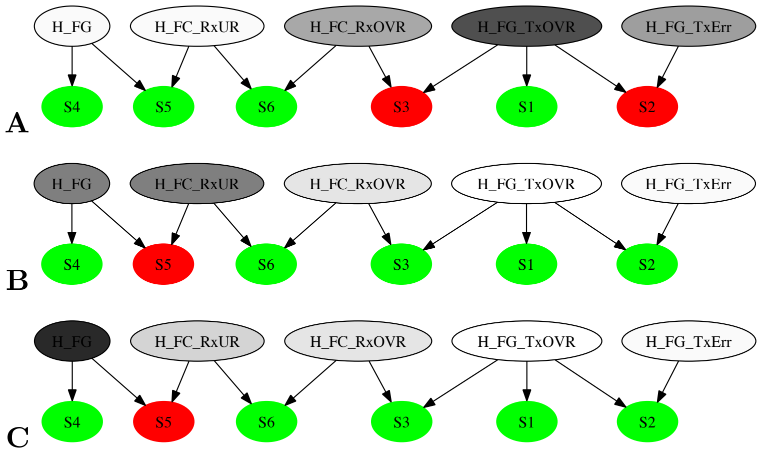
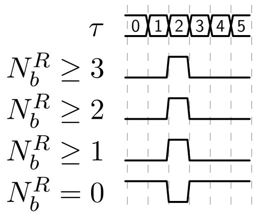


Figure 3.7: **A, B, C**: posterior probabilities (lighter shading corresponds to values closer to 1.0) for different input conditions.

Figure 3.7A 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 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 3.7B, C 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.



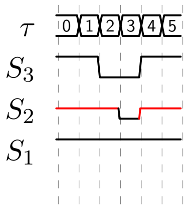


Figure 3.8: Recorded traces: sensor signals (left), trace of $S_1 \dots S_3$ (right).

		$\tau = 1$	$\tau = 2$	$\tau = 3$
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%

Table 3.2: Data of health nodes (right) reflecting the buffer overflow situation shown in 3.7A.

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 3.8 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 3.8. 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.

3.7 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.

Description	Formula
S_1: The FG packet transmission rate N_{tot}^R is appropriate: about 64 per second.	$63 \leq N_{tot}^R \leq 66$
S_2: The number of bad packets N_b^R is low, no more than one bad packet every 30 seconds.	$\Box_{[0,30]}(N_b^R = 0 \vee (N_b^R \geq 1\ \mathcal{U}_{[0,30]} N_b^R = 0))$
S_3: The bad packet rate N_b^R does not appear to be increasing; we do not see a pattern of three bad packets within a short period of time.	$\neg(\Diamond_{[0,30]} N_b^R \geq 2 \wedge \Diamond_{[0,100]} N_b^R \geq 3)$
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_x^N, FG_y^N). To avoid any false positive evaluations due to a noisy sensor, we filter the input signal.	$((Hd_x \geq 0 \rightarrow FG_x^N \geq 0) \wedge$ $(Hd_x < 0 \rightarrow FG_x^N < 0)) \vee$ $((Hd_y \geq 0 \rightarrow FG_y^N \geq 0) \wedge$ $(Hd_y < 0 \rightarrow FG_y^N < 0))$
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.	$Eul := \quad (\|p\| > \theta \vee \|q\| > \theta \vee \|r\| > \theta) \rightarrow$ $(\|FG_x\| > \theta_{FG} \vee \|FG_y\| > \theta_{FG} \vee$ $\|FG_z\| > \theta_{FG})$ $\neg(\neg Eul \wedge (\Diamond_{[2,100]}(\neg Eul \wedge \Diamond_{[2,100]} \neg Eul)))$
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.	$E^{log} \rightarrow ((E_g^{log} \wedge \neg E_b^{log}) \vee (E_b^{log} \wedge \neg E_g^{log})$
S_6': This case study uses a 1Hz sampling rate. We are losing precision and S_6 becomes $N_g^R + N_b^R = N_{tot}^R = 64$.	$N_{tot}^R = 64$

Table 3.3: Temporal formula specifications that are translated into paired runtime observers for the fluxgate magnetometer (FG) health model

Chapter 4

UAS Platforms and Integration

This chapter discusses details of implementation of the rt-R2U2 framework on the Parallella board and its integration into the DragonEye UAS in Section 4.1 Initial work for integration on-board the Swift UAS is also included; that effort had to be suspended with the suspension of this UAS platform. We present the instrumentation of the Arduino Flight software for monitoring relevant software and sensor data. Section 4.2 presents a detailed risk analysis, as a part of the Flight Readiness Review (FRR). This chapter concludes with details concerning the actual hardware integration in Section 4.3 and initial results of a series of bench flight tests in Section 4.4.

4.1 Architecture and Implementation

4.1.1 Initial Work: Swift UAS and Beyond

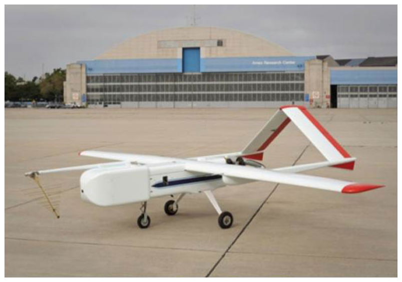
Since we tested our rt-R2U2 framework by playing back all of the available recorded data streams from flight tests of the NASA Swift UAS and analyzing those streams as if we were flying rt-R2U2 on-board, the project team initially considered the NASA Swift UAS as the platform for flight testing. However, after an internal NASA review and consultation with the NASA Ames AFSRB review board per NASA NPR 7900.3C, the lithium-based battery system assembled and delivered by the MLB Company, the original SWIFT hand glider manufacturer, was not deemed safe to operate at Moffett Field due to the lack of monitoring and protection in the system delivered by the company. Alternative systems with adequate safety and monitoring functionality could be developed by external companies but no off-the-shelf commercially available product was identified. The team received outside quotes from companies for constructing a custom propulsion battery system solution. Responding companies included A123 Systems and the Tenergy Corporation. Unfortunately, the required resources for acquisition, integration and testing of a new battery system were significantly beyond the resources of this project.

Three alternative UAS platforms were identified: the NASA SIERRA UAS (Figure 4.1b), the NASA Viking-400 UAS (Figure 4.1a), and the NASA Dragon Eye UAS (Figure 4.1c). These candidate vehicle systems are developed and operated by the Earth Science division (Code SG) at NASA Ames and would provide frequent future flight testing opportunities for the IVHM payload as tag-along payloads as opportunities present themselves. The specifications for the Viking and Dragon Eye vehicles are shown in Table 4.1 and Table 4.2 below, respectively. The SIERRA and Viking UAS vehicles are similar in size and specification. Both aircraft are in the largest UAS size category (CAT-III per NASA NPR 7900.3C, Appendix I), providing validation and experimentation at the largest CAT-III size. The SIERRA and NASA Viking 400 also share a common control system and avionics infrastructures based on the Cloud Cap Technologies Piccolo 2 autopilot system [4]. However, integration of hardware and experimental software into the larger vehicles would require a formal design, development and review process beyond the resources and constraints of the Phase 1 activity, but appropriate for future tasks (Phase 2 and beyond). Also, both vehicle platforms were under development, and their programs are still in the integration and testing phase as of the time of this report. The alternative NASA Dragon Eye aircraft are small low-cost aircraft with the most frequent flight opportunities. The Dragon Eyes provide the lowest cost of entry, provide the ability for the project team to quickly develop and test concepts, and allow the team full access to the interior components with the most flexibility as a CAT-I UAS aircraft. However, the closed nature of their autopilot and avionics system, and the export restriction designation of the hardware would make experimentation difficult.

Characteristic	Data
MGTOW (fuel + max payload)	540 pounds
Empty Weight	320 pounds
Wing Span	20.0 feet
Length	14.7 feet
Height	5 feet (Base of wheels to top of vertical stabilizer)
Power Plant	498is Twin Boxer Engine @ 38HP
Endurance	8-12 hours
Cruise Sped	60 knots
Dash Speed	90 knots
Launch/Recovery Method	Autonomous on wheeled gear

Table 4.1: Viking-400 UAS Specifications

(a) NASA Viking-400 UAS


(b) NASA SIERRA UAS

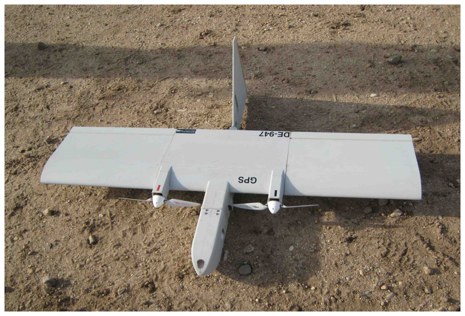

(c) DragonEye UAS

Figure 4.1: NASA UAS Candidates for Flight Testing

The project team decided on a two-vehicle approach for Phase 1 activities. The project would target the NASA Dragon Eye UAS as the primary Phase 1 platform for implementation and testing, but would targeted the NASA Viking-400 UAS as the principle flight test vehicle platform to support future Phase 2 and beyond flight experiments. For the larger aircraft, the team developed a hardware ground-based integration and testing facility for the Viking 400 aircraft. The team members modified the Dragon Eye UAS, completely stripping out the old stock avionics and replacing it with NASA and open-source autopilot hardware/software that provided the team full access to the flight system and onboard components. The team developed a duplicate iron-bird ground test system for the modified Dragon Eyes (Figure 4.2), allowing for simulation in the loop testing.

To support both Phase 1 and future activities, two ground-based hardware-in-the-loop simulation systems were developed in this project. For the Viking 400, the project team acquired a complete Piccolo autopilot system, providing hardware testing and integration that is would support testing, verification, and validation on both the NASA Viking and SIERRA platforms. The ground system includes a flight management system, ground station, developers kit, flight sensors, antennas, and ground/air radio modems. A ground-based training simulation system was acquired and configured in building N269 at NASA Ames Research Center; it is pictured in Figure 4.3. Team members took part

Characteristic	Data
Airframe Mnf/Mdl	AeroVironment Rq-14 DragonEye
Features	fully autonomous operation, in-flight reprogramming, small size, lightweight, bungee-launched, waypoint navigation, laptop mapping, image capture
Range	5 km
Speed	35 km/h
Op. Altitude (Typ.)	100-500 ft AGL
Span	3.75 ft (1.1 m)
Length	3 ft
Weight	5.9 lb (2.7 kg)
Launch Method	Bungee-launched
Recovery Method	Conventional horizontal landing

Table 4.2: DragonEye UAS Specifications

in a three-day on-site training and information session provided by L-3 Communications at NASA Ames.

4.1.2 Arduino-based DragonEye

Figure 4.4 illustrates the overall HW architecture of the DragonEye. It is based on the APM 2.6 HW [3] and was previously used for test flights without the rt-R2U2 monitoring. The only change to the previous flights is the additional payload (the Parallella board) with a read-only UART Interface. The Parallella board is highlighted in red in the schematic.

The main components are:

- **APM Board**: from 3DRobotics is the Arduino based Hardware, where the Autopilot software APMPlane is running
- **Power**: LiPo Battery to power all the components, Current/Voltage Monitor, Power Regulator to generate 5.3 Volts required by the APM Board, Sub-Components and the Parallella Board
- **Multiplexer Pololu721**: Failsafe mechanism switches control of the Motors/Servos between Autopilot-Software and the direct radio signal from the operator
- **Communication**: RC receiver to directly control the Motors/Servos from Ground, Telemetry Modem for communication between the Software and the Ground Control Station
- **Sensors**: Airspeed Probe, GPS, Compass, Barometer, Inertial Navigation

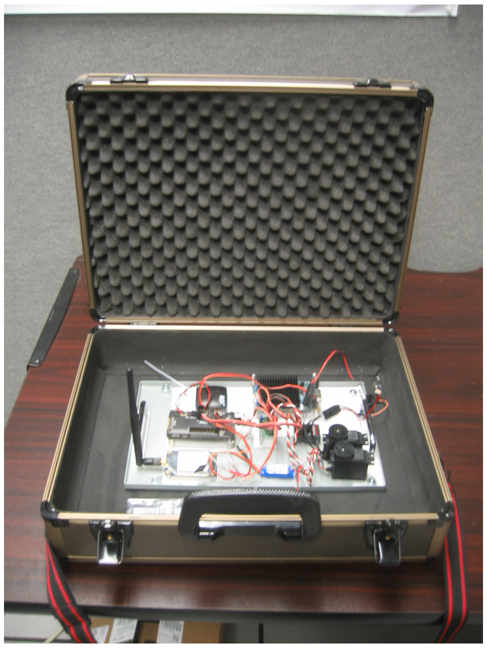


Figure 4.2: Ironbird ground test system for modified Dragon Eyes

- **Actuators**: Two engines with motor controllers, two servos (left and right elevon)
- **Parallella Board**: Performs Runtime Monitoring and Bayesian Reasoning of the of the Autopilot SW states

The redesigned Dragon Eye flight system is shown in Figure 4.5 and has now become the standard avionics suite for all NASA Dragon Eye vehicles operated at Ames. The system includes an open-source Ardupilot Mega 2.6 (APM) autopilot, a mesh-enabled Digi 9Xtend radio modem, and an Adapteva Parallella-16 secondary processor. The APM software was modified for use in the Dragon Eye configuration, and further modified for this project to include monitoring and communication through UART2.

The modifications to the Dragon Eye vehicles were made in collaboration with NASA Code SG team members and subject to a number of ground tests, included range, EMI/EMC, electrical, thermal, and endurance testing. Figure 5 below shows six of the Dragon Eye aircraft undergoing these modifications. The full vehicle system was implemented, tested, reviewed, and approved for flight testing, receiving all requirements needed for flight testing per NASA NPR 7900.3C, including obtaining a certificate of airworthiness from the Airfield Flight Safety Review Board (AFSRB) and passing a Flight Readiness Review (FRR). The project teams software was developed and tested in the Dragon Eye iron-bird, then installed and tested on one of the Dragon Eye vehicle systems (tail number DE 1029) and is ready for flight test data collection.

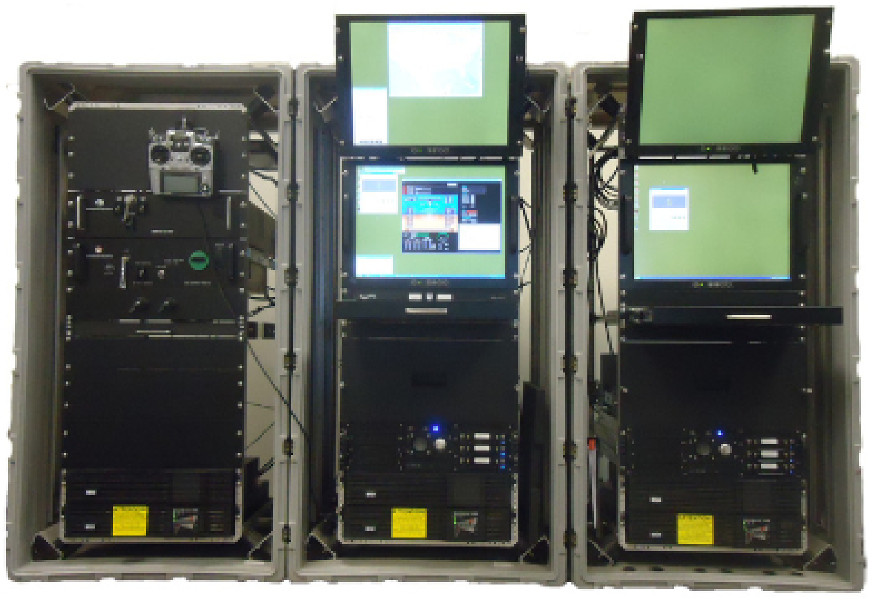


Figure 4.3: Ground simulation and testing system for the Viking 400 UAS

4.1.2.1 DragonEye RC-Mode

During normal operation, the autopilot software is in control of the DragonEye motors/servos. Driven by periodic interrupt, a watchdog routine regularly checks if the autopilot software is still running. If the check fails, the software is entering a failsafe mode, where the controls from the radio are directly forwarded to the motor controllers (see low-level SW Failsafe in figure 4.6) every 20ms. The rest of the calculation time is given to the autopilot software in order to recover itself. If the check reports that the software is operational again, the Low-level SW Failsafe is deactivated and the control is handed over to the Autopilot Software again.

As illustrated in figure 4.6, the DragonEye is also equipped with a HW-failsafe switch. In case of unexpected behavior, the hardware multiplexer (see Mux Pololu 721 in schematic) can be directly controlled by the ground operator on ground via the radio modem. In this case, the airplane is in control of the operator and the Ardupilot Software and Hardware is bypassed completely.

4.1.3 Hardware Components for rt-R2U2

4.1.3.1 Parallella Board

The Parallella board [1] is a credit-card sized computing platform created by Adapteva with a Xilinx Zynq 7xxx SoC as Central Processing Unit. A

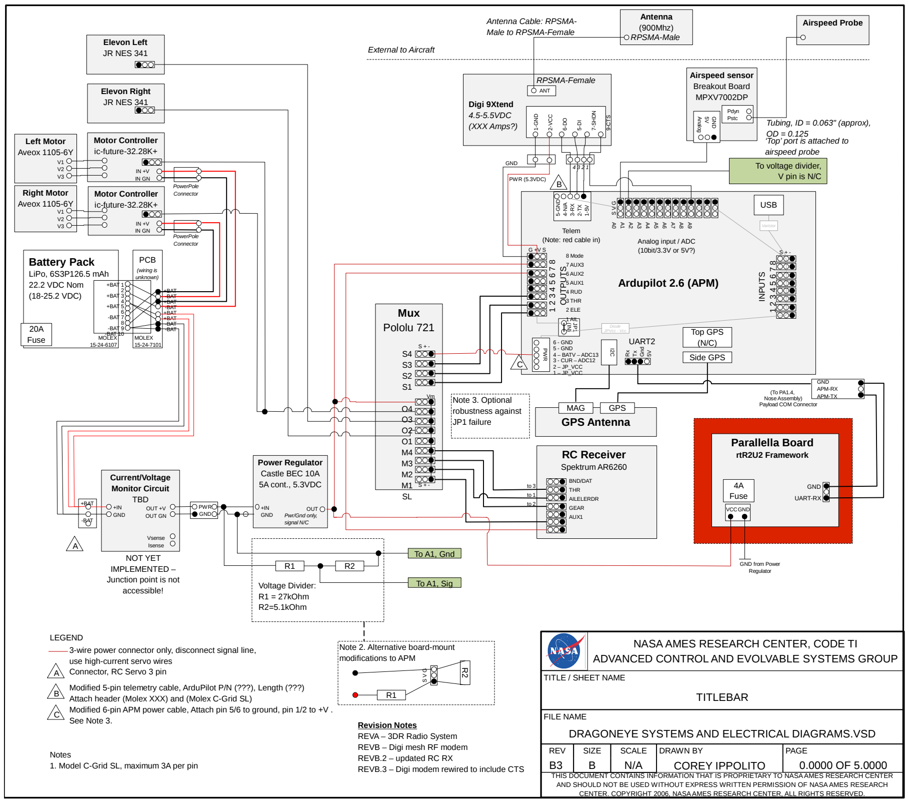

Elevon Left
JR NES 341
Elevon Right
JR NES 341
Left Motor
Aveox 1105-6Y
Right Motor
Aveox 1105-6Y
Motor Controller
ic-future-32.28K+
Battery Pack
LiPo, 6S3P126.5 mAh
22.2 VDC Nom
(18-25.2 VDC)
20A Fuse
PCB
Mux
Pololu 721
Antenna
(900Mhz)
RPSMA-Male
Antenna Cable: RPSMA-Male to RPSMA-Female
External to Aircraft
Airspeed Probe
Digi 9Xtend
4.5-5.5VDC
(XXX Amps?)
Airspeed sensor
Breakout Board
MPXV7002DP
Tubing, ID = 0.063" (approx), OD = 0.125
'Top' port is attached to airspeed probe
To voltage divider, V pin is N/C
Telem
Analog input / ADC (10bit/3.3V or 5V?)
Ardupilot 2.6 (APM)
USB
OUTPUTS
INPUTS
UART2
Top GPS (N/C)
Side GPS
MAG
GPS
GPS Antenna
Note 3. Optional robustness against JP1 failure
RC Receiver
Spektrum AR6260
Parallella Board
rtR2U2 Framework
4A Fuse
GND
UART-RX
GND from Power Regulator
Power Regulator
Castle BEC 10A
5A cont., 5.3VDC
Current/Voltage Monitor Circuit TBD
NOT YET IMPLEMENTED – Junction point is not accessible!
Voltage Divider:
R1 = 27kOhm
R2=5.1kOhm
To A1, Gnd
To A1, Sig
Note 2. Alternative board-mount modifications to APM
LEGEND
3-wire power connector only, disconnect signal line, use high-current servo wires
Connector, RC Servo 3 pin
Modified 5-pin telemetry cable, ArduPilot P/N (???), Length (???)
Attach header (Molex XXX) and (Molex C-Grid SL)
Modified 6-pin APM power cable, Attach pin 5/6 to ground, pin 1/2 to +V. See Note 3.
Notes
1. Model C-Grid SL, maximum 3A per pin
Revision Notes
REVA – 3DR Radio System
REVB – Digi mesh RF modem
REVB.2 – updated RC RX
REVB.3 – Digi modem rewired to include CTS
NASA AMES RESEARCH CENTER, CODE TI
ADVANCED CONTROL AND EVOLVABLE SYSTEMS GROUP
TITLE / SHEET NAME
TITLEBAR
FILE NAME
DRAGONEYE SYSTEMS AND ELECTRICAL DIAGRAMS.VSD
REV B3
SIZE B
SCALE N/A
DRAWN BY COREY IPPOLITO
PAGE 0.0000 OF 5.0000
THIS DOCUMENT CONTAINS INFORMATION THAT IS PROPRIETARY TO NASA AMES RESEARCH CENTER AND SHOULD NOT BE USED WITHOUT EXPRESS WRITTEN PERMISSION OF NASA AMES RESEARCH CENTER. COPYRIGHT 2006, NASA AMES RESEARCH CENTER, ALL RIGHTS RESERVED.

