Temporal-Logic Based Runtime Observer Pairs for System Health Management of Real-Time Systems*

Thomas Reinbacher[1], Kristin Y. Rozier[2], and Johann Schumann[3]

[1] Vienna University of Technology, Austria, treinbacher@ecs.tuwien.ac.at
[2] NASA Ames Research Center, Moffett Field, CA, USA, Kristin.Y.Rozier@nasa.gov
[3] SGT, Inc., NASA Ames, Moffett Field, CA, USA, Johann.M.Schumann@nasa.gov

Abstract. We propose a real-time, Realizable, Responsive, Unobtrusive Unit (rt-R2U2) to meet the emerging needs for System Health Management (SHM) of new safety-critical embedded systems like automated vehicles, Unmanned Aerial Systems (UAS), or small satellites. SHM for these systems must be able to handle unexpected situations and adapt specifications quickly during flight testing between closely-timed consecutive missions, not mid-mission, necessitating fast reconfiguration. They must enable more advanced probabilistic reasoning for diagnostics and prognostics while running aboard limited hardware without affecting the certified on-board software. We define and prove correct translations of two real-time projections of Linear Temporal Logic to two types of efficient observer algorithms to continuously assess the status of the system. A *synchronous* observer yields an instant abstraction of the satisfaction check, whereas an *asynchronous* observer concretizes this abstraction at a later, a priori known, time. By feeding the system's real-time status into a statistical reasoning unit, e.g., based on Bayesian networks, we enable advanced health estimation and diagnosis. We experimentally demonstrate our novel framework on real flight data from NASA's Swift UAS. By on-boarding rt-R2U2 aboard an existing FPGA already built into the standard UAS design and seamlessly intercepting sensor values through read-only observations of the system bus, we avoid system integration problems of software instrumentation or added hardware. The flexibility of our approach with regard to changes in the monitored specification is not due to the reconfigurability offered by FPGAs; it is a benefit of the modularity of our observers and would also be available on non-reconfigurable hardware platforms such as ASICs.

1 Introduction

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) [48] 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. [48, 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.

* A full version with appendices containing full proofs of correctness for all observer algorithms is available at http://research.kristinrozier.com/TACAS14.html. This work was supported in part by the Austrian Research Agency FFG, grant 825891, and NASA grant NNX08AY50A.

We need a new SHM framework for real-time systems like the Swift [47] electric UAS (see Fig. 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 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.

1.1 Related Work

Existing methods for Runtime Verification (RV) [35] 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 [46]; 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 [56]. 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 [38]. Mainly due to promising complexity results [37, 42], 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 [52], flight rules require future-time reasoning. To enable more intuitive specifications, others have studied monitoring of future-time claims; see [55] for a survey and [36, 42, 45, 54, 60, 61] 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. [43], however all either resynthesize monitors from scratch or exclude checking real-time properties [33]. 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. 4.1 only interconnects these blocks. Others have proposed using Bayesian inference techniques [41] 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.

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) [41] 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) [32] or a specialization of LTL for mission-time bounded characteristics, which we define in Sec. 2. To ensure RESPONSIVENESS of our rt-R2U2, we design two kinds of observer algorithms in Sec. 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. 4) and return an instant, three-valued abstraction $\{\textbf{true}, \textbf{false}, \textbf{maybe}\}$ 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 [59].

We implement the rt-R2U2 in hardware as a self-contained unit, which runs externally to the system, to support UNOBTRUSIVENESS; see Sec. 4. Safety-critical embedded systems often use industrial, vehicle bus systems, such as CAN and PCI, interconnecting hardware and software components, see Fig 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 [44, 53, 57] or use powerful database systems [34], 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