Figure 4.4: DragonEye Schematic with Parallella Board Payload

Ubuntu Linux Distribution is running on the ARM-Cores which is used to communicate and pre-process the data received from the Autopilot Software. The Temporal Logic monitors are running in the SoC's FPGA section. There is also a High performance 16 or 64 Core Epiphany Co-Processor on the board that can be used for high-performance calculations (e.g. Software Evaluation of Bayesian Networks). The Parallella board requires a heat sink.

4.1.3.2 Power Connection for Parallella Power

In order to power the Parallella Board, additional cables and a connector to power the board is installed. The Parallella board uses +5VDC at up to 2A.

4.1.3.3 UART Connection for Parallella Communication

Two additional wires (GND and APM-TX) are installed to send data from the APM Board to the Parallella board (Figure 4.8. The UART2 port of the Arduino Flight computer is used to transmit the monitored data.

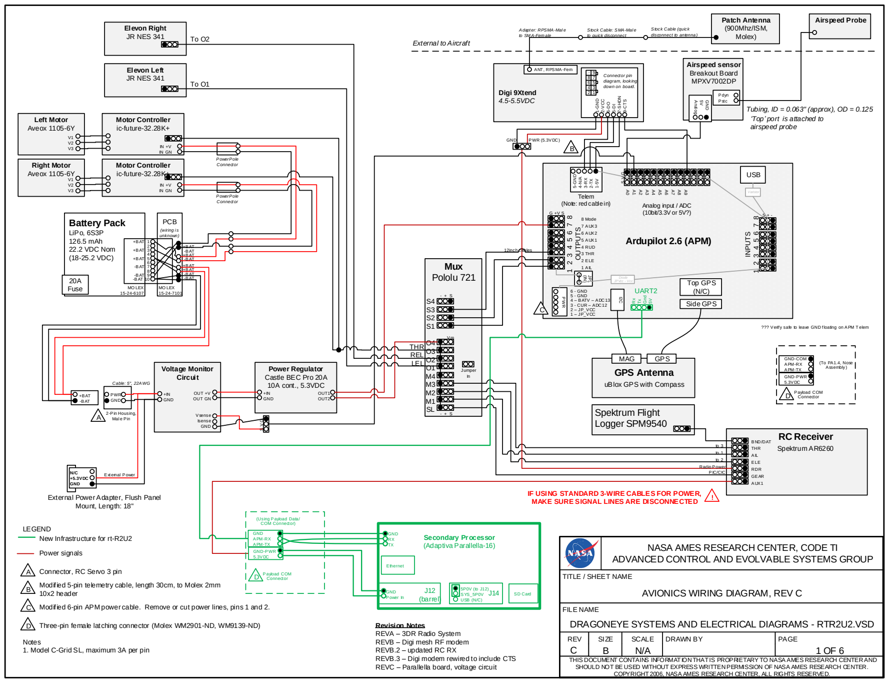


Figure 4.5: DragonEye Schematic with Parallella Board Payload: Green elements display changed to standard DragonEye configuration made to incorporate rt-R2U2.

4.1.3.4 Modified Hardware Components

The Parallella board is mounted in one of the wings of the DragonEye, which has enough space to hold the board. The power distribution is changed in order to power the Parallella board. As shown in the schematic in figure 4.4 additional wires are directly connected to the output of the Power Regulator. Furthermore, 2 additional wires are connecting the APM-UART Interface with the Parallella UART Interface. The RX-Line of the APM stays unconnected in order to avoid any communication from the Parallella to the APM Board.

4.1.4 Instrumentation of Flight Software

The Flight Software was branched from the DragonEye SW, that was already tested during a couple of Flights. Figure 4.9 shows the altered SW-components. The main SW modification is the addition of an additional Operating System Task for gathering the data to monitor and send it to the Parallella Board. The task runs at the lowest priority of all tasks and has the same frequency as the default logging task, which is 10Hz, whereas the higher priority tasks like reception of radio signal, high-level failsafe checks, inertial navigation (ahrs) update, speed/altitude calculations, stabilization or servo commands are called at a frequency of 50 Hz.

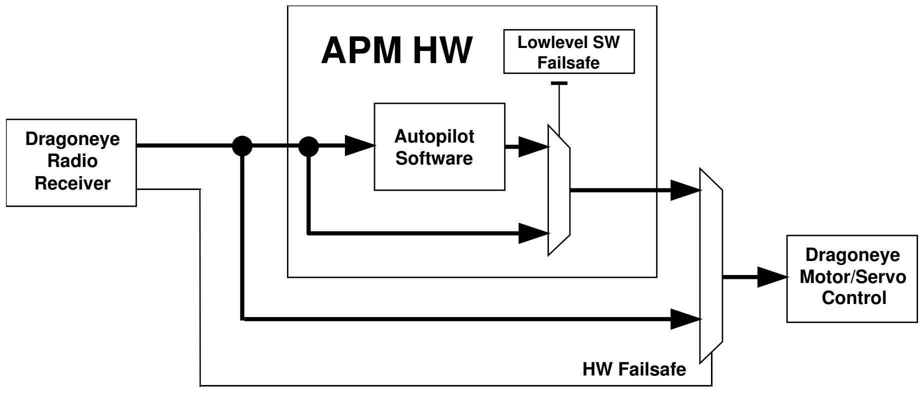


Figure 4.6: RC and SW Failsafe

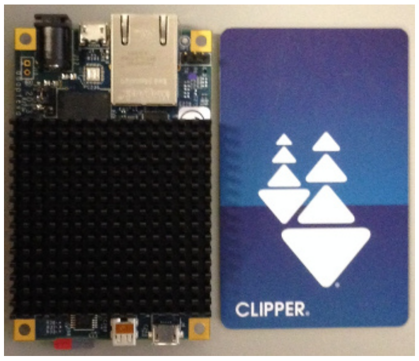


Figure 4.7: Parallella Board
Size: 3.4" x 2.15"
Weight: 64.9 g

In the original MAVlink (Protocol to communicate with Ground Control Station (GCS)) Driver, all telemetry was not only streamed on the UART A, where the telemetry modem is connected, but also the UART C. Hence, we modified the MAVlink driver to not send or receive data on UART C, since UART C is used for the monitoring of the Software. However, the UART driver itself was not modified and the default Methods are used for Communication.

Besides the above-mentioned modifications, some parts of the code are instrumented in order to gather data. In general, no calculations (depending on configuration exceptions are possible) or other instructions that could cause uncertain delays in the software were inserted with the instrumentation since they are limited to read only (const) direct memory copy instructions (passed by reference) to a (semaphore protected) buffer, which is then read by the low-priority monitoring task and transmitted on the UART Interface.

Wherever possible, the instrumentation is limited to a single copy instruction during the initialization/instantiations of the static classes,

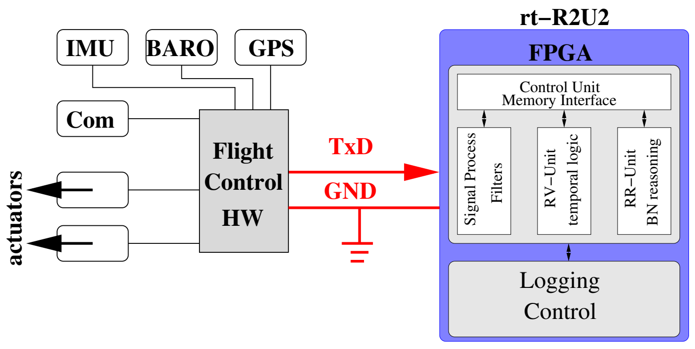


Figure 4.8: High-level architecture with read-only connection between FSW and rt-R2U2

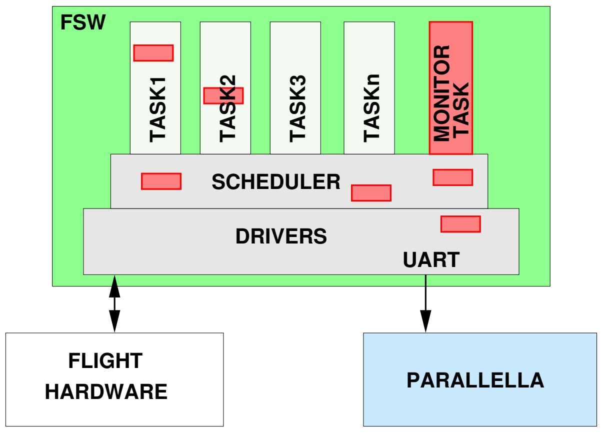


Figure 4.9: Software Architecture

where we only copied the address of the variables that are of interest. With this approach, we are limiting the timing influence of the SW by the instrumentation to the point in time when the software is initialized. The data that is stored at the gathered addresses are read only during the monitoring task and never written. Where data is only available on the stack, we inserted a single the read/copy by reference instruction in the original code.

Since the scheduler data is only available on the stack, we also added some copy instructions in order to collect task statistics. This task statistics were used during the testing phase to check if any of the tasks overrun their deadlines due to the instrumentation. They are also used to detect any other bad timing behaviors during runtime with our monitor.

In order to avoid a delay during the transmission of the data via the UART interface, only non-blocking transmission is used. Furthermore, the available space in the transmission ring buffer is checked and the data to transmit is immediately discarded if the buffer does not have enough space at this time.

4.1.4.1 Instrumentation Requirements

For rt-R2U2 monitoring, information about the sensor and software status is collected on a regular basis on the Arduino flight computer and transmitted via read-only UART to the rt-R2U2 Parallella processing board.

The following requirements must hold

- the instrumentation shall not influence the system behavior in a negative way, cause delay of tasks, lock-ups or even system resets
- variables from numerous subsystems, components and classes should be assembled
- a simple but robust serial communication protocol shall be used to transmit the assembled monitoring data over UART

The instrumentation architecture consists of three major parts:

1. means for accessing important variables
2. a task to update the buffer, pack the buffer, and initiate transmission of the buffer, and
3. CRC calculation, communication protocol

4.1.4.2 SW-safety design decisions

- all memory buffers are allocated statically; no dynamic memory allocated,
- modification to existing code at as few places as possible,
- write-access to the monitoring buffer is guarded by a *non-blocking* semaphore, i.e., in case the semaphore is taken, the task does not block; rather the current attempted access is aborted and that data-frame skipped. Thus, no other parts of the system is affected; in the worst-case monitoring data might be lost (but a skipped-counter is noting that situation)
- the monitoring buffer is kept to a minimal size, which is smaller than the maximum transfer buffer size of 256Bytes.
- UART transfer times are considerably lower than the monitoring task cycle time
- an efficient CRC generation enables error detection during transmission of the buffer data

4.1.4.3 Accessing ArduPlane Variables and Buffer Storage

Important system variables can contain information about the software (e.g., number of times a task did not finish in time or free memory), sensor status and values (e.g., barometric altitude), as well as behavioral data, e.g., mode of the aircraft, distance to next way-point, etc.

Different variables are places in different parts of the ArduPlane memory:

- global variable, member of global data structure, or (global) function
- variable is part of a (static) object and can be accessed directly or via accessor method
- variable is a local variable and only resides on the stack
- variable is local to ISR

Our approach enables simple and reliable access for almost all types of variables. A number of data structures are defined and there is one uniform routine `rtr2u2_update_signal` to obtain the value and update the buffer. Furthermore, during system initialization, the function `rtr2u2_register_static_address` is used to set up access capabilities for object members.

Variables of interest are listed in the enumeration `rtr2u2_signal_type`. These entries are used to access the corresponding variable during update, packing of the buffer, as well as unpacking on the Parallella-board side.

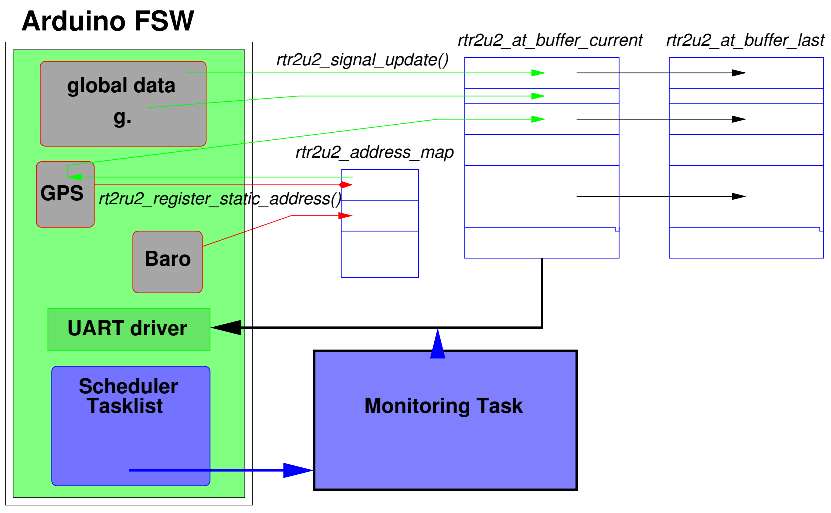


Figure 4.10: SW architecture details

4.2 Risk Analysis

For preparation of flight tests, a detailed risk analysis was performed as part of the FRR process. The risk analysis includes hardware (Parallella board) and its integration, software instrumentation of the flight software, as well as risks during flight tests due to injected failures.

4.2.1 Risks and Mitigation—Introduction

The Risks and Mitigation are categorized by their likelihood and consequence as defined by the U.S. Department of Transportation [47].

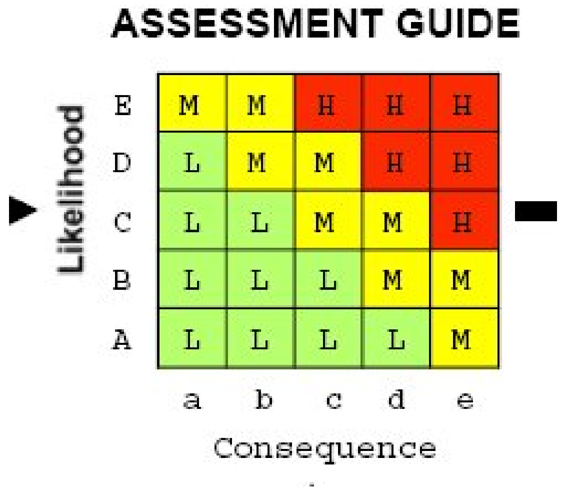

ASSESSMENT GUIDE
Likelihood
E
D
C
B
A
M M H H H
L M M H H
L L M M H
L L L M M
L L L L M
a b c d e
Consequence

Figure 4.11: Risk assessment from DOT [47] showing risk assessment levels (high–red, moderate–yellow, low–green) over likelihood vs. Consequence

Level	Likelihood
A	remote
B	unlikely
C	likely
D	highly likely
E	near certainty

Table 4.3: Levels for likelihood

4.2.2 Intended Flight Profiles

In order to test our monitoring and diagnosis capabilities, failures will be injected into the system. Permanent failures are injected before the flight starts and they will remain active during the entire duration of the flight. Dynamically injected failures will be injected by the flight-computer at a given time or by an operator command. These failures only concern *software* failures.

Level	Schedule	Impact
a	minimal or no impact	no impact
b	additional resources required; able to meet	slight impact, loss of evaluation data
c	minor slip in key milestones; not able to meet date	medium impact, loss of data, necessary work on FSW or operational procedures
d	major slip in key milestone; critical path impacted	minor/repairable damage to airframe of hardware components
e	Can't achieve key major program milestone	crash of DragonEye; possible damage to ground and safety risk to personnel

Table 4.4: Levels for schedule

4.2.2.1 Permanent Failures

Hardware failures obviously produce more realistic scenarios. However, the transition nominal-to-failure cannot be observed and it must be made certain that the UAS can operate safely (takeoff, operator-controlled) even with this failure.

Following scenarios will be included.

- covered GPS antenna: a low signal strength causes failing locks with satellites and or a lower number of satellites to be received. In general, that should produce navigation data of poor quality and (potentially) some errors.

- disconnected GPS antenna: no satellite lock; should produce errors in flight software

- disconnected GPS system: this produce errors in the flight software.

- covered/misaligned Pitot tube: causes erroneous measurements of airspeed. Depending on the flight SW, some of the errors can be compensated by GPS navigation. A fully covered Pitot tube (famous example) might prevent the UAS to take off altogether

- wrong subsystem configuration (e.g., baud rate): perhaps we can mimic the SWIFT magnetometer failure

- one bad engine: can/should this accomplished via hardware or via software?

4.2.2.2 Dynamically Injected Failures

We only inject failures into FSW.

- Add delays in tasks (to trigger failsafe mode)
- Overflow in UART buffer
- UART flush (causes UART driver to stop operation)
- Cause a scheduler.panic (block SPI semaphore)
- Trigger battery.exhausted
- Delay timer task so that it is still running when it is called again
- add bias/noise to actuator and motor commands.

4.2.3 Structure of Risks

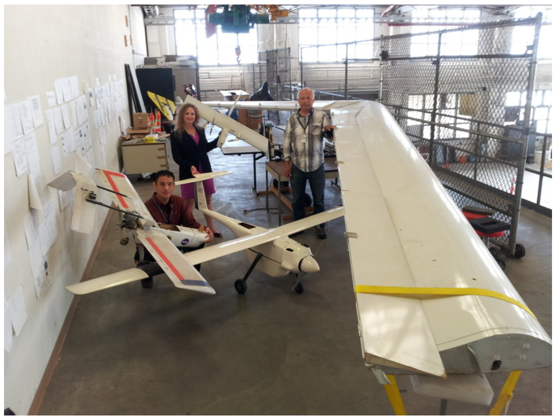
In the following sections, we list risks, severity/likelihood, and measures for mitigation and risk avoidance.

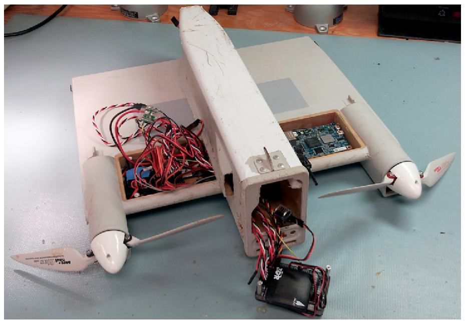
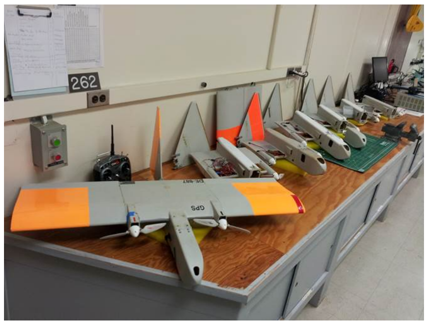
Figure 4.12 gives an overview and points to the individual sections in the Appendix, where the individual risk is discussed.

likely-	consequence				
hood	a	b	c	d	e
E	--	--	--	--	--
D	--	--	--	--	--
C	D.3.1 D.5.2 D.5.2	--	--	--	--
B	D.4.1 D.5.1 D.5.2 D.5.2 D.5.3 D.5.4 D.5.5 D.5.6 D.5.7 D.5.8	D.2.4 D.4.2	--	--	--
A	D.1.3	D.2.1 D.2.2	--	D.1.1 D.1.2	--

Figure 4.12: Overview of flight risks

4.3 Hardware Integration


(a) rt-R2U2 Target UAS Platforms

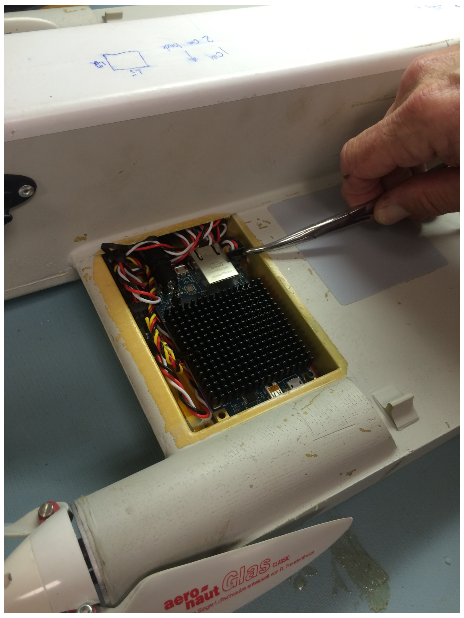
(b) Core DragonEye component (fuselage, inner wing segments, and propellers) containing all electronics; access panels on the underside of both wings near the fuselage and a removable nose cone enable installation of hardware including our Parallella board

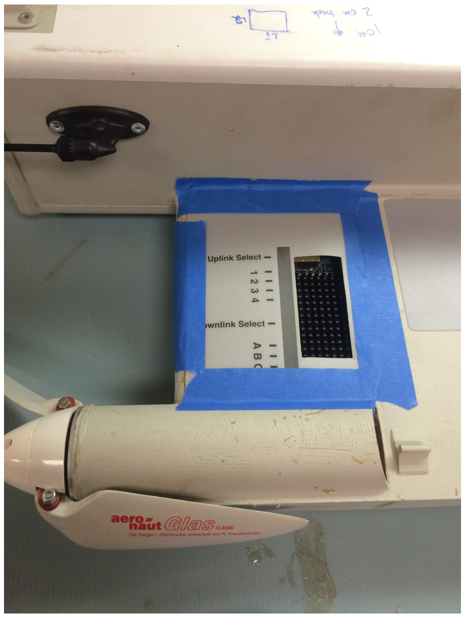
(c) Assembled DragonEye test vehicle next to array of spare parts in N-211 test laboratory

Figure 4.13: Indoor flight test photos from NASA Ames Building N-211-E: six airframes were modified and tested for this project in collaboration with NASA Ames Code SG

We investigated several available UAS platforms as candidates for our first hardware demonstrations and test flights; these are pictured in Figure 4.13a. Initially, we targeted the NASA Swift UAS as a demonstration platform; it is the 13-foot wingspan flying wing pictured on the far right. However, following a NASA decision to suspend operations using the Swift, we were required to focus on a new platform in order to continue our research. We next targeted the NASA DragonEye UAS; it is the 3-foot wingspan hand-launched UAS held in the middle of Figure 4.13a. This UAS had several advantages for hardware demonstrations and test flights, including that NASA had a large number of available DragonEye

UAS and spare parts, some of which are pictured in Figure 4.13c.


(a) Parallella board with heat sink in DragonEye wing.



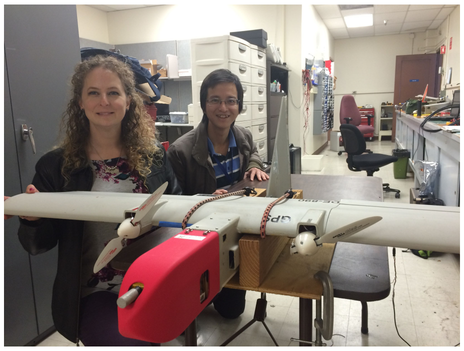
(b) Parallella board compartment in DragonEye wing covered for flight.

Figure 4.14: Photos of Parallella board integration in a DragonEye UAS from the series of indoor test-stand flight tests.

While space inside the DragonEye was constrained, we were able to fit the Parallella board in a compartment in the wing, near the fuselage, as pictured in Figure 4.13b. The Parallella board would not fit inside the fuselage itself due to the structure of the DragonEye, including the custom-made battery which occupies most of the available space in the fuselage, and the shape of the remaining space. We surmised that, had we been able to mount and connect the Parallella board within the fuselage, we would very likely have been able to cool it better, which was a major concern with its placement. The increased airflow through the fuselage due to a small vent in the DragonEye frame and the possibility of adding a fan could have helped with cooling. The compartment in the wing where we ultimately mounted the Parallella board was just large enough to accommodate it; wiring had to be connected in a tight space. For flight testing, both wing compartments are covered with a removable skin, creating a smooth wing surface, as pictured in Figure 4.14.

4.4 Initial Flight Tests

The project team laid out a series of incremental flight tests that were to be conducted from October to December of 2014 in the project plan. The first of these flight tests occurred ahead of schedule in September of


(a) PI Kristin Yvonne Rozier and Ph.D. Research Intern Quoc-Sang Phan assembling a DragonEye on the test stand in December, 2015.

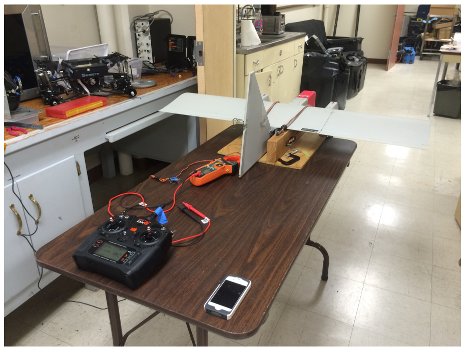

(b) DragonEye in a test stand flight test, with motors running, without fan. For tests with a fan, the large fan was placed on a table in front of the test stand.

Figure 4.15: Photos from series of indoor test-stand flight tests.

2014, with successful flight testing of the hardware configuration shown in Figure 4.5 at Moffett Federal Airfield (NUQ), representing the Rev C version of the hardware. Unfortunately, immediately after the September 2014 flight tests, UAS operations were unexpectedly suspended at Moffett Federal Airfield due in part to a close-call incident and ensuing mishap investigation (both related to aircraft operations from a separate and unrelated project). As of the time of this report, flight operations have not yet resumed. A dedicated vehicle, DragonEye tail number DE 1029, has been set aside in this projects ready-to-fly configuration and the project team is awaiting notification of the next flight test opportunity and resumption of flight operations at NUQ, which may occur later in 2015.

4.4.1 Results of Initial Stand Flight Tests

During the performance period, the team did receive temporary clearance for a series of indoor, stationary tests with all hardware integrated into the DragonEye. For an initial test in the lab, the DragonEye was mounted to a stand and power turned on. Power was tested and data from the flight software were transmitted to the Parallella board. The computing compartment was covered and the motors were kept off. In subsequent tests, motors were turned on and a large fan was pointed at the DragonEye to test properties like cooling in the presence of airflow over the wings and operation of the components in the presence of vibrations from the motors.

4.4.1.1 Temperature

Figure 4.16 shows the temperature of the FPGA chip on the Parallella board from the initial flight test with the motors kept off. Since there was no cooling via an air-stream, the unit heats up relatively quickly and shuts down itself after approximately 15 minutes into the test. Shutdown is initiated when the temperature reaches $87°C$. A warning is issued whenever the temperature reaches $75°C$.

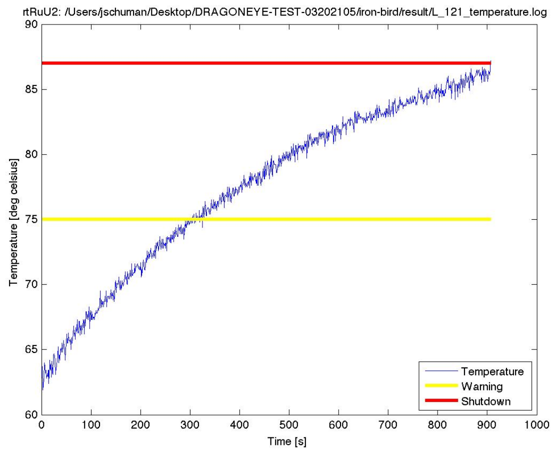


Figure 4.16: Development of chip temperature (y-axis) over time (x-axis, in seconds). Computer compartment closed, engines off, no airflow along the aircraft.

4.4.1.2 Raw Data

The raw data as transmitted to the Parallella board were recorded and analyzed. This test demonstrates that the instrumentation of the flight software is working correctly. Figure 4.17 shows temporal traces of the measured airspeed (A), altitude (B), and task-related data (C) over a scaled time (roughly 0.5 seconds per data point). Since the aircraft is fixed to the stand, the airspeed is minimal and remains constant over time. The altitude of the UAS is visible given a number of variables. `bar-alt` is the barometric altitude (scaled), `current-loc-alt` gives the current altitude of the AC as integrated by the state estimation filter. Finally, the current altitude can be estimated by integrating the `climb-rate`, i.e., $alt = 20 \times \int climb\_rate\ dt$. The constant 20 is a result of the 50Hz update rate of this signal. All three signals are close together indicating that the instrumentation and recording are working properly.

Finally, Figure 4.17 shows temporal traces of variables that are governed by the internal scheduler of the Arduino Flight software. For selected tasksthe number of slippages and the accumulative maximum delay are shown. Note that each task in the Arduino flight software is a piece of code that performs a specific operation, e.g., reading a sensor value, updating the AC position, generating an actuator signal. These tasks are executed according to their frequency and allotted time requirements. Tasks that cannot be executed at their given time might not be

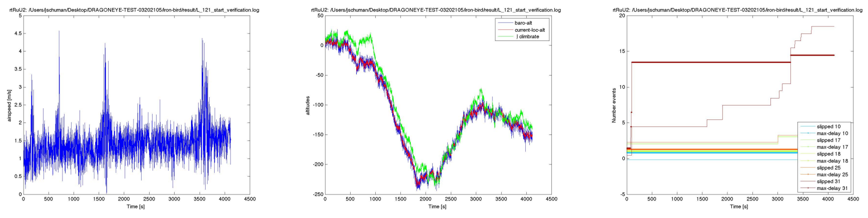


Figure 4.17: Temporal traces of airspeed (left), altitude (middle) and data related to software tasks in the APM (right)

executed during this round or might get delayed. Whereas most of the tasks have no slippage, task 25 and, in particular, task 31 is not executed regularly and assembles delay times.

4.4.1.3 Reasoning Output: Temporal Logic

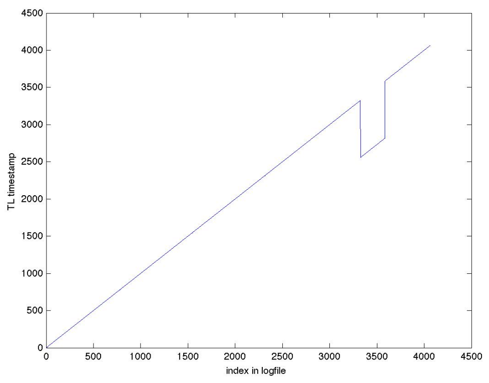


Figure 4.18: Values of time stamps as produced by the temporal reasoner. A reset of time stamp values seem to have occurred around record number 3,300 through 3,700.

Figure 4.19 shows temporal traces of several temporal monitors. In all cases, the values of the synchronous and asynchronous observers are shown as well as T_{async}, i.e., the time stamp for which the value of the given asynchronous observer is valid. A simple diagonal corresponds to a formula, which does not have to deal with longer and varying time intervals. Nevertheless, a consistent change in T_{async} as observed between 3000 and 4000 indicates some reset or glitch in the processing. This is also evident in the sequence of time stamps as produced by the reasoner as shown in Figure 4.18

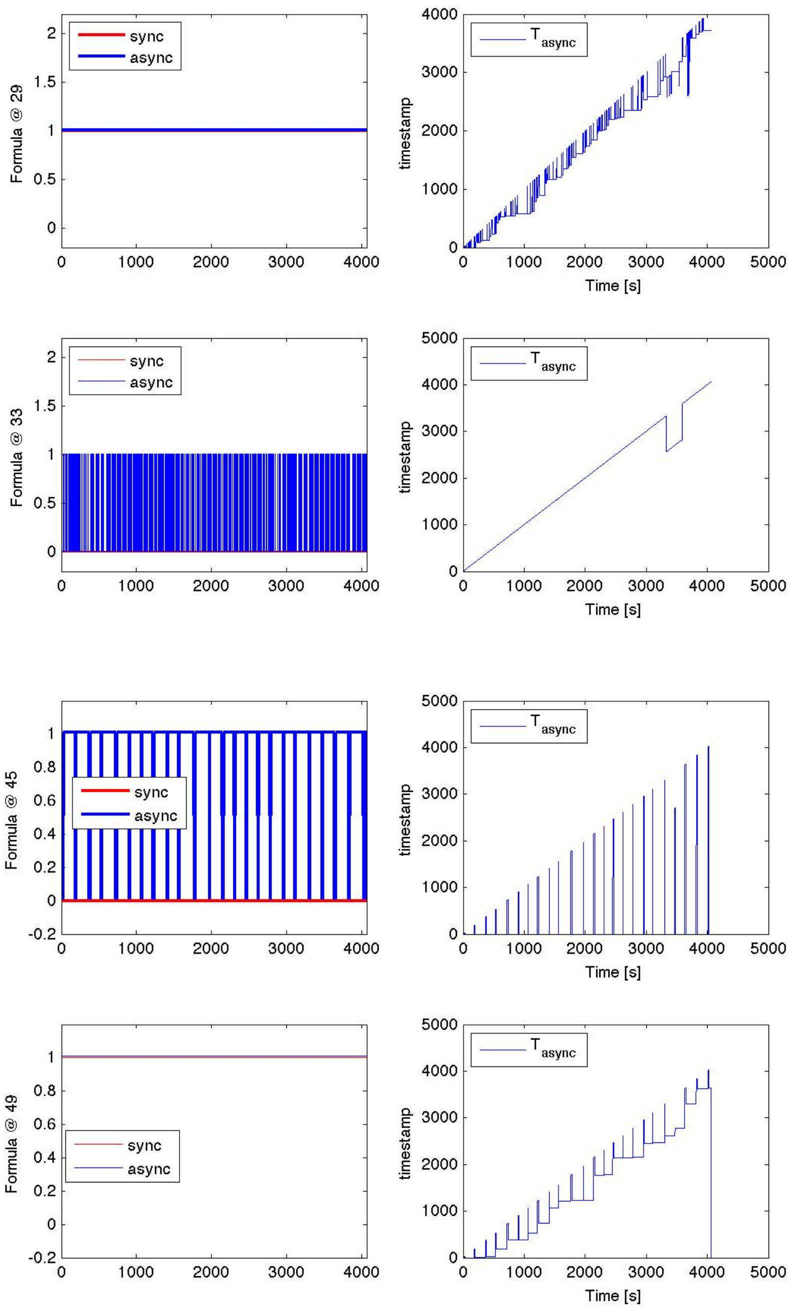


Figure 4.19: Output of four temporal monitors

Chapter 5

Conclusion

In this NASA technical memorandum, we designed a novel System Health Management framework, named rt-R2U2 after the FAA requirements that it addresses. Our rt-R2U2 performs real-time assessment of the system status of an embedded system, in this case a UAS, with respect to temporal-logic-based specifications. This enables better statistical reasoning to estimate its health at runtime than previous methods. To ensure REALIZABILITY, we observe specifications given in two real-time projections of LTL that naturally encode future-time requirements such as flight rules. Real-time health modeling via Bayesian networks allows mitigative reactions inferred from complex relationships between observations. To ensure RESPONSIVENESS, we run both an over-approximative, but *synchronous* to the system's real-time clock (RTC), and an exact, but *asynchronous* to the RTC, observer in parallel for every specification. To ensure UNOBTRUSIVENESS to flight-certified systems, we designed our observer algorithms with a light-weight, FPGA-based implementation in mind and showed how to map them into efficient, but reconfigurable circuits.

We have designed an FPGA-based architecture for the runtime monitoring and analysis of important safety and performance properties on-board complex, intelligent, autonomous UAS. A combination of signal processing and filtering, LTL and MTL runtime observers, as well as Bayesian networks makes it possible to set up expressive, yet compact, no-overhead health models. We discussed the details of implementation of the temporal observers and the Bayesian networks on the FPGA hardware. By using efficient algorithms and data structures for the realization of the temporal observers [39] and arithmetic circuits [14] implemented as special purpose engines, our health model can be executed in real time; new or updated health models can be loaded onto the FPGA without having to re-synthesize its entire configuration in a time-consuming process.

Several case studies herein detailed our success using rt-R2U2 to analyze NASA's entire library of real flight data recorded by NASA's

Swift UAS. Playing back the recorded flight data as if rt-R2U2 were flying on-board the Swift, we were able to show that rt-R2U2 detected the real faults that occurred, such as a failure of the laser altimeter, while introducing no false positive fault identifications. We showed the correct operation of rt-R2U2 in nominal conditions as well, for example detecting the successful execution of flight commands. We also demonstrated modeling and analysis capability on a health model that assesses the (serial) communication between the main flight computer and an on-board fluxgate magnetometer. Using real data from an actual test flight of the NASA Swift UAS, we showed that rt-R2U2 could have quickly detected a configuration problem of the fluxgate magnetometer unit as the cause for a buffer overflow. Flying rt-R2U2 on this test flight would have saved NASA considerable time and money as the original problem grounded the aircraft for two days until the human operators could detect the root cause of the problem.

Following decommissioning of the Swift and examination of alternative test platforms, we successfully integrated rt-R2U2 into the architecture of NASA's DragonEye UAS. We documented our hardware integration, risk analysis, and initial testing efforts on-board the DragonEye. We assessed challenges with integration into this platform, including the potential for the Parallella board to overheat during a flight test. We analyzed data streams from the DragonEye, like with the Swift, and demonstrated both that we were monitoring the system health correctly and not producing false-positive identifications of faults.

We made considerable progress toward an in-the-air test-flight series and have a DragonEye UAS configured and ready to start testing as soon as test flights are allowed to resume at Moffett Field. Our accomplishments include hardware integration into the DragonEye UAS, a single initial test-flight to record data, a successful series of on-the-test-stand test flights that demonstrate correct operation of the UAS, and completion of the necessary approval process for flight testing. In the future, we plan to run a series of flight tests, first flying with nominal conditions to check for and eliminate any false-positives, then progressing to the controlled injection of faults to check that each fault is correctly identified in real time by rt-R2U2.

We plan to integrate rt-R2U2 into additional UAS platforms, including larger UAS platforms like the Swift, Viking, and Sierra UAS, and more configurable small UAS platforms. One of the challenges we ran into with integrating rt-R2U2 on-board the DragonEye UAS was the limited ability to fix problems during test flights due to the tight fit of the Parallella board and other components and the small access panels. During pre-flight testing for one indoor flight test we detected that an essential data cable must have a loose connection; since the connection was on the underside of a wing-mounted board, we had to disassemble and then re-assemble the autopilot to plug in the data cable, a labor-intensive task that required hours to complete. A more modular and easily reconfigurable platform would be a better choice for future small UAS tests.