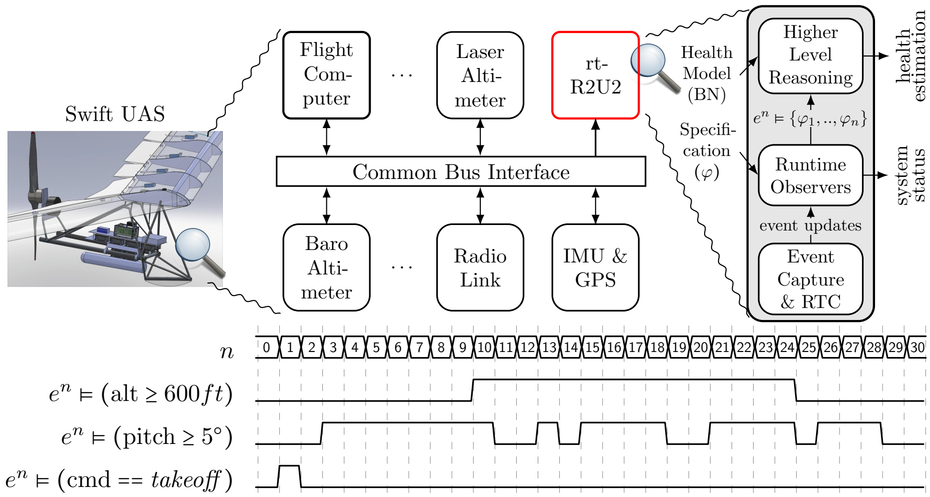


Fig. 1. rt-R2U2: An instance of our SHM framework rt-R2U2 for the NASA Swift UAS. Swift subsystems (top): The laser altimeter maps terrain and determines elevation above ground by measuring the time for a laser pulse to echo back to the UAS. The barometric altimeter determines altitude above sea level 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.

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 paper, we compute these health estimations by adapting the BN-based inference algorithms of [41] 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. 5).

2 Real-time projections of LTL

MTL replaces the temporal operators of LTL with operators that respect time bounds [32].

Definition 1 (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 $\{i \in \mathbb{N}_0 \mid t \le i \le t'\}$. 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 $n \ge 0$ 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 $e^n \vDash \varphi$, inductively as follows:

$e^n \vDash true$ is **true**, $e^n \vDash \sigma$ iff σ holds in s_n, $e^n \vDash \neg\varphi$ iff $e^n \nvDash \varphi$,
$e^n \vDash \varphi \wedge \psi$ iff $e^n \vDash \varphi$ and $e^n \vDash \psi$, $e^n \vDash \mathcal{X}\,\varphi$ iff $e^{n+1} \vDash \varphi$,
$e^n \vDash \varphi\,\mathcal{U}_J\,\psi$ iff $\exists i(i \ge n) : (i - n \in J \wedge e^i \vDash \psi \wedge \forall j(n \le j < i) :\ e^j \vDash \varphi)$.

With the dualities $\Diamond_J\varphi \equiv$ **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 $\Diamond_J\varphi$: $◧_\tau\varphi \equiv \Box_{[0,\tau]}\ \varphi$ (*φ is an invariant within the next τ time units*) and $⬖_\tau\varphi \equiv \Diamond_{[0,\tau]}\varphi$ (*φ holds eventually within the next τ time units*). For example, the flight rule from Sec. 1, "After receiving the takeoff command reach an altitude of $600ft$ within five minutes," is efficiently captured in MTL by (cmd == *takeoff*) $\rightarrow ⬖_5$(alt $\ge 600ft$), assuming a time-base of one minute and the atomic propositions (alt $\ge 600ft$) and (cmd == *takeoff*) as in Fig. 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.

Definition 2 (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 $◧_\tau\varphi$, $⬖_\tau\varphi$, 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.

Definition 3 (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 $v \in \{$**true**, **false**, **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 $n \in [\langle T_\varphi^{i-1} \rangle.\tau_e + 1, \langle T_\varphi^i \rangle.\tau_e]$, $e^n \vDash \varphi$ in case $\langle T_\varphi^i \rangle.v$ is **true** and $e^n \nvDash \varphi$ in case $\langle T_\varphi^i \rangle.v$ is **false**. In case $\langle T_\varphi^i \rangle$ is **maybe**, neither $e^n \vDash \varphi$ nor $e^n \nvDash \varphi$ is defined.

In the remainder of this paper, we will frequently refer to execution sequences collected from the Swift UAS as shown in Fig. 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, $\langle T_{\text{pitch}\ge 5^\circ} \rangle$ = ((**false**, 0), (**false**, 1), (**false**, 2), (**true**, 3), ..., (**true**, 17), (**true**, 18)) describes $e^n \vDash$ (pitch $\ge 5^\circ$) sampled over $n \in [0, 18]$ and $\langle T_{\text{pitch}\ge 5^\circ} \rangle$ holds 19 elements.