The need for real-time, realizable, responsive, and unobtrusive system health management is not limited to UAS. In the future, we plan to integrate rt-R2U2 into many other embedded platforms, including rovers, robots, and small satellites like CubeSATs. While rt-R2U2 provides essential capabilities to enable such systems to operate intelligently and autonomously while also ensuring they operate safely, this need is not limited to totally autonomous air- and spacecraft. We envision integration into manned or remote-piloted air- and spacecraft as well. For example, rt-R2U2 could be used to simplify and improve the diagnosis capabilities of cockpit systems in commercial aircraft. It could also provide a valuable back-up to human eyes in remotely piloted systems, ensuring that the human operators become aware of any disruptions to system health. In addition to the current capability of identifying faults in real time, the output of rt-R2U2 could be connected to an on-board decision-making component, which could suggest or issue commands to respond to the detected faults, such as mitigative actions or emergency landing.

To better enable adaptation of rt-R2U2 for new platforms and streamline integration into different hybrid hardware/software systems, we plan to investigate formal techniques for automated or semi-automated synthesis of the rt-R2U2 components. In our report, we briefly examined the trade-offs in any compositional SHM framework like the one we have introduced 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? We plan to investigate automated assessment of questions like whether a small wobble in a data value should be filtered out via a standard analog filter, accepted by a reasonably lenient temporal logic observer, or flagged by the BN back-end but then monitored for a high enough frequency to indicate a real problem (perhaps in combination with other events) by a temporal logic observer taking as input the outputs of the BN. Investigating efficient design patterns for compositional system health management could help automate the process of mapping the types of checks we need to perform to their most efficient implementations in rt-R2U2.

Bibliography

1. Adapteva: Parallella board homepage, `http://www.parallella.org/`

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

3. APM Copter, D.R.: Apm wiki, `http://copter.ardupilot.com/wiki/apm25board_overview/`

4. Backasch, R., Hochberger, C., Weiss, A., Leucker, M., Lasslop, R.: Runtime verification for multicore SoC with high-quality trace data. ACM Trans. Des. Autom. Electron. Syst. 18(2), 18:1–18:26 (2013)

5. Barre, B., Klein, M., Soucy-Boivin, M., Ollivier, P.A., Hallé, S.: MapReduce for parallel trace validation of LTL properties. In: RV. Lecture Notes in Computer Science, vol. 7687. Springer Verlag (2012)

6. Barringer, H., Falcone, Y., Finkbeiner, B., Havelund, K., Lee, I., Pace, G.J., Rosu, G., Sokolsky, O., Tillmann, N. (eds.): Runtime Verification - First International Conference, RV 2010, Proceedings, Lecture Notes in Computer Science, vol. 6418. Springer Verlag (2010)

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

8. Basin, D., Klaedtke, F., Zălinescu, E.: Algorithms for monitoring real-time properties. In: RV. Lecture Notes in Computer Science, vol. 7186, pp. 260–275. Springer Verlag (2011)

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

10. Bauer, A., Leucker, M., Schallhart, C.: Runtime verification for LTL and TLTL. ACM Transactions on Software Engineering and Methodology 20, 14:1–14:64 (2011)

11. 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)

12. Colombo, C., Pace, G., Abela, P.: Safer asynchronous runtime monitoring using compensations. Formal Methods in System Design 41, 269–294 (2012)

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

14. Darwiche, A.: Modeling and reasoning with bayesian networks. In: Modeling and Reasoning with Bayesian Networks (2009)

15. Darwiche, A.: Modeling and Reasoning with Bayesian Networks. Cambridge University Press, 1st edn. (2009), ISBN: 0521884381

16. Divakaran, S., D'Souza, D., Mohan, M.R.: Conflict-tolerant real-time specifications in Metric Temporal Logic. In: TIME. pp. 35–42. IEEE Computer Society Press (2010)

17. Drusinsky, D.: The temporal rover and the ATG rover. In: SPIN. Lecture Notes in Computer Science, vol. 1885, pp. 323–330. Springer Verlag (2000)

18. Finkbeiner, B., Kuhtz, L.: Monitor circuits for LTL with bounded and unbounded future. In: RV, Lecture Notes in Computer Science, vol. 5779, pp. 60–75. Springer Verlag (2009)

19. Fischmeister, S., Lam, P.: Time-aware instrumentation of embedded software. IEEE Transactions on Industrial Informatics 6(4), 652–663 (2010)

20. Geilen, M.: An improved on-the-fly tableau construction for a real-time temporal logic. In: CAV. pp. 394–406 (2003)

21. Geist, J., Rozier, K.Y., Schumann, J.: Runtime observer pairs and bayesian network reasoners on-board fpgas: Flight-certifiable system health management for embedded systems. In: Runtime Verification - 5th International Conference, RV 2014, Toronto, ON, Canada, September 22-25, 2014. Proceedings. pp. 215–230 (2014), `http://dx.doi.org/10.1007/978-3-319-11164-3_18`

22. Havelund, K.: Runtime verification of C programs. In: TestCom/-FATES. pp. 7–22. Springer Verlag (2008)

23. 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)

24. 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)

25. Kleene, S.C.: Introduction to Metamathematics. North Holland, 11th edn. (1996), ISBN: 978-0720421033

26. Lichtenstein, O., Pnueli, A., Zuck, L.: The glory of the past. In: Logics of Programs, Lecture Notes in Computer Science, vol. 193, pp. 196–218. Springer Verlag (1985)

27. Lu, H., Forin, A.: The design and implementation of P2V, an architecture for zero-overhead online verification of software programs. Tech. Rep. MSR-TR-2007-99, Microsoft Research (2007)

28. Majzoobi, M., Pittman, R.N., Forin, A.: gnosis: Mining fpgas for verification (2011)

29. Maler, O., Nickovic, D., Pnueli, A.: On synthesizing controllers from bounded-response properties. In: CAV. Lecture Notes in Computer Science, vol. 4590, pp. 95–107. Springer Verlag (2007)

30. Maler, O., Nickovic, D., Pnueli, A.: Checking temporal properties of discrete, timed and continuous behaviors. In: Pillars of Comp. Science. pp. 475–505. Springer Verlag (2008)

31. 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)

32. 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)

33. 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`

34. Pearl, J.: A constraint propagation approach to probabilistic reasoning. In: UAI. pp. 31–42. AUAI Press (1985)

35. Pellizzoni, R., Meredith, P., Caccamo, M., Rosu, G.: Hardware runtime monitoring for dependable COTS-based real-time embedded systems. RTSS pp. 481–491 (2008)

36. Pike, L., Niller, S., Wegmann, N.: Runtime verification for ultra-critical systems. In: RV. Lecture Notes in Computer Science, vol. 7186, pp. 310–324. Springer Verlag (2011)

37. Pike, L., Wegmann, N., Niller, S., Goodloe, A.: Copilot: monitoring embedded systems. Innovations in Systems and Software Engineering 9(4), 235–255 (2013)

38. Reinbacher, T., Függer, M., Brauer, J.: Real-time runtime verification on chip. In: RV. Lecture Notes in Computer Science, vol. 7687, pp. 110–125. Springer Verlag (2012)

39. 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)

40. Reinbacher, T., Rozier, K.Y., Schumann, J.: Temporal-logic based runtime observer pairs for system health management of real-time systems. In: Tools and Algorithms for the Construction and Analysis of Systems - 20th International Conference, TACAS 2014, Held as Part of the European Joint Conferences on Theory and Practice of Software, ETAPS 2014, Grenoble, France, April 5-13, 2014. Proceedings. pp. 357–372 (2014), `http://dx.doi.org/10.1007/978-3-642-54862-8_24`

41. 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)

42. Schumann, J., Mengshoel, O.J., Mbaya, T.: 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. SMC-IT '11 (2011)

43. 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)

44. 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)

45. Tabakov, D., Rozier, K.Y., Vardi, M.Y.: Optimized temporal monitors for SystemC. Formal Methods in System Design 41(3), 236–268 (2012)

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

47. U.S. Department of Transportation: Risk assessment, `http://international.fhwa.dot.gov/riskassess/risk_hcm06_03.cfm`

Appendix A

Proofs of Correctness

Theorem A1 (Correctness of the Observer for $\neg\varphi$) *For any execution sequence $\langle T_\varphi \rangle$, the observer stated in Algorithm 7 implements $e^n \models \neg\varphi$.*

Algorithm 7 Observer for $\neg\varphi$.

```
1: At each new input $T_\varphi$:
2: $T_\xi \leftarrow (\neg T_\varphi.v, T_\varphi.\tau_e)$
3: return $T_\xi$
```

Proof. The theorem follows immediately from the definition of $e^n \models \neg\varphi$ and the definition of an execution sequence.

Theorem A2 (Correctness of the Observer for $◧_\tau \varphi$) *For any execution sequence $\langle T_\varphi \rangle$, the observer stated in Algorithm 8 implements $e^n \models ◧_\tau \varphi$.*

Algorithm 8 Observer for $◧_\tau \varphi$. Initially, $m_{\uparrow\varphi} = m_{\tau_s} = 0$.

```
1: At each new input $T_\varphi$:
2: $T_\xi \leftarrow T_\varphi$
3: if _|‾ transition of $T_\xi$ occurs then
4:     $m_{\uparrow\varphi} \leftarrow m_{\tau_s}$
5: end if
6: $m_{\tau_s} \leftarrow T_\varphi.\tau_e + 1$
7: if $T_\xi$ holds then
8:     if $m_{\uparrow\varphi} \leq (T_\xi.\tau_e - \tau)$ holds then
9:         $T_\xi.\tau_e \leftarrow T_\xi.\tau_e - \tau$
10:    else
11:        $T_\xi \leftarrow (␣, ␣)$
12:    end if
13: end if
14: return $T_\xi$
```

Proof. We first observe the equivalences

$$
\begin{aligned}
e^n \models ◧_\tau \varphi \quad &\Leftrightarrow \quad e^n \models \Box_{[0,\tau]} \varphi,\\
&\Leftrightarrow \quad e^n \models \neg(\mathbf{true}\,\mathcal{U}_{[0,\tau]} \neg \varphi),\\
&\Leftrightarrow \quad \neg(\exists i(i \geq n) : (i - n \in [0,\tau] \wedge e^i \models \neg\varphi \wedge \forall j(n \leq j < i) : \; e^j \models \mathbf{true})),\\
&\Leftrightarrow \quad \neg(\exists i(i \geq n) : (i - n \in [0,\tau] \wedge e^i \models \neg\varphi \wedge \mathbf{true})),\\
&\Leftrightarrow \quad \neg(\exists i(i \geq n) : (i - n \in [0,\tau] \wedge e^i \models \neg\varphi)),\\
&\Leftrightarrow \quad \forall i(i \geq n) : \neg(i - n \in [0,\tau] \wedge e^i \models \neg\varphi),\\
&\Leftrightarrow \quad \forall i(i \geq n) : (\neg(i - n \in [0,\tau]) \vee e^i \models \varphi),\\
&\Leftrightarrow \quad \forall i(i \geq n) : (i - n \in [0,\tau] \to e^i \models \varphi),\\
&\Leftrightarrow \quad \forall i : (i \in [n, n+\tau] \to e^i \models \varphi).
\end{aligned}
$$

Note that interval $[n, n+\tau]$ is never empty, since $n, \tau \in \mathbb{N}_0$. Therefore, the equivalences above holds iff a $\uparrow$ transition of φ occurred at a time at least n and no $\downarrow$ transition of φ occurred since then until time $n+\tau$ (ensured by lines 3 and 6 and the $\mathbf{valid}^{◧}(m_{\uparrow\varphi}, T_\xi, \tau)$ check in line 8 of Algorithm 8).

The theorem follows.

Theorem A3 (Correctness of the Observer for $\Box_J \varphi$) *For any execution sequence $\langle T_\varphi \rangle$, the observer stated in Algorithm 9 implements $e^n \models \Box_J \varphi$.*

Algorithm 9 Observer for $\Box_J \varphi$.

```
1: At each new input $T_\varphi$:
2: $T_\xi \leftarrow ◧_{\mathtt{dur}(J)}\, T_\varphi$
3: if ($T_\xi.\tau_e - \min(J) \geq 0$) then
4:     $T_\xi.\tau_e \leftarrow T_\xi.\tau_e - \min(J)$
5: else
6:     $T_\xi \leftarrow (␣, ␣)$
7: end if
8: return $T_\xi$
```

Proof. We first observe the equivalences

$$
\begin{aligned}
&e^n \models \Box_J \varphi\\
&\quad \Leftrightarrow e^n \models \neg\,(\mathbf{true}\,\mathcal{U}_{[\min(J),\max(J)]} \neg \varphi),\\
&\quad \Leftrightarrow \forall i(i \geq n) : (i - n \in [\min(J), \max(J)] \to e^i \models \varphi),\\
&\quad \Leftrightarrow \forall i : (i \in [n + \min(J), n + \max(J)] \to e^i \models \varphi). \qquad \text{(A1)}
\end{aligned}
$$

By Theorem A2 we have

$$e^n \models ◧_\tau \varphi \Leftrightarrow \forall i : (i \in [n, n+\tau] \to e^i \models \varphi).$$

With $\tau = \mathtt{dur}(J)$ we arrive at

$$\begin{aligned}
&e^n \models \blacksquare_{\mathtt{dur}(J)}\,\varphi,\\
&\quad\Leftrightarrow \forall i : (i \in [n, n + \mathtt{dur}(J)] \rightarrow e^i \models \varphi),\\
&\quad\Leftrightarrow \forall i : (i \in [n, n + \max(J) - \min(J)] \rightarrow e^i \models \varphi). \qquad \text{(A2)}
\end{aligned}$$

By the Equivalences A1 and A2 we observe that both $e^n \models \Box_J\,\varphi$ and $e^n \models \blacksquare_\tau\,\varphi$ require that φ holds for an interval of length $\max(J) - \min(J) = \mathtt{dur}(J)$, however, $e^n \models \Box_J\,\varphi$ requires that φ holds for an interval that is $\min(J)$ ahead (i.e., in the future) of $e^n \models \blacksquare_\tau\,\varphi$. Subtracting $\min(J)$ (equals to a shift into the past by $\min(J)$ time stamps) from

$$e^n \models \blacksquare_{\mathtt{dur}(J)}\,\varphi \Leftrightarrow \forall i : (i \in [n, n + \max(J) - \min(J)] \rightarrow e^i \models \varphi),$$

yields

$$\begin{aligned}
&\forall i : (i - \min(J) \in [n, n + \max(J) - \min(J)] \rightarrow e^i \models \varphi),\\
&\quad\Leftrightarrow \forall i : (i \in [n + \min(J), n + \max(J) - \min(J) + \min(J)] \rightarrow e^i \models \varphi),\\
&\quad\Leftrightarrow \forall i : (i \in [n + \min(J), n + \max(J)] \rightarrow e^i \models \varphi),\\
&\quad\Leftrightarrow \Box_J\,\varphi \qquad\qquad\qquad\qquad \text{(cf. Equation A1)}.
\end{aligned}$$

Since Algorithm 9 instantiates a $\blacksquare_\tau\,\varphi$ observer in line 2 and subtracts $\min(J)$ from the result, it establishes the required equivalence. The check in line 3 of Algorithm 9 prevents the observer from returning execution sequences where $T_\xi.\tau_e \notin \mathbb{N}_0$.

The theorem follows.

Theorem A4 (Correctness of the Observer for $\varphi \wedge \psi$) *For any two execution sequences $\langle T_\varphi \rangle$, $\langle T_\psi \rangle$, the observer stated in Algorithm 10 implements $e^n \models \varphi \wedge \psi$.*

Proof. To prove the correctness of Algorithm 10, it needs to be shown that both the truth value $T_\xi.v$ and the time stamp $T_\xi.\tau_e$ of the output tuple T_ξ, generated in line 14 of Algorithm 10, are correct – for arbitrary inputs.

a) Correctness of $T_\xi.v$. The proof is by showing that a correct output verdict $T_\xi.v$ of Algorithm 10 is equivalent to the result of a conjunction of the inputs encoded in Kleene logic [25]. We then enumerate the inputs by means of a truth table and verify that the proposed algorithm generates the correct outputs. Recall that the observer reads tuples (T_φ, T_ψ) from the two synchronization queues q_φ and q_ψ and that the verdicts $T_\varphi.v, T_\psi.v \in \{\mathbf{true}, \mathbf{false}\}$. Depending on the state of the synchronization queues, we distinguish the following cases:

Algorithm 10 Observer for $\varphi \wedge \psi$.

```
1: At each new input $(T_\varphi, T_\psi)$:
2: if $T_\varphi$ holds and $T_\psi$ holds and $q_\varphi \neq ()$ holds and $q_\psi \neq ()$ holds then
3:     $T_\xi \leftarrow (\mathbf{true}, \min(T_\varphi.\tau_e, T_\psi.\tau_e))$
4: else if $\neg T_\varphi$ holds and $\neg T_\psi$ holds and $q_\varphi \neq ()$ holds and $q_\psi \neq ()$ holds
   then
5:     $T_\xi \leftarrow (\mathbf{false}, \max(T_\varphi.\tau_e, T_\psi.\tau_e))$
6: else if $\neg T_\varphi$ holds and $q_\varphi \neq ()$ holds then
7:     $T_\xi \leftarrow (\mathbf{false}, T_\varphi.\tau_e)$
8: else if $\neg T_\psi$ holds and $q_\psi \neq ()$ holds then
9:     $T_\xi \leftarrow (\mathbf{false}, T_\psi.\tau_e)$
10: else
11:     $T_\xi \leftarrow (\textvisiblespace, \textvisiblespace)$
12: end if
13: dequeue$(q_\varphi, q_\psi, T_\xi.\tau_e)$
14: return $T_\xi$
```

Case (i): if both q_φ and q_ψ are non-empty (i.e., both elements in the input (T_φ, T_ψ) are available), the output is **true** only in case $T_\varphi.v = T_\psi.v = \mathbf{true}$ and **false** otherwise.

Case (ii): if both q_φ and q_ψ are empty, the input tuple (T_φ, T_ψ) is empty too, thus, the observer cannot produe a new output. We map this to a **maybe** output in Kleene logic representation.

Case (iii): if either q_φ or q_ψ is empty, one element of the input tuple (T_φ, T_ψ) is empty, and the result of the observer depends on the other, non-empty input.

We observe that with the encoding

$$a = \begin{cases} \mathbf{true} & \text{if} \quad T_\varphi.v = \mathbf{true} \ \wedge \ q_\varphi \neq (), \\ \mathbf{false} & \text{if} \quad T_\varphi.v = \mathbf{false} \ \wedge \ q_\varphi \neq (), \\ \mathbf{maybe} & \text{otherwise.} \end{cases}$$

$$b = \begin{cases} \mathbf{true} & \text{if} \quad T_\psi.v = \mathbf{true} \ \wedge \ q_\psi \neq (), \\ \mathbf{false} & \text{if} \quad T_\psi.v = \mathbf{false} \ \wedge \ q_\psi \neq (), \\ \mathbf{maybe} & \text{otherwise.} \end{cases}$$

the expected output verdict $T_\xi.v$ of a $\varphi \wedge \psi$ observer is exactly the result of $a \wedge b$ in Kleene logic.

Table A1 enumerates the possible inputs of the algorithm in terms of a truth table. For example, in case $T_\varphi.v$ is **false**, $T_\psi.v$ is **true**, synchronization queues q_φ and q_ψ are non-empty (see #2 in Table A1), the expected output is $a \wedge b = \mathbf{true} \wedge \mathbf{false} = \mathbf{false}$. In case $T_\varphi.v$ is **false**, $T_\psi.v$ is **true**, and queue q_φ is empty, and q_ψ is non-empty (see #10 in Table A1), the expected output is $a \wedge b = \mathbf{maybe} \wedge \mathbf{true} = \mathbf{maybe}$.

It remains to be shown that Algorithm 10 generates these outputs. We study the column "Outputs of Algorithm 10" of Table A1, which states $T_\xi.v$ as generated by Algorithm 10 and the corresponding line number of the respective assignments. For example, in case $T_\varphi.v$ is **false** and $T_\psi.v$ is **true** and synchronization queues q_φ and q_ψ are non-empty (see #2 in Table A1), the algorithm returns $T_\xi.v =$ **false**, matching the expected output.

We have shown the correctness of the truth value $T_\xi.v$ of the output tuple T_ξ of Algorithm 10 for all possible inputs; it remains to be shown that the corresponding time stamp $T_\xi.\tau_e$ of the output tuple T_ξ is correct too.

b) Correctness of $T_\xi.\tau_e$. For analogous arguments as above, in cases where the verdict of the computed output tuple $T_\xi.v$ is **maybe**, the corresponding time stamp $T_\xi.\tau_e$ is undefined too, see #7,8,10,12-16 in Table A1. For the remaining input conditions we distinguish the following two cases:

> *Case (i):* if either q_φ or q_ψ is empty and the verdict $T_\xi.v$ of the output tuple is **false**, the time stamp of the output is the time stamp of the non-empty element in the input tuple (T_φ, T_ψ), see #5,6,9,11 in Table A1.
>
> *Case (ii):* if neither q_φ nor q_ψ is empty, the time stamp of the output depends on the truth values of $T_\varphi.v$ and $T_\psi.v$, see #2-4 in Table A1. In the special case that both $T_\varphi.v$ and $T_\psi.v$ are **false** (#1 in Table A1), the time stamp of the output can be extended to the maximum of the time stamps found in the input tuple (T_φ, T_ψ), see #1 in Table A1.

For example, consider the queue contents $q_\varphi = ((\textbf{false}, 1), (\textbf{true}, 10))$ and $q_\psi = ((\textbf{false}, 5), (\textbf{true}, 8))$. When reading the input $((\textbf{false}, 1), (\textbf{false}, 5))$ the observer can already output $(\textbf{false}, 5)$; regardless of the truth values of T_φ for times $n \in [2, 5]$, the result will be **false**. Applying $\textbf{dequeue}(q_\varphi, q_\psi, 5)$ yields $q_\varphi = ((\textbf{true}, 10))$ and $q_\psi = ((\textbf{true}, 8))$.

For similar arguments, the scenario described in #2,3 of Table A1 requires to output the time stamp of the element in the input tuple (T_φ, T_ψ) whose truth value is **false**. If both $T_\varphi.v$ and $T_\psi.v$ are **true** (#4 in Table A1), the output can only be resolved until the minimum (i.e., the earlier) of the time stamps found in the input tuple (T_φ, T_ψ). For example, consider the queue content: $q_\varphi = ((\textbf{true}, 1), (\textbf{false}, 10))$ and $q_\psi = ((\textbf{true}, 5), (\textbf{false}, 8))$. When reading the input $((\textbf{true}, 1),(\textbf{true}, 5))$ the observer needs to output $(\textbf{true}, 1)$. The next input $((\textbf{false}, 10),(\textbf{true}, 5))$ generates the output $(\textbf{false}, 10)$, i.e., the output is **false** for times $n \in [2, 10]$. Results for the remaining cases are derived in a similar way.

	Inputs				Expected result		Outputs of Algorithm 10	
#	φ	ψ	q_φ	q_ψ	$T_\xi.v$	$T_\xi.\tau_s$	$(T_\xi.v, T_\xi.\tau_e)$	line#
1	0	0	0	0	0	max(time stamp φ, time stamp ψ)	$(\mathbf{false}, \max(T_\varphi.\tau_e, T_\psi.\tau_e))$	5
2	0	1			0	time stamp of φ	$(\mathbf{false}, T_\varphi.\tau_e))$	7
3	1	0			0	time stamp of ψ	$(\mathbf{false}, T_\psi.\tau_e))$	9
4	1	1			1	min(time stamp φ, time stamp ψ)	$(\mathbf{true}, \min(T_\varphi.\tau_e, T_\psi.\tau_e))$	3
5	0	0	0	1	0	time stamp of φ	$(\mathbf{false}, T_\varphi.\tau_e))$	7
6	0	1			0	time stamp of φ	$(\mathbf{false}, T_\varphi.\tau_e))$	7
7	1	0			?	-	(␣, ␣)	11
8	1	1			?	-	(␣, ␣)	11
9	0	0	1	0	0	time stamp of ψ	$(\mathbf{false}, T_\psi.\tau_e))$	9
10	0	1			?	-	(␣, ␣)	11
11	1	0			0	time stamp of ψ	$(\mathbf{false}, T_\psi.\tau_e))$	9
12	1	1			?	-	(␣, ␣)	11
13	0	0	1	1	?	-	(␣, ␣)	11
14	0	1			?	-	(␣, ␣)	11
15	1	0			?	-	(␣, ␣)	11
16	1	1			?	-	(␣, ␣)	11

Table A1: Enumeration of input combinations, expected results, and outputs of Algorithm 10. For brevity, we use the abbreviations: $\varphi = T_\varphi.v$, $\psi = T_\psi.v$, and write 0 for **false**, 1 for **true**, and ? for **maybe**. q_φ is set "1" iff $q_\varphi = ()$ and q_ψ is set "1" iff $q_\psi = ()$.

It remains to be shown that Algorithm 10 generates these time stamps. Again, we study the column "Outputs of Algorithm 10" of Table A1, which states $T_\xi.\tau_e$ as generated by Algorithm 10 and the corresponding line number of the respective assignment. For example, in case $T_\varphi.v$ is **false** and $T_\psi.v$ is **true** and synchronization queues q_φ, q_ψ are non-empty (see #2 in Table A1), the algorithm returns $T_\xi.\tau_e = T_\varphi.\tau_e$, matching the expected output. The same holds for the remaining cases.

We have shown the correctness of both the output verdict $T_\xi.v$ and the time stamp $T_\xi.\tau_e$ of the output tuple T_ξ of Algorithm 10 for all possible inputs.

The theorem follows.

Theorem A5 (Correctness of the Observer for $\varphi \mathcal{U}_J \psi$) *For any two execution sequences $\langle T_\varphi \rangle$, $\langle T_\psi \rangle$, the observer stated in Algorithm 12 implements $e^n \models \varphi \mathcal{U}_J \psi$.*

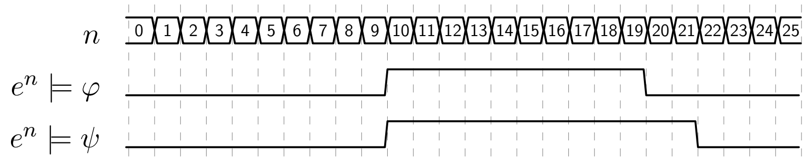
The observer for $\varphi \mathcal{U}_J \psi$, as stated in Algorithm 12, expects a tuple (T_φ, T_ψ) as input. Similar to the observer for $\varphi \wedge \psi$, T_φ of this tuple is an element from the execution sequence $\langle T_\varphi \rangle$ stored in synchronization queue q_φ and T_ψ of this tuple is an element from the execution sequence $\langle T_\psi \rangle$ stored in synchronization queue q_ψ. Input tuples are processed in a **lockstep mode** to ensure that the observer outputs only a single tuple at each run, thereby, avoiding additional output buffers, which would account for additional hardware resources. This lockstep mode is achieved by the following transformation on the input tuple (T_φ, T_ψ): (T_φ, T_ψ) is transformed into (possibly several) tuples $(T'_\varphi, T'_\psi), (T''_\varphi, T''_\psi), \ldots$, such

Algorithm 12 Observer for $\varphi\,\mathcal{U}_J\,\psi$. Initially, $m_{pre} = m_{\uparrow\varphi} = 0$, $m_{\downarrow\varphi} = -\infty$, and $p =$ **false**.

```
At each new input $(T_\varphi, T_\psi)$ in lockstep mode:
if _|‾ transition of $T_\varphi$ occurs then
    $m_{\uparrow\varphi} \leftarrow \tau_e - 1$
    $m_{pre} \leftarrow -\infty$
end if
if ‾|_ transition of $T_\varphi$ occurs and $T_\psi$ holds then
    $T_\varphi.v,\, p \leftarrow$ true, true
    $m_{\downarrow\varphi} \leftarrow \tau_e$
end if
if $T_\varphi$ holds then
    if $T_\psi$ holds then
        if $(m_{\uparrow\varphi} + \min(J) < \tau_e)$ holds then
            $m_{pre} \leftarrow \tau_e$
            return $(\mathbf{true}, \tau_e - \min(J))$
        else if $p$ holds then
            return $(\mathbf{false}, m_{\downarrow\varphi})$
        end if
    else if $(m_{pre} + \mathtt{dur}(J) \le \tau_e)$ holds then
        return $(\mathbf{false}, \max(m_{\uparrow\varphi}, \tau_e - \max(J)))$
    end if
else
    $p \leftarrow$ false
    if $(\min(J) = 0)$ holds then
        return $(T_\psi.v, \tau_e)$
    end if
    return $(\mathbf{false}, \tau_e)$
end if
return $(\textvisiblespace, \textvisiblespace)$
```

that $T'_\varphi.\tau_e = T'_\psi.\tau_e$ holds and for the time stamp $T''_\varphi.\tau_e$ of the next tuple (T''_φ, T''_ψ) it holds that $T''_\varphi.\tau_e = T'_\varphi.\tau_e + 1$.

The motivation for the lockstep mode stems from the intended hardware implementation of the observer. To illustrate, we assume the existence of a correct observer, possibly implemented in software, for $\varphi\,\mathcal{U}_{[2,3]}\,\psi$ and that the execution sequences stored in the two synchronization queues q_φ and q_ψ describe the executions $e^n \models \varphi$ and $e^n \models \psi$ over times $n \in [0, 25]$ as shown below:



In a non-lockstep mode implementation, the observer for $e^n \models \varphi \mathcal{U}_{[2,3]} \psi$ may read the following sequence of tuples (T_φ, T_ψ) from q_φ and q_ψ[A1]: $((\mathbf{false}, 9), (\mathbf{false}, 9))$, $((\mathbf{true}, 19), (\mathbf{true}, 19))$, $((\mathbf{false}, 21), (\mathbf{true}, 21))$, $((\mathbf{false}, 25), (\mathbf{false}, 25))$. The observer will produce the following outputs:

1. For input $((\mathbf{false}, 9), (\mathbf{false}, 9))$, the observer returns $(\mathbf{false}, 9)$.
2. For input $((\mathbf{true}, 19), (\mathbf{true}, 19))$, the observer returns $(\mathbf{true}, 17)$.
3. For input $((\mathbf{false}, 21), (\mathbf{true}, 21))$, the observer returns $(\mathbf{true}, 18)$ and $(\mathbf{false}, 21)$.
4. For input $((\mathbf{false}, 25), (\mathbf{true}, 25))$, the observer returns $(\mathbf{false}, 25)$.

To comply with our Responsiveness requirement, our observers need to ensure that any input tuple is processed within a tight time bound. This includes reading a new input tuple from the synchronization queues, calculating the output tuple, and committing this new tuple to the observer's output. This is feasible for inputs (1), (2), and (4) in the example above where one output tuple is generated at a time. For input (3), however, the observer needs to output two tuples at the same time.

To implement this functionality in hardware an additional output buffer to temporarily store the second tuple while the first one is committed is required. This accounts for an additional clock cycle to commit the second tuple $(\mathbf{false}, 21)$ to the output and additional hardware resources to implement and control this buffer. To avoid a blowup of the hardware design, we opted to design our $\varphi \mathcal{U}_J \psi$ observer to work on inputs given in lockstep mode. For input (3) from the example above, our implementation will transform the input $((\mathbf{false}, 21), (\mathbf{true}, 21))$ into $((\mathbf{false}, 20), (\mathbf{true}, 20))$ (3.1) and $((\mathbf{false}, 21), (\mathbf{true}, 21))$ (3.2) and calculate:

3.1 For input $((\mathbf{false}, 20), (\mathbf{true}, 20))$, the observer in Algorithm 12 returns $(\mathbf{true}, 18)$.

3.2 For input $((\mathbf{false}, 21), (\mathbf{true}, 21))$, the observer in Algorithm 12 returns $(\mathbf{false}, 21)$.

The lockstep mode, thus, helps us to guarantee that, for any input tuple, the observer is not required to output multiple tuples. This avoids additional hardware overhead for output buffers and meets our Unobtrusiveness requirement.

Proof. Theorem A5 holds if we can show that both directions of the statement "The observer for $\varphi \mathcal{U}_J \psi$ stated in Algorithm 12 returns $(\mathbf{true}, n)$ iff $e^n \models \varphi \mathcal{U}_J \psi$" hold:

[A1] To simplify the discussion (T_φ, T_ψ) is such that $T_\varphi.\tau_e = T_\psi.\tau_e$.

If: The observer for $\varphi \mathcal{U}_J \psi$ returns $(\mathbf{true}, n)$ if $e^n \models \varphi \mathcal{U}_J \psi$ holds.

Only If: $e^n \models \varphi \mathcal{U}_J \psi$ holds if the observer for $\varphi \mathcal{U}_J \psi$ returns $(\mathbf{true}, n)$.

If we show the correctness of both statements, Theorem A5 holds. We will start by proving the following lemma that helps us to simplify the proof of Theorem A5.

Lemma A6 (Unrolling) *The observer in Algorithm 12 decides the truth value of* $e^n \models \varphi \mathcal{U}_J \psi$*, where* $\min(J) > 0$*, at a time* n' *bounded by* $n' \leq n + \max(J)$*.*

Proof. From the definition of $e^n \models \varphi \mathcal{U}_J \psi$ we have

$$\begin{aligned}
&e^n \models \varphi \mathcal{U}_J \psi \\
&\Leftrightarrow \exists i (i \geq n) : (i - n \in J \wedge e^i \models \psi \wedge \forall j (n \leq j < i) :\ e^j \models \varphi), \\
&\Leftrightarrow \exists i (i \geq n) : (i - n \in [\min(J), \max(J)] \wedge e^i \models \psi \wedge \forall j (n \leq j < i) : \\
&\qquad e^j \models \varphi), \\
&\Leftrightarrow \exists i (i \geq n) : (i \in [n + \min(J), n + \max(J)] \wedge e^i \models \psi \wedge \forall j (n \leq j < i) : \\
&\qquad e^j \models \varphi).
\end{aligned} \tag{A3}$$

In order to build a correct observer algorithm, we can incrementally step through all i (i.e., τ_e) starting from n until we (a) find a location i where Equation A3 holds (then the result is $e^n \models \varphi \mathcal{U}_J \psi$), or (b) we have reached an $i > n + \max(J)$ where Equation A3 cannot hold anymore because $i \notin [n + \min(J), n + \max(J)]$ (then the result is $e^n \not\models \varphi \mathcal{U}_J \psi$).

We now show that when reaching $i = n + \max(J)$, the observer stated in Algorithm 12 has already decided the truth value of $e^n \models \varphi \mathcal{U}_J \psi$. We distinguish three cases depending on the value of i:

Case (i): $n \leq i < n + \min(J)$, since $i \notin [n + \min(J), n + \max(J)]$, Equation A3 does not hold, but might hold at a later i, i.e., at $i > n + \min(J)$. Observe that this is captured by the check at line 12 in Algorithm 12. For $n \leq i < n + \min(J)$ the check does not hold and the Algorithm will not output a verdict for $e^n \models \varphi \mathcal{U}_J \psi$, i.e., returns $(\text{␣}, \text{␣})$ on line 28.

Case (ii): $n + \min(J) \leq i \leq n + \max(J)$, since $i \in [n + \min(J), n + \max(J)]$, Equation A3 can hold if we find an i for which $e^i \models \psi$ and $\forall j (n \leq j < i) :\ e^j \models \varphi$ holds. We distinguish two cases: we first find an i where φ does not hold or we first find an i for which both $e^i \models \psi$ and $\forall j (n \leq j < i) :\ e^j \models \varphi$ hold. For the former, Equation A3 does not hold. The algorithm immediatelly returns $(\mathbf{false}, \tau_e)$ in line 26 (since T_φ does not hold). For the latter, Equation A3 holds, and the algorithm returns $(\mathbf{true}, \tau_e - \min(J))$ in line 14. By our assumptions, we have $\tau_e = i$ and $\tau_e - \min(J) = n$.

Note that the Algorithm returns a verdict at a time earlier than $i = n + \max(J)$ in both cases (i) and (ii).

Case (iii): $i = n + \max(J) + 1$, since $i \not\in [n + \min(J), n + \max(J)]$, Equation A3 does not hold, and cannot hold for any later i. Note that this is captured by the predicates in lines 18 and 19.

Combining the arguments of i-iii), we have that Algorithm 12 decides the truth value of $e^n \models \varphi \mathcal{U}_J \psi$ where $\min(J) > 0$ at a time n' not later than $n' \leq n + \max(J)$ in all cases.

The lemma follows.

We continue with the proof of Theorem A5. We first show that:

If: The observer for $\varphi \mathcal{U}_J \psi$ returns $(\mathbf{true}, n)$ if $e^n \models \varphi \mathcal{U}_J \psi$ holds. Assume by means of a contradiction that $e^n \models \varphi \mathcal{U}_J \psi$ does not hold and the observer returns $(\mathbf{true}, n)$. We observe that Algorithm 12 may return tuples T where $T.v = \mathbf{true}$ either in line 14 (case (i)) or in line 24 (case (ii)).

Case (i): the observer returns $(\mathbf{true}, \tau_e - \min(J))$ in line 14. We have witnessed that both T_φ and T_ψ held at time τ_e (lines 10 and 11) and that $(m_{\uparrow\varphi} + \min(J) < \tau_e)$ holds. Observe that this implies that $\forall j(\tau_e - \min(J) \leq j < \tau_e) :\ e^j \models \varphi$ holds too. Further, by the definition of $e^n \models \varphi \mathcal{U}_J \psi$ we have

$$\begin{aligned} e^n \models \varphi \mathcal{U}_J \psi \quad &\Leftrightarrow \quad \exists i(i \geq n) : (i - n \in J \\ &\qquad \wedge e^i \models \psi \wedge \forall j(n \leq j < i) :\ e^j \models \varphi), \\ &\Leftrightarrow \quad \exists i(i \geq n) : (i - n \in [\min(J), \max(J)] \\ &\qquad \wedge e^i \models \psi \wedge \forall j(n \leq j < i) :\ e^j \models \varphi). \end{aligned}$$

we may choose $i = \tau_e$ and substitute $n = \tau_e - \min(J)$. Combined with the observation from above, we arrive at

$$\begin{aligned} e^{\tau_e - \min(J)} \models \varphi \mathcal{U}_J \psi \quad &\Leftrightarrow \quad (\min(J) \in [\min(J), \max(J)] \wedge e^{\tau_e} \models \psi \wedge \mathbf{true}), \\ &\Leftrightarrow \quad \mathbf{true} \wedge e^{\tau_e} \models \psi \wedge \mathbf{true} \Leftrightarrow e^{\tau_e} \models \psi. \end{aligned}$$

Since we can reach line 14 only in case T_ψ holds (ensured by line 11) at time τ_e, we have $e^{(\tau_e - min(J))} \models \varphi \mathcal{U}_J \psi$, contradicting our assumption for case (i).

Case (ii): the observer returns $(\mathbf{true}, n)$ in line 24. We have witnessed that T_φ does not hold (the check in line 10 does not hold) at time τ_e and that $\min(J) = 0$ (line 23). By the definition of $e^n \models \varphi \mathcal{U}_J \psi$ (we substitute $\min(J) = 0$ and $i = \tau_e$):

$$\begin{aligned} &e^n \models \varphi \mathcal{U}_{[0,\max(J)]} \psi \\ &\Leftrightarrow \exists i(i \geq n) : (i - n \in J \wedge e^i \models \psi \wedge \forall j(n \leq j < i) :\ e^j \models \mathbf{false}), \\ &\Leftrightarrow \exists i(i \geq n) : (\tau_e - n \in [0, \max(J)] \wedge e^{\tau_e} \models \psi \wedge \forall j(n \leq j < \tau_e) : \\ &\qquad e^j \models \mathbf{false}), \end{aligned} \tag{A4}$$

we observe that $\varphi \mathcal{U}_{[0,\max(J)]}\, \psi$ can only hold (under the precondition that T_φ does not hold) in case we find a j that does not satisfy $(n \leq j < \tau_e)$, because the right hand side of the conjunction in Equation A4 vacuously holds – regardless of the truth value of φ. This is exactly the case when we choose $\tau_e = n$. Then, $\tau_e - n = 0$ and since $0 \in [0, \max(J)]$ holds, the left hand side of the conjunction in Equation A4 holds. We arrive at

$$e^{\tau_n} \models \varphi \mathcal{U}_{[0,\max(J)]}\, \psi \Leftrightarrow (\tau_e = n) : (\mathbf{true} \wedge e^{\tau_e} \models \psi \wedge \mathbf{true}).$$

Thus, in case $\min(J) = 0$ and $e^{\tau_e} \not\models \varphi$, the truth value of $e^{\tau_e} = \varphi \mathcal{U}_J\, \psi$ is equal to $e^{\tau_e} \models \psi$. This is ensured by line 23, contradicting our assumption in case (ii).