3 Asynchronous and Synchronous Observers

The problem of monitoring a real-time specification has been studied extensively in the past; see [39, 55] for an overview. Solutions include: (a) translating the temporal formula into a finite-state automaton that accepts all the models of the specification [42,43,45,61], (b) restricting MTL to its *safety* fragment and waiting until the operators' time bounds have elapsed to decide the truth value afterwards [36, 54], and (c) restricting LTL to its past-time fragment [37, 42, 57]. 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 [43], 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 $\neg\,\varphi$, $\varphi \wedge \psi$, $\blacksquare_\tau\,\varphi$, $\square_J\,\varphi$, 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).

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 $T.v \in \{\textbf{true}, \textbf{false}\}$ and $T.\tau_e \in [0, n]$. Since verdicts are exact evaluations of a future-time specification φ for each clock tick they may resolve φ for clock ticks prior to the current time n if the information required for this resolution was not available until n.

[1] Similar terms have been used by others [40] 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. 1).

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

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 $n = \langle T^i_\varphi \rangle.\tau_e + 1$ iff $(\langle T^i_\varphi \rangle.v \oplus \langle T^{i+1}_\varphi \rangle.v) \wedge \langle T^{i+1}_\varphi \rangle.v$ holds. Similarly, we say transition ‾|_ of execution sequence $\langle T_\varphi \rangle$ occurs at time $n = \langle T^i_\varphi \rangle.\tau_e + 1$ iff $(\langle T^i_\varphi \rangle.v \oplus \langle T^{i+1}_\varphi \rangle.v) \wedge \langle T^{i}_\varphi \rangle.v$ holds ($\oplus$ denotes the Boolean exclusive-or). For example, transitions _|‾ and ‾|_ of $\langle T_{\text{pitch}\geq 5^\circ} \rangle$ in Fig. 1 occur at times 3 and 11, respectively.

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 $(\ldots, (\textbf{true}, 2), (\textbf{false}, 3), \ldots)$.

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. 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 (␣, ␣) 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.

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. 1. For $n \in [0, 4]$ the algorithm returns (␣, ␣), since $(\langle T^{0\ldots4}_{\text{alt}\geq 600ft} \rangle.\tau_e - 5) \geq 0$ (line 3 of Alg. 9) does not hold. At $n = 5$ the underlying observer for ◧$_5$ (alt $\geq 600ft$) returns $(\textbf{false}, 5)$, which is transformed (by line 4) into the output $(\textbf{false}, 0)$. For similar arguments, the outputs for $n \in [6, 9]$ are $(\textbf{false}, 1)$, $(\textbf{false}, 2)$, $(\textbf{false}, 3)$, and $(\textbf{false}, 4)$. At $n \in [10, 14]$, the observer for ◧$_5$ (alt $\geq 600ft$) returns (␣, ␣). At $n = 15$, ◧$_5$ (alt $\geq 600ft$) yields $(\textbf{true}, 10)$, which is transformed (by line 4) into the output is $(\textbf{true}, 5)$. 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 $\neg\,\varphi$.

At each new input T_φ:
$T_\xi \leftarrow (\neg\, T_\varphi.v, T_\varphi.\tau_e)$
return T_ξ

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

At each new input T_φ:
$T_\xi \leftarrow T_\varphi$
if _/‾ transition of T_ξ occurs **then**
  $m_{\uparrow\varphi} \leftarrow m_{\tau_s}$
end if
$m_{\tau_s} \leftarrow T_\varphi.\tau_e + 1$
if T_ξ holds **then**
  if $m_{\uparrow\varphi} \leq (T_\xi.\tau_e - \tau)$ holds **then**
    $T_\xi.\tau_e \leftarrow T_\xi.\tau_e - \tau$
  else
    $T_\xi \leftarrow (\llcorner\!\lrcorner, \llcorner\!\lrcorner)$
  end if
end if
return T_ξ

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

At each new input (T_φ, T_ψ):
if T_φ holds and T_ψ holds and $q_\varphi \neq ()$ holds and $q_\psi \neq ()$ holds **then**
  $T_\xi \leftarrow (\textbf{true}, \min(T_\varphi.\tau_e, T_\psi.\tau_e))$
else if $\neg T_\varphi$ holds and $\neg T_\psi$ holds and $q_\varphi \neq ()$ holds and $q_\psi \neq ()$ holds **then**
  $T_\xi \leftarrow (\textbf{false}, \max(T_\varphi.\tau_e, T_\psi.\tau_e))$
else if $\neg T_\varphi$ holds and $q_\varphi \neq ()$ holds **then**
  $T_\xi \leftarrow (\textbf{false}, T_\varphi.\tau_e)$
else if $\neg T_\psi$ holds and $q_\psi \neq ()$ holds **then**
  $T_\xi \leftarrow (\textbf{false}, T_\psi.\tau_e)$
else
  $T_\xi \leftarrow (\llcorner\!\lrcorner, \llcorner\!\lrcorner)$
end if
dequeue$(q_\varphi, q_\psi, T_\xi.\tau_e)$
return T_ξ

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

At each new input T_φ:
$T_\xi \leftarrow$ ◧$_{\texttt{dur}(J)}\,T_\varphi$
if $(T_\xi.\tau_e - \min(J) \geq 0)$ **then**
  $T_\xi.\tau_e \leftarrow T_\xi.\tau_e - \min(J)$
else
  $T_\xi \leftarrow (\llcorner\!\lrcorner, \llcorner\!\lrcorner)$
end if
return T_ξ

Algorithm 5 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_φ, T_ψ) in lockstep mode:
if _/‾ transition of T_φ occurs **then**
  $m_{\uparrow\varphi} \leftarrow \tau_e - 1$
  $m_{pre} \leftarrow -\infty$
end if
if ‾_ transition of T_φ occurs and T_ψ holds **then**
  $T_\varphi.v$, $p \leftarrow$ **true**, **true**
  $m_{\downarrow\varphi} \leftarrow \tau_e$
end if
if T_φ holds **then**
  if T_ψ holds **then**
    if $(m_{\uparrow\varphi} + \min(J) < \tau_e)$ holds **then**
      $m_{pre} \leftarrow \tau_e$
      return $(\textbf{true}, \tau_e - \min(J))$
    else if p holds **then**
      return $(\textbf{false}, m_{\downarrow\varphi})$
    end if
  else if $(m_{pre} + \texttt{dur}(J) \leq \tau_e)$ holds **then**
    return $(\textbf{false}, \max(m_{\uparrow\varphi}, \tau_e - \max(J)))$
  end if
else
  $p \leftarrow$ **false**
  if $(\min(J) = 0)$ holds **then**
    return $(T_\psi.v, \tau_e)$
  end if
  return $(\textbf{false}, \tau_e)$
end if
return $(\llcorner\!\lrcorner, \llcorner\!\lrcorner)$

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 ◧$_5$ $(\text{alt} \geq 600ft) \wedge (\text{pitch} \geq 5°)$ and the execution in Fig. 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 $\blacksquare_5$ $(\text{alt} \geq 600ft)$ returns $(\text{\textvisiblespace}, \text{\textvisiblespace})$, while in the meantime, the atomic proposition $(\text{pitch} \geq 5°)$ 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 $\blacksquare_5$ $(\text{alt} \geq 600ft)$ 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 $\langle T_\varphi^i \rangle, \langle T_\varphi^{i+1} \rangle$ in $\langle T_\varphi \rangle$ by $\langle T_\varphi^{i+1} \rangle$ iff $\langle T_\varphi^i \rangle.v = \langle T_\varphi^{i+1} \rangle.v$, 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 $q_{\blacksquare_5 \, (\text{alt}\geq 600ft)} = ((\textbf{true}, 10))$. 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 $q_{\blacksquare_5 \, (\text{alt}\geq 600ft)} = ()$.