Since we arrived at a contradiction for both cases, we have shown that: *the observer for $\varphi \mathcal{U}_J\, \psi$ stated in Algorithm 12 returns $(\boldsymbol{true}, n)$ if* $e^n \models \varphi \mathcal{U}_J\, \psi$ holds.

To complete the proof it remains to be shown that:

Only If: $e^n \models \varphi \mathcal{U}_J\, \psi$ holds if the observer for $\varphi \mathcal{U}_J\, \psi$ returns $(\mathbf{true}, n)$. The proof is by induction on $n \in \mathbb{N}_0$.

Base Case $(n = 0)$: we consider the four possible truth value combinations of the input tuple (T_φ, T_ψ).

Case (i): assume both T_φ and T_ψ do not hold. We have that $e^0 \not\models \varphi$ and $e^0 \not\models \psi$. By substituting into the definition of $e^n \models \varphi \mathcal{U}_J\, \psi$ we get

$$\begin{aligned} & e^0 \models \varphi \mathcal{U}_J\, \psi \\ & \Leftrightarrow \exists i (i \geq 0) : (i - 0 \in [\min(J), \max(J)] \wedge e^i \models \psi \wedge \forall j (0 \leq j < i) : \\ & \qquad\qquad e^j \models \varphi). \end{aligned}$$

By our assumption $e^0 \not\models \varphi$, $\forall j (0 \leq j < i) :\ e^j \models \varphi$ evaluates to **true** iff $i = 0$. We distinguish two cases (a) $\min(J) = 0$ and (b) $\min(J) > 0$. Since T_φ does not hold, we only consider lines 22-26 of Algorithm 12.

(a) $e^0 \not\models \varphi \mathcal{U}_J\, \psi$ since $0 \in [0, \max(J)]$ holds, however, $e^0 \not\models \psi$. We observe that the algorithm returns $(\mathbf{false}, 0)$ for this case in line 24.

(b) $e^0 \not\models \varphi \mathcal{U}_J\, \psi$ since $0 \notin [\min(J), \max(J)]$ with $\min(J) > 0$. We observe that the algorithm returns $(\mathbf{false}, 0)$ for this case in line 26.

By the arguments from above, the induction base follows in this case.

Case (ii): assume T_φ does not hold and T_ψ holds. We have that $e^0 \not\models \varphi$ and $e^0 \models \psi$. For analogous arguments as in case (i), we distinguish the two cases (a) $\min(J) = 0$ and (b) $\min(J) > 0$.

(a) $e^0 \models \varphi \mathcal{U}_J\, \psi$ since $0 \in [0, \max(J)]$ holds and $e^0 \not\models \psi$. We observe that the algorithm returns $(\mathbf{true}, 0)$ for this case in line 24.

(b) $e^0 \not\models \varphi \mathcal{U}_J \psi$ since $0 \notin [\min(J), \max(J)]$ with $\min(J) > 0$. We observe that the algorithm returns $(\textbf{false}, 0)$ for this case in line 26.

By the arguments from above, the induction base follows in this case.

Case (iii): assume T_φ holds and T_ψ does not hold. We have that $e^0 \models \varphi$ and $e^0 \not\models \psi$. We distinguish two cases (a) $\min(J) = \max(J) = 0$ and (b) $\min(J) > 0$. Since T_φ holds, we will only consider lines 10-20 of Algorithm 12.

(a) $e^0 \not\models \varphi \mathcal{U}_J \psi$ since with $i \in [0, 0]$ we have $e^0 \not\models \psi$. Initially, we have $m_{pre} = 0$ and $m_{\uparrow\varphi} = 0$. Therefore, the condition in line 18 holds and the algorithm returns $(\textbf{false}, 0)$ for this case in line 19.

(b) Since $e^0 \models \varphi$ and $\min(J) > 0$, the validity of $e^0 \models \varphi \mathcal{U}_J \psi$ cannot be determined at time $n = 0$ as we can choose an arbitrary $i \in [\min(J), \max(J)]$ and need to test $e^i \models \psi$ and $\forall j (0 \leq j < i) : e^j \models \varphi$ at times $i > n$. The induction base follows by Lemma A6.

By the arguments from above, the induction base follows in this case.

Case (iv): assume both T_φ and T_ψ hold. We have that $e^0 \models \varphi$ and $e^0 \models \psi$. As in cases (i) and (ii), we distinguish the two cases (a) $\min(J) = 0$ and (b) $\min(J) > 0$.

(a) $e^0 \models \varphi \mathcal{U}_J \psi$ since with $i \in [0, \max(J)]$ we choose $i = 0$ we have $e^0 \models \psi$. Since φ holds, we must have witnessed a _|‾ transition of φ (by definition, for times prior to 0, φ does not hold). The check in line 2 holds and we have $m_{\uparrow\varphi} = -1$ and $m_{pre} = -\infty$. The condition in line 12 holds $(-1 + 0 < -0)$ and the algorithm returns $(\textbf{true}, 0)$ for this case in line 14.

(b) Since $e^0 \models \varphi$ and $\min(J) > 0$, the validity of $e^0 \models \varphi \mathcal{U}_J \psi$ cannot be determined at time $n = 0$. For analogous arguments as in case (iii.b), the induction base follows by Lemma A6.

By the arguments from above, the induction base follows in this case.

Induction Step $(n - 1 \rightarrow n)$ with the induction hypothesis: assume that $e^{n-1} \models \varphi \mathcal{U}_J \psi$ if the observer for $\varphi \mathcal{U}_J \psi$ returns $(\textbf{true}, n - 1)$ holds for $n - 1 \geq 0$. We will show that it holds for n, too. We consider the same cases (i) to (iv) for the truth values of the input (T_φ, T_ψ) as in the base case.

Case (i): assume both T_φ and T_ψ do not hold. For analogous arguments as in the base case, the algorithm returns $(\textbf{false}, n)$ in either line 24 or 26.

By the arguments from above, the induction step follows in this case.

Case (ii): assume T_φ does not hold and T_ψ holds. We distinguish two cases for φ: a ‾|_ transition of φ did not (ii.a) or did occur at time n (ii.b).

(ii.a) For the same arguments as in the base case, the algorithm returns $(\textbf{true}, n)$ in case $\min(J) = 0$ (line 24) and $(\textbf{false}, n)$ if $\min(J) > 0$ (line 26). Thus, the induction step follows in this case.

(ii.b) We have that $p = \textbf{true}$, $m_{\downarrow\varphi} = n$, and $T_\varphi.v = \textbf{true}$. Clearly, the algorithm only executes lines 11-17 in this case. We need to distinguish the following two cases (ii.b.1) The latest _⌈ transition of φ occurred at a time earlier than $n - \min(J)$ and (ii.b.1) the latest _⌈ transition of φ did occur at a time later than or equal to $n - \min(J)$.

- (ii.b.1) By assumption for this case we have $m_{\uparrow\varphi} < n - \min(J)$ and by the semantics of the $\varphi \mathcal{U}_J \psi$ operator we know that the $e^n \models \varphi \mathcal{U}_J \psi$ holds up to time $n - \min(J)$. We observe that in this case, the check in line 12 holds and the algorithm returns $(\textbf{true}, n - \min(J))$ in line 14. The induction step follows in this case.
- (ii.b.2) By assumption for this case and the semantics of the $\varphi \mathcal{U}_J \psi$ operator we know that the $e^n \not\models \varphi \mathcal{U}_J \psi$ holds up to time n. Intuitively, the number of time stamps we saw φ to be true was shorter than $\min(J)$. We observe that in this case, the check in line 12 does not hold. Since p is **true** the check on line 15 holds and the algorithm returns $(\textbf{false}, n)$ in line 16. The induction step follows in this case.

By the arguments from above, the induction step follows in all three cases.

Case (iii): assume T_φ holds and T_ψ does not hold. We distinguish two cases for φ: a _⌈ transition of φ did not (iii.a) or did occur at time n (iii.b).

(iii.a) We distinguish two cases: $\min(J) = \max(J) = 0$ (iii.a.1) and $(\min(J) > 0) \wedge (\max(J) > 0)$ (iii.a.2).

- (iii.a.1) By the assumption $\texttt{dur}(J) = 0$ and $m_{pre} \leq n$ trivially holds, the algorithm returns $(\textbf{false}, n)$ in line 19. Thus, the induction step follows in this case.
- (iii.a.2) Suppose that the predicate in line 18 holds. In this case we observe that there is no previous $i : (\tau_e - \texttt{dur}(J) \leq i \leq \tau_e)$ for that the algorithm returned $(\textbf{true}, i - \min(J))$ in line 14. If this would be the case we would have set m_{pre} to i in line 13 and the predicate in line 18 would not hold anymore. By the definition of $e^n \models \varphi \mathcal{U}_J \psi$ we have

$$e^n \models \varphi \mathcal{U}_J \psi \Leftrightarrow \exists i (i \geq n) : \\ (i - n \in J \wedge e^i \models \psi \wedge \forall j (n \leq j < i) : \; e^j \models \textbf{false})$$

and with the observation from above, $e^n \models \varphi \mathcal{U}_J \psi$ can only hold for an $i > \tau_e$. This implies that $e^n \models \varphi \mathcal{U}_J \psi$ does not hold for an n up to $\tau_e - \max(J)$. Since the algorithm returns $(\textbf{false}, n - \max(J))$ in line 19, the induction step follows in this case. In case we have witnessed a $\_\!\!\ulcorner$ transition of φ in the meantime we have $m_{\uparrow\varphi} \geq n - \max(J)$. Then, by the semantics of the $\varphi \mathcal{U}_J \psi$ operator, $e^n \models \varphi \mathcal{U}_J \psi$ cannot hold until a time stamp n that is equal to the time stamp of the $\_\!\!\ulcorner$ transition of φ, stored in $m_{\uparrow\varphi}$. In this case, the algorithm returns $(\textbf{false}, m_{\uparrow\varphi})$ in line 19 and the induction step follows.

(iii.b) We have that $m_{\uparrow\varphi} = n - 1$, and $m_{pre} = -\infty$. Then, $(m_{pre} + \texttt{dur}(J) \leq n)$ holds. We distinguish two cases: $\min(J) = \max(J) = 0$ (iii.b.1) and $(\min(J) > 0) \wedge (\max(J) > 0)$ (iii.b.2).

(iii.b.1) By our assumption $\min(J) = \max(J) = 0$ we arrive at

$$e^n \models \varphi \mathcal{U}_{[0,0]} \psi \Leftrightarrow \exists i (i \geq n) : \\ (i - n \in [0,0] \wedge e^i \models \psi \wedge \forall j (n \leq j < i) : \; e^j \models \varphi).$$

For $n = i$, $i - n \in [0,0]$ holds, however, by our assumption for this case (T_φ holds and T_ψ does not hold), we immediately have that $e^i \not\models \psi$ and thus $e^n \not\models \varphi \mathcal{U}_{[0,0]} \psi$. Note that the algorithm returns $(\textbf{false}, \max(n-1, n))$ in line 19, which simplifies to $(\textbf{false}, n)$. The induction step follows in this case.

(iii.b.2) By assumption $\min(J) > 0$ and since $\max(J) \geq \min(J)$, the algorithm returns $(\textbf{false}, n - 1)$ in line 19. The induction step follows in this case.

By the arguments from above, the induction step follows in both cases.

Case (iv): assume both T_φ and T_ψ hold. We distinguish two cases for φ: a $\_\!\!\ulcorner$ of φ did not (iv.a) or did occur at time n (iv.b).

(iv.a) We distinguish two cases $\min(J) = 0$ (iv.a.1) and $\min(J) > 0$ (iv.a.2).

(iv.a.1) $(m_{\uparrow\varphi} + \min(J) < n)$ holds and the algorithm returns $(\textbf{true}, n)$. The induction step follows in this case.

(iv.a.2) $(m_{\uparrow\varphi} + \min(J) < n)$ only holds if the latest $\_\!\!\ulcorner$ transition occurred at least $\min(J)$ time units in the past. If this is the case, the algorithm returns $(\textbf{true}, n - \min(J))$. By similar arguments as in the base case, the induction step follows in this case. Suppose that the latest $\_\!\!\ulcorner$ transition of φ occurred at a time later than $n - \min(J)$. Then, the induction step follows by Lemma A6.

(iv.b) We have that $m_{\uparrow\varphi} = n - 1$ and $m_{pre} = -\infty$. We distinguish two cases $\min(J) = 0$ (iv.b.1) and $\min(J) > 0$ (iv.b.2).

  (iv.b.1) $(m_{\uparrow\varphi} + \min(J) < n)$ holds and the algorithm returns $(\mathbf{true}, n)$. The induction step follows in this case.

  (iv.b.2) $(m_{\uparrow\varphi} + \min(J) < n)$ does not hold. The induction step follows by Lemma A6.

By the arguments from above, the induction step follows in both cases.

The theorem follows.

Appendix B

Proofs of Complexity Results

Theorem B1 (Space Complexity of Asynchronous Observers) *The respective asynchronous observer for a given MTL specification φ has a space complexity, in terms of memory bits, bounded by $(2 + \lceil \log_2(n) \rceil) \cdot (2 \cdot m \cdot p)$, where m is the number of binary observers (i.e., $\varphi \wedge \psi$ or $\varphi \mathcal{U}_J \psi$) in φ, p is the worst-case delay of a single predecessor chain in* $\mathrm{AST}(\varphi)$*, and $n \in \mathbb{N}_0$ is the time stamp it is executed.*

Proof. We first make the following observations:

a) The asynchronous observer algorithms for unary MTL operators, i.e., $◧_{\tau}\, \varphi$ (Algorithm 8), $◧_{\tau}\, \varphi$ (Algorithm 9), and $\neg\, \varphi$ (Algorithm 7), are memory-less, i.e., do not use synchronization queues.

b) The asynchronous observer algorithms for binary MTL operators, i.e., $\varphi \wedge \psi$ (Algorithm 10) and $\varphi \mathcal{U}_J \psi$ (Algorithm 12) use two synchronization queues, q_φ and q_ψ. The sizes $|q_\varphi|$ and $|q_\psi|$ are assigned in step **MA3** of the synthesis procedure and depend on the time bounds assigned to the observers to compute their subformulas φ and ψ.

For example, the size of the synchronization queues of the observers required to evaluate the specification $◧_{100}\, \varphi_1 \wedge ◧_{10}\, \psi_1$ depends on the time bounds 100 and 10 assigned to the subformulas φ_1 and ψ_1. The algorithm for assigning queue sizes assigns $|q_{\psi_1}| = 100$ and $|q_{\varphi_1}| = 10$. Now suppose that subformula φ_1 is computed by another observer (e.g., $\varphi_1 := \neg ◧_{50}\, \varphi_{11}$), then $|q_{\psi_1}| = 100 + 50 = 150$.

In the general case, for an arbitrary MTL specification φ, the maximum queue size assigned by the algorithm for assigning queue sizes equals to the weight of the longest path in $\mathrm{AST}(\varphi)$; the weight on the edges is the value computed by $\mathbf{wcd}(\boxast)$, where $\boxast$ is the observer for the respective subformula of φ. We write p to denote the weight of this longest path. For example, the longest path in $\mathrm{AST}(\varphi)$ of $(◧_{100}\, (\neg ◧_{50}\, \varphi_{11})) \wedge (◧_{140}\, \psi_1)$ is 150. Consequently, all other queue sizes are equal or less than p. With the number of observers for binary operators in φ being equal to m, the total number of queues created for φ is, by observation b), equal to $2 \cdot m$. Then, the total size of all queues is bounded by $2 \cdot m \cdot p$. Recall that, a single element $T = (v, \tau_e)$ in a synchronization queue accounts for $w = \lceil \log_2(n) \rceil + 2$ bits. We need $\lceil \log_2(n) \rceil$ bits to store the time stamp $T.\tau_e$ and two additional bits to encode the three valued verdict $T.v$.

For a given MTL specification φ, we thus arrive at a worst-case space complexity, in terms of memory bits, of $(2 + \lceil \log_2(n) \rceil) \cdot (2 \cdot m \cdot p)$ for an asynchronous observer for φ.

The theorem follows.

Theorem B2 (Time Complexity of Asynchronous Observers) *The respective asynchronous observer for a given MTL specification φ has an asymptotic time complexity of $\mathcal{O}\Big(\log_2 \log_2 \max(p,n) \cdot d\Big)$, where p is the maximum worst-case-delay of any observer in $\mathrm{AST}(\varphi)$, d the depth of AST(φ), and $n \in \mathbb{N}_0$ the time stamp it is executed.*

Proof. As shown in [**?**] one can construct circuits that perform addition of two integers of bit complexity $w \in \mathbb{N}$ within time $\mathcal{O}(\log_2(w))$. Subtraction and relational operators as required by the asynchronous observer algorithms can be built around adders. We observe that, when $\mathrm{Add}(\langle a\rangle, \langle b\rangle, c)$ is a ripple carry adder for arbitrary length unsigned vectors $\langle a\rangle$ and $\langle b\rangle$ and c the carry in, then a subtraction of $\langle a\rangle - \langle b\rangle$ is equivalent to $\mathrm{Add}(\langle a\rangle, \langle \overline{b}\rangle, 1)$, where $\overline{b}$ denotes the bitwise negation of vector b. Relational operators can be built around adders in a similar way, for example, as described in [**?**, Chap. 6].

Since evaluating any of the conditionals and predicates (for example, the check in line 8 in Algorithm 8) occuring in the asynchronous observer algorithms at time $n \in \mathbb{N}_0$ requires addition of integers of bit complexity at most $\max(\log_2(p), \log_2(n))$, we arrive at an asymptotic time complexity of $\mathcal{O}(\log_2 \log_2 \max(p,n))$ for any of the proposed asynchronous observers, executed at time n.

For a given MTL specification φ, we can determine its depth d by the number of nodes of the longest-path in the parse tree of φ. We then arrive at an asymptotic time complexity of $\mathcal{O}(d \cdot \log_2 \log_2 \max(p,n))$ for an asynchronous observer for φ.

The theorem follows.

Theorem B3 (Circuit-Size Complexity of Synchronous Observers) *For a given MTL formula φ, the circuit to monitor $\widehat{\mathbf{eval}}(\varphi)$ has a circuit-size complexity bounded by $11 \cdot m$, where m is the number of observers in AST(φ).*

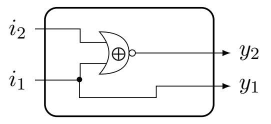
Proof. We want to show that the circuit required to implement a synchronous observer to monitor an arbitrary MTL specification φ has a circuit-size complexity [**?**] bounded by $11 \cdot m$, where m is the number of observers in $\mathrm{AST}(\varphi)$. This statement holds, if we can show that any of the synchronous observers for $\widehat{\mathbf{eval}}(\neg\,\varphi)$, $\widehat{\mathbf{eval}}(\varphi \wedge \psi)$, $\widehat{\mathbf{eval}}(\blacksquare_{\tau}\,\varphi)$, $\widehat{\mathbf{eval}}(\Box_J\,\varphi)$, and $\widehat{\mathbf{eval}}(\varphi\,\mathcal{U}_J\,\psi)$ can be built with at most 11 two-input gates. The circuits in Figure B1 show that any of the synchronous observers can be built with at most 11 two-input gates.

The theorem follows.