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 $\uparrow$ transition ($\downarrow$ 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 $(\text{pitch} \geq 5°) \, \mathcal{U}_{[5,10]} \, (\text{alt} \geq 600ft)$ over the execution in Fig. 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 $\uparrow$ 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 $\uparrow$ 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 $e^n \vDash (\text{pitch} \geq 5°) \, \mathcal{U}_{[5,10]} \, (\text{alt} \geq 600ft)$ for $n \in [3,5]$. At time $n = 11$, a $\downarrow$ transition of $\langle T_{\text{pitch}\geq 5°} \rangle$ occurs and since $\langle T^{11}_{\text{alt}\geq 600ft} \rangle$ holds, p and the truth value of the current input $\langle T^{11}_{\text{pitch}\geq 5°} \rangle.v$ 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 $\langle T^{12}_{\text{pitch}\geq 5°} \rangle$ does not hold, we clear p in line 22 and the algorithm returns $(\textbf{false}, 12)$ in line 26, i.e., $e^n \nvDash (\text{pitch} \geq 5°) \, \mathcal{U}_{[5,10]} \, (\text{alt} \geq 600ft)$ for $n \in [7,12]$. At time $n = 13$, a $\uparrow$ transition of $\langle T_{\text{pitch}\geq 5°} \rangle$ 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 $\downarrow$ transition of $\langle T_{\text{pitch}\geq 5°} \rangle$ occurs, thus p and $\langle T^{14}_{\text{pitch}\geq 5°} \rangle.v$ are set **true** and

$m_{\downarrow\varphi} = 14$. The predicate in line 15 holds, and the algorithm outputs $(\textbf{false}, 14)$, revealing that $e^n \not\models (\text{pitch} \geq 5°)\, \mathcal{U}_{[5,10]}\, (\text{alt} \geq 600ft)$ for $n \in [13, 14]$.

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 $\widehat{\textbf{eval}} : \circledast \to \{\textbf{true}, \textbf{false}, \textbf{maybe}\}$, where $\circledast \in \{\neg\varphi, \varphi \wedge \psi, \blacksquare_\tau \varphi, \Box_J \varphi, \varphi\, \mathcal{U}_J\, \psi\}$. The implementation of $\widehat{\textbf{eval}}(\neg\varphi)$ and $\widehat{\textbf{eval}}(\varphi \wedge \psi)$ follows the rules for Kleene logic [49]. 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:

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

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

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

To illustrate our synchronous observer algorithms, consider the previously discussed formula $\blacksquare_5\, (\text{alt} \geq 600ft) \wedge (\text{pitch} \geq 5°)$, which we want to evaluate using the synchronous observer:

$$\xi = \widehat{\textbf{eval}}(\widehat{\textbf{eval}}(\blacksquare_5\, (\text{alt} \geq 600ft)) \wedge (\text{pitch} \geq 5°))$$

For $n \in [0, 9]$, as in the case of the *asynchronous* observer, we can immediately output $(\textbf{false}, n)$. At $n = 10$, $\widehat{\textbf{eval}}(\blacksquare_5\, (\text{alt} \geq 600ft))$ yields $(\textbf{maybe}, n)$, thus, the observer is inconclusive about the truth value of $e^{10} \models \xi$. At $n \in [11, 12]$ since $(\text{pitch} \geq 5°)$ does not hold, the outputs are $(\textbf{false}, n)$. For analogous arguments, the output at $n = 13$ is $(\textbf{maybe}, 13)$, at $n = 14$ $(\textbf{false}, 14)$, and at $n = 15$ $(\textbf{maybe}, 15)$. 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 $n \in \{16, 17, 19\}$.

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 $\boxminus_\tau \varphi$ (Fig. 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 $\_\!\ulcorner$ 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 $(\text{␣}, \text{␣})$.

Synchronous observers do not require calculations on time stamps and directly map to basic digital logic gates. Fig. 2 shows a circuit representing an $\widehat{\textbf{eval}}\,(\boxminus_\tau \varphi)$ 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: **true** $(0, 0)$, **false** $(0, 1)$, and **maybe** $(1, 0)$. Input j is set if $\tau_e = 0$ and cleared otherwise.

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 $\widehat{\textbf{eval}}\,(\xi)$, works as follows:

- Preprocessing. By the equivalences given in Sect. 2 rewrite ξ to ξ', such that operators in ξ' are from $\{\neg\,\varphi, \varphi \wedge \psi, \boxminus_\tau \varphi, \Box_J\, \varphi, \varphi\, \mathcal{U}_J\, \psi\}$ (**SA1**).
- Parsing. Parse ξ' to obtain an Abstract Syntax Tree (AST), denoted by AST(ξ'). The leaves of this tree are the atomic propositions Σ of ξ' (**SA2**).
- Allocating observers. For all nodes q in AST(ξ') allocate both the corresponding synchronous and the asynchronous hardware observer block (**SA3**).
- Adding synchronization queues. $\forall q \in$ AST(ξ'): 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 AST(ξ'). Execute Alg. 6 (**MA2**).