$\widehat{\mathbf{eval}}\,(\neg\,\varphi)$

$y_1 = i_1$
$y_2 = \neg(i_1 \oplus i_2)$

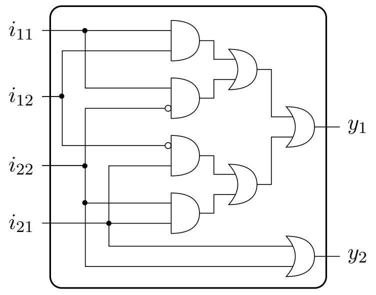
two-input gates: 1



$\widehat{\mathbf{eval}}\,(\varphi \wedge \psi)$

$y_1 = (i_{11} \wedge i_{12}) \vee (i_{11} \wedge \neg i_{22}) \vee (\neg i_{12} \wedge i_{21}) \vee (i_{21} \wedge i_{22})$
$y_2 = i_{12} \vee i_{22}$

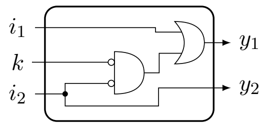
two-input gates: 8



$\widehat{\mathbf{eval}}\,(◧_\tau\,\varphi)$

$y_1 = (\neg k \wedge \neg i_2) \vee i_1$
$y_2 = i_2$
$k = 1$ if $\tau = 0$ and $k = 0$ othw.

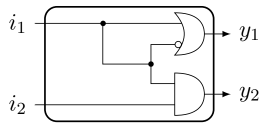
two-input gates: 2



$\widehat{\mathbf{eval}}\,(\Box_J\,\varphi)$

$y_1 = (\neg i_1 \vee i_1)$
$y_2 = (i_1 \wedge i_2)$

two-input gates: 2



$\widehat{\mathbf{eval}}\,(\varphi\,\mathcal{U}_J\,\psi)$

$y_1 = i_{11} \vee (\neg i_{12} \wedge i_{21}) \vee (\neg i_{12} \wedge i_{22}) \vee (\neg i_{12} \wedge \neg k) \vee (i_{21} \wedge i_{22})$
$y_2 = i_{11} \vee (\neg i_{12} \wedge i_{21} \wedge i_{22})$
$k = 1$ if $\min(J) = 0$ and $k = 0$ othw.

two-input gates: 11

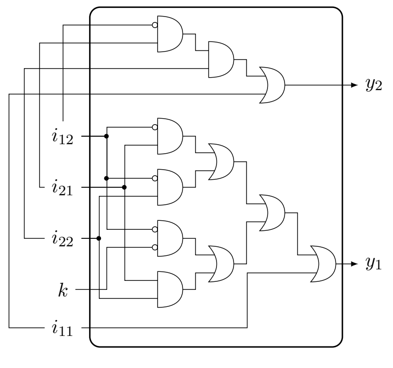


Figure B1: Mapping of synchronous MTL observers to circuits of two-input gates.

Theorem B4 (Circuit-Depth Complexity of Synchronous Observers) *For a given MTL formula φ, the circuit to monitor $\widehat{\mathbf{eval}}\,(\varphi)$ has a circuit-depth complexity of $4 \cdot d$.*

Proof. We want to show that the circuit for a synchronous observer to monitor an arbitrary MTL formula φ has a circuit-depth complexity [**?**] bounded by $4 \cdot d$, where d is the depth of AST(φ). This statement holds, if we can show that any of the synchronous observers can be built with a circuit of depth at most 4. From Figure B1 we can observe the depth of these circuits:

1. $\widehat{\mathbf{eval}}\,(\neg\,\varphi)$ circuit depth: 1
2. $\widehat{\mathbf{eval}}\,(\varphi\,\wedge\,\psi)$ circuit depth: 3
3. $\widehat{\mathbf{eval}}\,(◧_{\tau}\,\varphi)$ circuit depth: 2
4. $\widehat{\mathbf{eval}}\,(\Box_J\,\varphi)$ circuit depth: 1
5. $\widehat{\mathbf{eval}}\,(\varphi\,\mathcal{U}_J\,\psi)$ circuit depth: 4

The theorem follows.

Appendix C

Simulation Results

In what follows, we will discuss simulation runs we recorded from a full-fledged VHDL Register Transfer Level (RTL) hardware simulation of the deployment of the rt-R2U2 to the Swift UAS. In this simulation, the rt-R2U2 runs with a clock frequency of 100 MHz and new sensor data is provided from the UAS with a frequency of 10 Hz. The hardware design processes 37,418 individual sensors readings (i.e, there are 37,418 individual laser altimeter readings, 37,418 individual barometric altimeter readings, ...). Table. C1 summarizes the relevant signals required to understand the simulation traces.

Discussion of the simulation trace in Fig. C1. At the cursor position (red, right), the time of the RTC (signal s_rtc_timestamp) equals to $n = 2619$ and the UAS on-board sensors indicate: an increase in the baro-metric altitude signal (see input signal baro_altitude in category Swift UAS sensor data), an increase in the laser altitude signal (see signal laser_altitude), a positive vertical velocity (see signal vertical_velocity), and a significant pitching of the UAS (see signal euler_pitch_angle). The atomic propositions as calculated by runtime observers of the rt-R2U2 capture this behavior: signal sigma_s_b (baro alt up) evaluates to **true**, i.e., the respective hardware observer of the rt-R2U2 determined that $e^{2619} \models \sigma_{S_{B\uparrow}}$ holds. Similarly, the other atomic propositions (as shown in Fig. C1) of the specification are evaluated to: $e^{2619} \not\models \sigma_{S_{B\downarrow}}$, $e^{2619} \models \sigma_{S_{L\uparrow}}$, and $e^{2619} \not\models \sigma_{S_{L\downarrow}}$. Additionally (not explicitly mentioned in Fig. C1) the example uses the atomic propositions: $\sigma_{S_{P\uparrow}} = (\mathsf{pitch} \geq 5°)$ and $\sigma_{S_{P\downarrow}} = (\mathsf{pitch} \leq 2°)$ to monitor a significant up/down pitching of the UAS (from the IMU sensors). $\sigma_{S_{V\uparrow}} = (\mathsf{vel\_up} \geq 2\frac{m}{s})$ and $\sigma_{S_{V\downarrow}} = (\mathsf{vel\_up} < -2\frac{m}{s})$ to monitor a significant up/down velocity of the UAS (from the IMU sensors). $\sigma_{S_{ct}} = (\mathsf{cmd} == \text{takeoff})$ and $\sigma_{S_{cl}} = (\mathsf{cmd} == \text{land})$ to monitor if takeoff/land commands were received from the ground station. $\sigma_{S_h} = (Alt_L \geq 600ft)$ to monitor if the laser altimeter of the UAS indicates an altitude greater then 600 ft (the intended flight height). The verdicts computed by the respective hardware observers of the rt-R2U2 for these atomic propositions are $e^{2619} \models \sigma_{S_{P\uparrow}}$, $e^{2619} \not\models \sigma_{S_{P\downarrow}}$, $e^{2619} \not\models \sigma_{S_{ct}}$, $e^{2619} \not\models \sigma_{S_{cl}}$, and $e^{2619} \models \sigma_{S_h}$.

Discussion of the simulation trace in Fig. C2. Initially, all inputs to the altimeter health model indicate an increasing altitude (i.e., $e^n \models \sigma_{S_{B\uparrow}}$, $e^n \models \sigma_{S_{L\uparrow}}$, and $e^n \models \varphi_{S_{S\uparrow}}$). The posterior marginals Pr(baro_alt=OK $\mid \pi \models \{\sigma_{S_{B\uparrow}}, \sigma_{S_{L\uparrow}}, \varphi_{S_{S\uparrow}}\}$), and Pr(baro_alt=BAD $\mid \pi \models \{\sigma_{S_{B\uparrow}}, \sigma_{S_{L\uparrow}}, \varphi_{S_{S\uparrow}}\}$), and Pr(laser_alt=OK $\mid \pi \models \{\sigma_{S_{B\uparrow}}, \sigma_{S_{L\uparrow}}, \varphi_{S_{S\uparrow}}\}$), and Pr(laser_alt=BAD $\mid \pi \models \{\sigma_{S_{B\uparrow}}, \sigma_{S_{L\uparrow}}, \varphi_{S_{S\uparrow}}\}$), as calculate by higher level reasoning module of the rt-R2U2, show a high likelihood (belief) of both a healthy laser altimeter reading and a healthy barometer altimeter reading. Then, at the cursor position (red), due to the outage of the laser altimeter, $e^n \not\models \sigma_{S_{L\downarrow}}$ holds and indicates a *decrease* in altitude, while the other inputs to the altimeter health model disagree and indicate an *increase* (i.e., $e^n \models \sigma_{S_{B\uparrow}}, e^n \models \varphi_{S_{S\uparrow}}$ still hold). This is revealed by the new health assessment computed by the rt-R2U2: we see a significant drop in the health assessment of the laser altimeter reading (signal Pr(laser_alt=OK $\mid \pi \models \{\sigma_{S_{B\uparrow}}, \sigma_{S_{L\uparrow}}, \varphi_{S_{S\uparrow}}\}$)), while the belief in a healthy altimeter reading remains high (signal Pr(baro_alt=OK $\mid \pi \models \{\sigma_{S_{B\uparrow}}, \sigma_{S_{L\uparrow}}, \varphi_{S_{S\uparrow}}\}$)).

Table C1: Interpretation of the simulation signals in Fig. C1 and Fig. C2: *RTC = Real Time Clock*

Signal Name	**Interpretation**
rt-R2U2 System Signals	
s_clk	system clock of the rt-R2U2
s_reset_n	asynchronous reset of the rt-R2U2 (issued when low)
RTC related Signals	
s_rtc_clock	clock signal generated by the RTC
s_rtc_timestamp	counter value of the RTC (i.e., time stamp n)
Inputs: Sensor RAW data as transferred over the communication bus of the Swift UAS	
baro_altitude	altitude in ft as measured by the onboard baro-metric altimeter
laser_altitude	altitude in ft as measured by the onboard laser altimeter
vertical_velocity	vertical velocity in $\frac{m}{s}$ as measured by the onboard IMU
euler_pitch_angle	euler pitch angle in degree as measured by the onboard IMU
takeoff_command	set if Swift UAS received *takeoff* command from ground station
land_command	set if Swift UAS received *land* command from ground station
Outputs: Atomic Propositions of the specification, calculated by the rt-R2U2	
sigma_S_b	baro-metric altitude: truth values of $e^n \models \sigma_{S_{B\uparrow}}$ and $e^n \models \sigma_{S_{B\downarrow}}$
sigma_S_l	laser altitude: truth values of $e^n \models \sigma_{S_{l\uparrow}}$ and $e^n \models \sigma_{S_{L\downarrow}}$
sigma_S_v	vertical velocity: truth values of $e^n \models \sigma_{S_{V\uparrow}}$ and $e^n \models \sigma_{S_{V\downarrow}}$
sigma_S_p	pitch angle: truth values of $e^n \models \sigma_{S_{P\uparrow}}$ and $e^n \models \sigma_{S_{P\downarrow}}$
sigma_S_ct	command start: truth values of $e^n \models (\mathsf{cmd} == start)$
sigma_S_cl	command land: truth values of $e^n \models (\mathsf{cmd} == land)$
sigma_S_h	laser altitude: truth values of $e^n \models Alt_l \geq 600$
Outputs: Relevant signals of the runtime observer part of the rt-R2U2	
phi_1 (sync)	output of synchronous observer for flight rule φ_1
$\hookrightarrow$ maybe	set if output of the observer $\widehat{\mathbf{eval}}(\varphi_1)$ is **maybe**, cleared otherwise.
$\hookrightarrow$ value	set if output of the observer $\widehat{\mathbf{eval}}(\varphi_1)$ is **true**, cleared if $\widehat{\mathbf{eval}}(\varphi_1)$ is **false**
phi_1 (async)	output of asynchronous observer for flight rule φ_1
$\hookrightarrow$ value	set if verdict of the output tuple $T_{\varphi_1}.\mathsf{v} =$ **true**, cleared otherwise.
$\hookrightarrow$ time	time stamp of the output tuple ($T_{\varphi_1}.\tau_e$)
phi_2 (sync)	output of synchronous observer for flight rule φ_2
$\hookrightarrow$ maybe	set if output of the observer $\widehat{\mathbf{eval}}(\varphi_2)$ is **maybe**, cleared otherwise.
$\hookrightarrow$ value	set if output of the observer $\widehat{\mathbf{eval}}(\varphi_2)$ is **true**, cleared if $\widehat{\mathbf{eval}}(\varphi_2)$ is **false**
phi_2 (async)	output of asynchronous observer for flight rule φ_2
$\hookrightarrow$ value	verdict of the output tuple ($T_{\varphi_2}.\mathsf{v}$)
$\hookrightarrow$ time	time stamp of the output tuple ($T_{\varphi_2}.\tau_e$)
phi_S_s_up (async)	output of asynchronous observer for $\varphi_{S_{s\uparrow}}$
$\hookrightarrow$ value	set if verdict of the output tuple $T_{\varphi_{S_{s\uparrow}}}.\mathsf{v} =$ **true**, cleared otherwise.
$\hookrightarrow$ time	time stamp of the output tuple ($T_{\varphi_{S_{s\uparrow}}}.\tau_e$)
phi_S_s_down (async)	output of asynchronous observer for $\varphi_{S_{s\downarrow}}$
$\hookrightarrow$ value	set if verdict of the output tuple $T_{\varphi_{S_{s\downarrow}}}.\mathsf{v} =$ **true**, cleared otherwise.
$\hookrightarrow$ time	time stamp of the output tuple ($T_{\varphi_{S_{s\downarrow}}}.\tau_e$)
Outputs: Relevant signals of the higher level reasoning part of the rt-R2U2 (only in Fig. C2)	
Pr(baro_alt=OK $\mid pi \models \varphi$)	posterior marginal (likelihood of a good barometric altimeter reading)
Pr(baro_alt=BAD $\mid pi \models \varphi$)	posterior marginal (likelihood of a bad barometric altimeter reading)
Pr(laser_alt=OK $\mid pi \models \varphi$)	posterior marginal (likelihood of a good laser altimeter reading)
Pr(laser_alt=BAD $\mid pi \models \varphi$)	posterior marginal (likelihood of a bad laser altimeter reading)

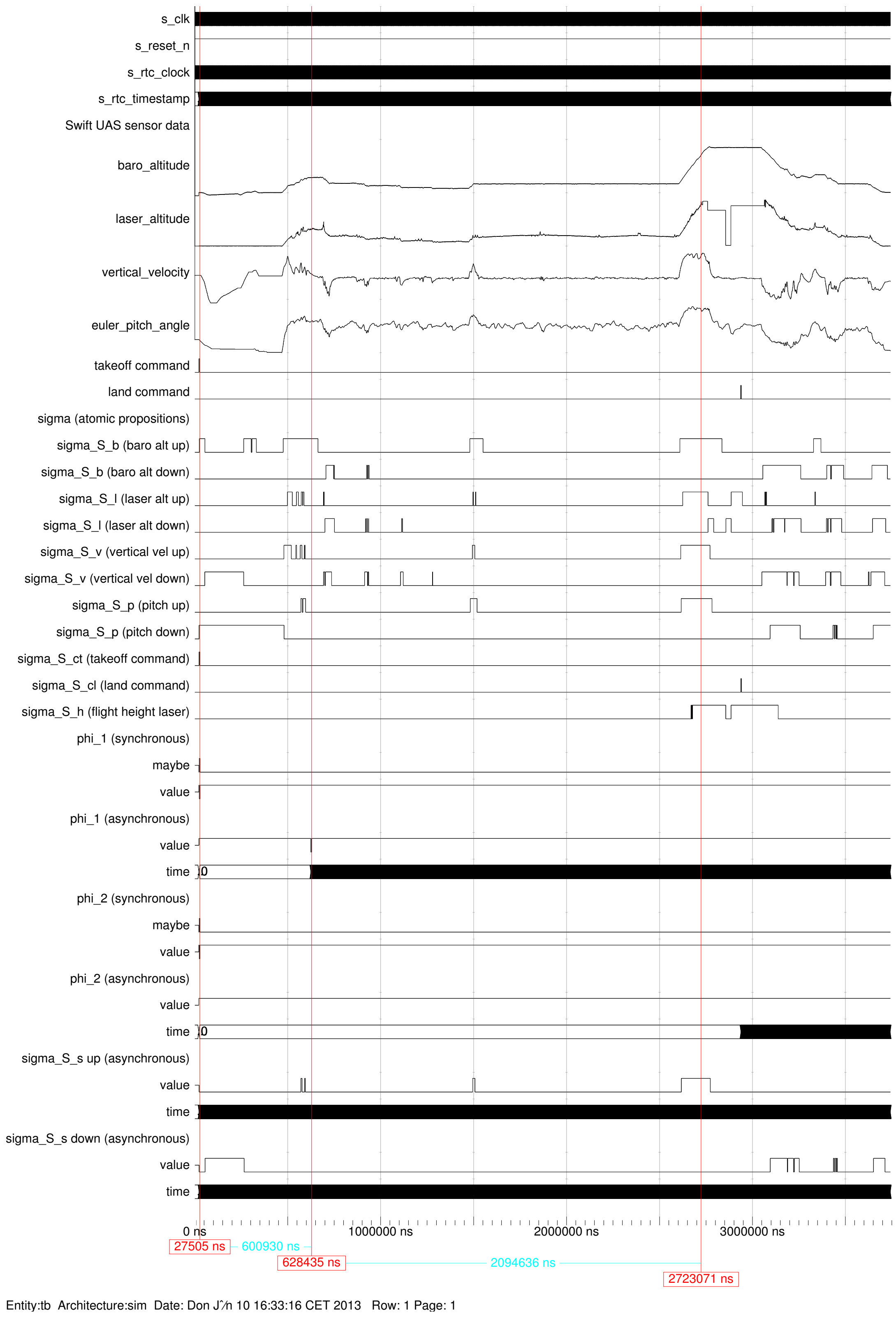


Figure C1: Hardware simulation traces for a complete test-flight data of the Swift UAS.

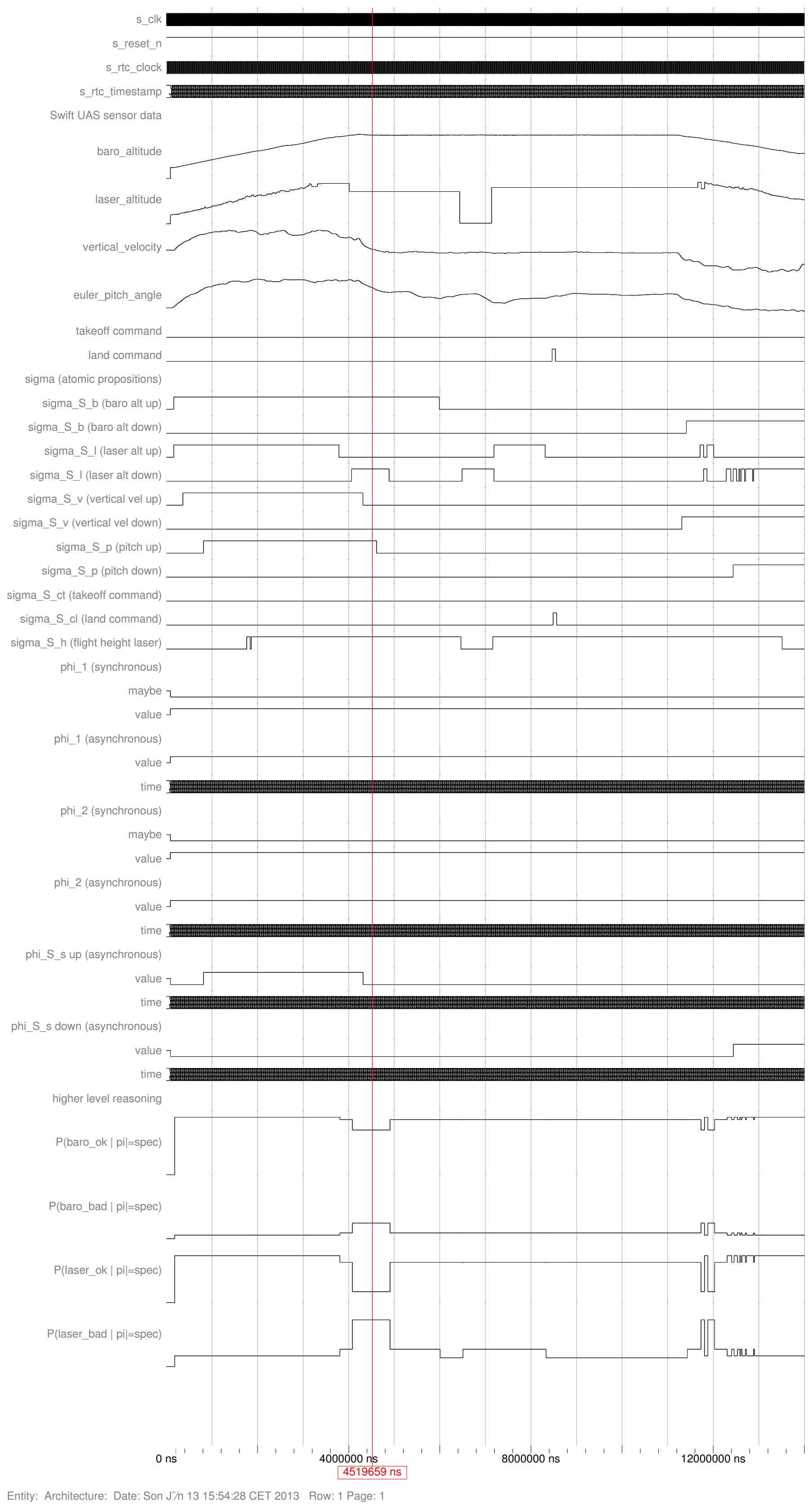


Figure C2: A section (laser altimeter outage) of the simulation traces with health assessment.

Appendix D

Detailed Risk Analysis

D.1 Risks and Mitigation—Mechanical

D.1.1 Center of Gravity

Description: Due to the additional payload, the Center of Gravity of the DragonEye will change.
Severity/Likelihood:d/A
Mitigation/Protection: Since there are already small weights in the cone we remove some of the weights to minimize the effect.

D.1.2 Nosecone Separation on hard landing

Description: In case of a hard landing, the Nosecone could be separated from the rest of the DragonEye. This could cause a hardware damage of the Parallella board or the APM, or the Power regulator since they are connected via wires. It could then be possible that the Power cables are shorted.
Severity/Likelihood:d/A
Mitigation/Protection:

- Nose is fixed securely.

D.1.3 Vibration

Description: The Vibrations could cause loose connectors. Also, the uSD card of the Parallella could become loose.
Severity/Likelihood: a/A
Mitigation/Protection: The SD-Card and connectors, Jumpers will be secured with RTV silicone.

D.2 Risks and Mitigation—Power

D.2.1 System Power Loss

Description: Short in Parallella, or intense computations might quickly drain flight battery
Severity/Likelihood: b/A
Mitigation/Protection: The current drawn by the Parallella board is limited by a 4A Fuse.
The DragonEye uses a battery monitor, a failsafe mechanism forces the DragonEye to RTL if the battery voltage is low

- RTV the jumper and board connectors, prevent connectors from failing in flight
- Fuse in power line to Parallella board
- Noticeable low-down of engines before power failure; switch to RC-mode

D.2.2 Excessive Power Consumption

Description: The Parallella board could either because of intense calculations or for unknown reasons drain a lot of power from the battery.
Severity/Likelihood: b/A
Mitigation: The current drawn by the Parallella board is limited by a 4A Fuse
The DragonEye uses a battery monitor, a failsafe mechanism forces the DragonEye to RTL if the battery voltage is low

- Measure in lab under full-load before the flight

D.2.3 Burn of Power cable/connector

Description: Burn of Power cable/connector due to short circuit
Severity/Likelihood: a/A
Mitigation/Protection: Fuses

D.2.4 Poor Power Quality for Parallella Board

Description: If the Power is noisy, a malfunction of the Parallella board is possible
Severity/Likelihood: a/B
Mitigation/Protection:

- Measure in lab before the flight
- Optional: Separate Power Regulator

D.3 Risks and Mitigation—Thermal

D.3.1 Parallella Board Too Hot

Description: Parallella board is getting too hot (above $70°C$)
Severity/Likelihood: a/C
Mitigation:

- Software on Parallella (XTemp) performs automatic shutdown of Parallella. No damage, but loss of evaluation/monitoring data from after the shutdown.
- Existing holes in nose-cone to higher airflow.

- Larger heat sink helps reduce temperature
- No aircraft changes

D.4 Risks and Mitigation—EMI and Signal Interference

D.4.1 Signal Interference: UART

Description: Additional UART connection can interfere with the APM Hardware/Software
Severity/Likelihood: a/B
Mitigation/Protection:

- APM-RX UART is disconnected
- Tests in the lab

D.4.2 RF Interference

Description: The Parallella board can introduce EMI Interference into the APM or vice versa leading to unexpected behavior.
Severity/Likelihood: b/B
Mitigation/Protection:

- Lab Testing
- Range testing as part of pre-flight
- 2 command links or separate frequencies (900MHz + 2.4MHz) - results in loss of one frequency, requiring GCS to take control on another frequency

D.5 Risks and Mitigation—FSW Interference

D.5.1 SW Interference UART Connection

Description: The additional connected UART could Interfere with the Software.
Severity/Likelihood: a/B
Mitigation/Protection:

- Review of SW Changes
- APM-RX UART is disconnected
- Low-Level SW failsafe mechanism automatically hands over control to the Operator
- Hardware MUX can be used to take over control of the DragonEye

D.5.2 FSW crash to due unspecified bug

Description: FSW crash due unspecified bug.
Severity/Likelihood: a/B-C
Mitigation/Protection:

- Review of SW Changes
- Low-Level SW failsafe mechanism automatically hands over control to the Operator
- Hardware MUX can be used to take over control of the DragonEye

D.5.3 Bad Task Timing due to modified Scheduler/Tasks

Description: Since the scheduler/tasks are instrumented, delays are introduced. If the delay is too big, the execution of the tasks can be delayed or some tasks might even starve since there is not enough time to run them.
Severity/Likelihood: a/B
Mitigation/Protection:

- Instrumentation of scheduler is limited to single copy by reference operations to minimize delay
- Review of modified Scheduler
- Since the monitoring gathers statistical data, this should be detected during HITL-Test
- Low-Level SW failsafe mechanism automatically hands over control to the Operator
- Hardware MUX can be used to take over control of the DragonEye

D.5.4 Time overrun in Monitoring Task

Description: Since the Scheduler is not preemptive, the additional monitoring task could overrun and delay other tasks or even cause starvation.
Severity/Likelihood: a/B
Mitigation/Protection:

- Packets are only sent non-blocking
- Packets are only sent when output ring buffer has enough free space
- Low-Level SW failsafe mechanism automatically hands over control to the Operator
- Hardware MUX can be used to take over control of the DragonEye

D.5.5 Time overrun in FSW Task

Description: Since some tasks are instrumented, delays are introduced. If the delay is too big, some other tasks might starve depending on the amount of introduced delay. Since the Scheduler is not preemptive, a long running task could trigger the Low-Level SW failsafe or all other task could be delayed.
Severity/Likelihood: a/B
Mitigation/Protection:

- Instrumentation of tasks was limited to single copy by reference operations to minimize delay
- Review of modified tasks
- Low-Level SW failsafe mechanism automatically hands over control to the Operator
- Hardware MUX can be used to take over control of the DragonEye

D.5.6 Inconsistent variable values in buffers

Description: due to different update rates in tasks, values of variables in the monitoring buffer might be inconsistent.
Severity/Likelihood: a/C
Mitigation/Protection: Since the monitoring rate is lower than rates of most tasks, averaging of values is performed by the monitoring software on the Parallella board. This step eliminated transient effects.

D.5.7 Interference with Failsafe modes

Description: For some reason the Low-Level SW failsafe mechanism could not operate correctly.
Severity/Likelihood: a/B
Mitigation/Protection:

- Hardware MUX can be used to take over control of the DragonEye

D.5.8 Mode-confusion for GCS or pilot

Description: Unexpected/erroneous behavior of the FSW could get the DragonEye into a mode, which is unexpected to the RC pilot. E.g., the AC flies a straight line as it should be commanded by the autopilot, but unknown to the operator, the autopilot has disengaged.
Severity/Likelihood: b/C
Mitigation/Protection: Operation at sufficient altitude to provide RC pilot ample time to react.

D.6 Risks and Mitigation—Operational

D.6.1 Failure in FSW

D.6.2 Risks in static failure-injected Operation

Description: Static injected failure can cause an unstable state of the AC when switched from RC-mode to autopilot. Could even be unable to launch. **Mitigation**:

- careful HIL evaluation of static failure scenarios
- safe transition points at an altitude, which gives RC pilot enough time to react
- prior reduction of failure magnitudes to ensure AC stability

D.6.3 Dynamic failure-injection: Transition on RC recovery

Description: Taking over control from autopilot to RC mode could cause undesired transient effects.

D.6.4 Dynamic failure-injection: Unrecoverable AC state

Description: Upon activation of an injected fault, the AC goes into an unrecoverable state.

- switch-over to RC control
- fault injection only at safe altitude

Appendix E

Software Annotations for ArduPlane

E.1 Modified Files

The following source files of the Arduino autopilot software have been added or modified.

AP-Rtr2u2 here are the framework and configuration header file and C++ file to implement the monitoring task and buffer transmission.

Arduplane.pde This file has been modified in the places: (1) include the appropriate header file, (2) register objects for access during initialization, and (3) add the monitoring task to the scheduler.

setup.pde This file has been modified in the places: (1) include the appropriate header file, (2) emit signal that EEPROM is going to be erased.

config.h This file has been modified in one place: define `SERIAL3_BAUD` as 115200, a constant to store the baud rate of rt-r2u2.

system.pde This file has been modified in one place: reset the UART C baud rate to `SERIAL3_BAUD` after parameter load.

FollowMe.pde This file has been modified in 4 places: (1) declare the channel, (2) set GCS_Mavlink library's comm 1 port to UART 2.

GCS_MAVLink.h This file has been modified in 6 places: (1) increase the number of MAVLink buffers, (2) declare MAVLink stream for ground control communication, (3) check the number of bytes that can be written to the stream, (4) write a byte to the stream, (5) check the number of bytes that can be read from the stream, (6) read a byte from the stream.

GCS_Mavlink.pde This file has been modified in 4 places: (1) init, set the port for the MAVLink channel, (2) forward unknown messages to the other link if there is one, (3) send a low priority formatted message to the GCS, (4) send airspeed calibration data.

AP_Scheduler.cpp This file has been modified in 5 places: (1) emit the signal that one tick has passed, (2) emit different signals when running one tick.

UARTDriver.h This file has been modified in one place: declare two variables to store the number of time that interrupt service routine is called in both transmitter and receiver. `AVRUARTDriverHandler` is modified to updated these two variables.

Scheduler.cpp This file has been modified in one place: send error message to rt-r2u2.

Semaphores.cpp This file has been modified in 5 places: (1) emit the error signal that semaphore has not been taken, (2) emit the error signal that failing to get semaphore, (3) emit the error signal that a blocking semaphore timeout was reached.

Storage.cpp This file has been modified in one place: emit the signal of writing to the parameter section of the EEPROM since last transmission.

AP_InertialSensor_MPU6000.cpp This file has been modified in 2 places: register the static address of the two variables: `_last_sample_time_micros` and `_spi_sem`.

AntennaTracker/GCS_Mavlink.pde This file has been modified in 2 places: (1) init, set the port for the MAVLink channel, (2) send a low priority formatted message to the GCS.

Makefiles the AP-Rtr2u2 files have been added to the makefile.

Name	cat	type	description
RTR2U2_LOW_LEVEL_OS			
s_boot_detected	G	uint8	new communication
s_clear_boot_detected		uint8	clear the reboot flag after transmitted
s_tick_counter		uint8	last value of scheduler tick counter
s_free_memory		uint8	free memory size
s_num_of_uart_isr_tx		uint8	number of calls to transmitter's ISR
s_num_of_uart_isr_rx		uint8	number of calls to receiver's ISR
s_num_of_scheduler_tick_overflow		uint8	number of scheduler tick overflow
s_task_i_num_runnable		uint8	number of times task i can run
s_task_i_num_run		uint8	number of times task i has run
s_all_tasks_zero_counters		uint8	clear all counters after a transmission
s_task_i_num_overrun		uint8	number of times task i has overrun
s_task_i_slipped		uint8	number of times task i has been slipped
s_task_i_delay		uint8	maximum delay
s_failsafe_cnt		uint8	number of times falling into sw-failsafe state
s_wipe_eeprom_cnt		uint8	number of times EEPROM has been erased

s_reset_eeprom_cnts		uint8	number of times resetting EEPROM monitor counter
s_write_eeprom_param_sect_cnts		uint8	number of times writing to the param section
RTR2U2_LOW_LEVEL_SENS_COMM			
s_compass_last_update		uint8	last update compass
s_baro_last_update		uint8	last update barometer
s_gps_lock_status		uint8	GPS Lock Status
s_gps_last_fix_time		uint8	last gps fix(time stamp)in ms
s_ins_last_sample_time		uint8	Last Sample time of the Inertial Sensor in us
s_spi_blocked_sem_cnt		uint8	Counter how often failed to get SPI Semaphore
s_spi_timeout_sem_cnt		uint8	Counter how often a Blocking Semaphore Time out was reached
s_spi_wrong_give_sem_cnt		uint8	Counter how often a Semaphore was tried to give without having it
RTR2U2_COMMAND_COMM			
s_last_heartbeat	G	int16	the time when the last HEARTBEAT message arrived from a GCS in ms
RTR2U2_BEHAVIORAL			
s_current_loc_alt		uint8	Current Location Altitude
s_current_loc_lat		uint8	Current Location Latitude
s_current_loc_lng		uint8	Current Location Longitude
s_gps_alt		uint8	GPS Altitude
s_gps_num_sats		uint8	Number of visible satellites
s_gps_ground_speed_cm		uint8	GPS Ground speed in cm/s
s_airspeed_m		uint8	airspeed in m/s
s_baro_alt		uint8	Barometric Altitude
s_baro_climb_rate		uint8	Barometer climb rate in m/s (positive=going up)
s_home_alt		uint8	Altitude of the home location
s_home_lat		uint8	Latitude of the home location
s_home_lng		uint8	Longitude of the home location
s_next_wp_alt		uint8	Altitude of the next Waypoint
s_next_wp_lat		uint8	Latitude of the next Waypoint
s_next_wp_lng		uint8	Longitude of the next Waypoint
s_control_mode		uint8	Mode of the Plane
RTR2U2_BATTERY			
s_battery_voltage		uint8	battery voltage
s_battery_low_voltage		uint8	battery low voltage

Table E2: Current variables monitored

RTR2U2_EN	enable monitoring
RTR2U2_LOW_LEVEL_OS	add subset
RTR2U2_LOW_LEVEL_SENS_COMM	add subset
RTR2U2_LOW_LEVEL_ACTUATOR_COMM	add subset
RTR2U2_COMMAND_COMM	add subset
RTR2U2_INTERNAL_CALCULATIONS	add subset
RTR2U2_BEHAVIORAL	add subset
RTR2U2_BATTERY	add subset

Table E3: #defines to control rt-R2U2 monitoring

Appendix F

List of Publications and Presentations

F.1 Publications

- Thomas Reinbacher, Kristin Y. Rozier, and Johann Schumann. "Temporal-Logic Based Runtime Observer Pairs for System Health Management of Real-Time Systems." In *Tools and Algorithms for the Construction and Analysis of Systems (TACAS)*, volume 8413 of Lecture Notes in Computer Science (LNCS), pages 357–372, Springer-Verlag, April, 2014.

- Johannes Geist, Kristin Yvonne Rozier, and Johann Schumann. "Runtime Observer Pairs and Bayesian Network Reasoners On-board FPGAs: Flight-Certifiable System Health Management for Embedded Systems." In *Runtime Verification (RV14)*, Springer-Verlag, September 22-25, 2014.

- Johann Schumann, Kristin Y. Rozier, Thomas Reinbacher, Ole J. Mengshoel, Timmy Mbaya, and Corey Ippolito. "Towards Real-time, On-board, Hardware-supported Sensor and Software Health Management for Unmanned Aerial Systems." In *International Journal of Prognostics and Health Management (IJPHM)*. (Invited journal paper; To appear)

F.2 Presentations

- K.Y. Rozier: "No More Helicopter Parenting: Intelligent Autonomous Unmanned Aerial Systems." NASA Ames' premier seminar series, the Directors Colloquium, special edition in honor of NASA Ames' 75th Anniversary celebration, by invitation of the Office of the Chief Scientist. NASA Ames Research Center, Moffett Field, California, June 10, 2014.

- J. Schumann: "Towards on-board, hardware-supported Sensor and Software Health Management for UAS." FORTISS, Technische Universität München, Germany, June 2014.

REPORT DOCUMENTATION PAGE	*Form Approved* *OMB No. 0704–0188*

1. REPORT DATE *(DD-MM-YYYY)* 01-05-2015	**2. REPORT TYPE** Technical Memorandum	**3. DATES COVERED** *(From - To)* 11/2013–12/2014

4. TITLE AND SUBTITLE
Intelligent Hardware-Enabled Sensor and Software Safety and Health Management for Autonomous UAS

5a. CONTRACT NUMBER

5b. GRANT NUMBER

5c. PROGRAM ELEMENT NUMBER

6. AUTHOR(S)
Kristin Y. Rozier, Johann Schumann, and Corey Ippolito

5d. PROJECT NUMBER

5e. TASK NUMBER

5f. WORK UNIT NUMBER

7. PERFORMING ORGANIZATION NAME(S) AND ADDRESS(ES)
NASA Ames Research Center
Moffett Field, California 94035-1000

8. PERFORMING ORGANIZATION REPORT NUMBER
L–

9. SPONSORING/MONITORING AGENCY NAME(S) AND ADDRESS(ES)
National Aeronautics and Space Administration
Washington, DC 20546-0001

10. SPONSOR/MONITOR'S ACRONYM(S)
NASA

11. SPONSOR/MONITOR'S REPORT NUMBER(S)
NASA/TM–2015–218817

12. DISTRIBUTION/AVAILABILITY STATEMENT
Unclassified-Unlimited
Subject Category 38
Availability: NASA STI Program (757) 864-9658

13. SUPPLEMENTARY NOTES
An electronic version can be found at http://ntrs.nasa.gov.

14. ABSTRACT
Unmanned Aerial Systems (UAS) can only be deployed if they can effectively complete their mission and respond to failures and uncertain environmental conditions while maintaining safety with respect to other aircraft as well as humans and property on the ground. We propose to design a real-time, onboard system health management (SHM) capability to continuously monitor essential system components such as sensors, software, and hardware systems for detection and diagnosis of failures and violations of safety or performance rules during the flight of a UAS. Our approach to SHM is three-pronged, providing: (1) real-time monitoring of sensor and software signals; (2) signal analysis, preprocessing, and advanced on-the-fly temporal and Bayesian probabilistic fault diagnosis; (3) an unobtrusive, lightweight, read-only, low-power hardware realization using Field Programmable Gate Arrays (FPGAs) in order to avoid overburdening limited computing resources or costly re-certification of flight software due to instrumentation. No currently available SHM capabilities (or combinations of currently existing SHM capabilities) come anywhere close to satisfying these three criteria yet NASA will require such intelligent, hardwareenabled sensor and software safety and health management for introducing autonomous UAS into the National Airspace System (NAS). We propose a novel approach of creating modular building blocks for combining responsive runtime monitoring of temporal logic system safety requirements with model-based diagnosis and Bayesian network-based probabilistic analysis. Our proposed research program includes both developing this novel approach and demonstrating its capabilities using the NASA Swift UAS as a demonstration platform.

15. SUBJECT TERMS

16. SECURITY CLASSIFICATION OF: a. REPORT	b. ABSTRACT	c. THIS PAGE	**17. LIMITATION OF ABSTRACT**	**18. NUMBER OF PAGES**	**19a. NAME OF RESPONSIBLE PERSON**
U	U	U	UU		STI Information Desk (email: help@sti.nasa.gov)
					19b. TELEPHONE NUMBER *(Include area code)* (757) 864-9658

Standard Form 298 (Rev. 8/98)
Prescribed by ANSI Std. Z39.18