Let $\{\sigma_1, \sigma_2, \sigma_3\} \in \Sigma$ and $\xi = \sigma_1 \rightarrow (\diamondminus_{10}\, (\sigma_2) \vee \diamondminus_{100}(\sigma_3))$ be an MTL formula we want to synthesize a configuration for. SA1 yields $\xi' = \neg(\sigma_1 \wedge \neg(\neg\boxminus_{10}\, (\neg\sigma_2)) \wedge \neg(\neg\boxminus_{100}\, (\neg\sigma_3)))$ which simplifies to $\xi = \neg(\sigma_1 \wedge \boxminus_{10}\, (\neg\sigma_2) \wedge \boxminus_{100}\, (\neg\sigma_3))$. SA2 yields AST($\xi'$). SA3 instantiates two $\varphi \wedge \psi$, three $\neg\,\varphi$, one $\boxminus_{10}\, T_\varphi$ and one $\boxminus_{100}\, T_\varphi$ 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.

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.

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

while S is not empty **do**
    $s, w \leftarrow$ get next node from S, 0
    if s is type $\varphi\,\mathcal{U}_J\,\psi$ or $\varphi \wedge \psi$ **then**
        $w \leftarrow \max(|q_\varphi|, |q_\psi|) + \mathrm{wcd}(s)$
    end if
    while s is not a synchronization queue **do**
        $s, w \leftarrow$ get predecessor of s in AST(ξ'), $w + \mathrm{wcd}(s)$
    end while
    Set $|q| = w$; (q is opposite synchronization queue of s)
    Add all $\varphi\,\mathcal{U}_J\,\psi$ and $\varphi \wedge \psi$ nodes that have unassigned synchronization queue sizes to S
end while

Theorem 1 (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* AST(φ)*, and $n \in \mathbb{N}_0$ is the time stamp it is executed.*

Theorem 2 (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* AST(φ)*, d the depth of AST(φ), 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.

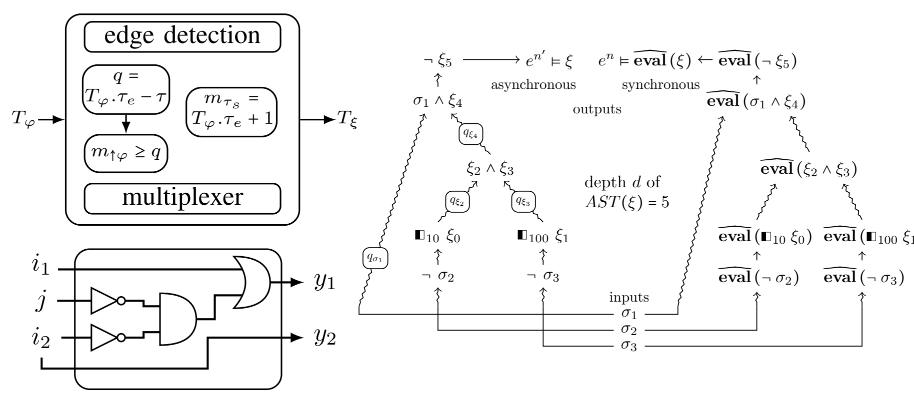


Fig. 2. Left: hardware implementations for ◧$_\tau\,\varphi$ (top) and $\widehat{\textbf{eval}}$ (◧$_\tau\,\varphi$) (bottom). Right: subformulas of AST(ξ), observers, and queues synthesized for ξ. Mapping the observers to hardware yields two levels of parallelism: (i) asynchronous (left) and the synchronous observers (right) run in parallel and (ii) observers for subformulas run in parallel, e.g., ◧$_{10}\,\xi_0$ and ◧$_{100}\,\xi_1$.

Theorem 3 (Circuit-Size Complexity of Synchronous Observers). *For a given MTL formula φ, the circuit to monitor $\widehat{\textbf{eval}}(\varphi)$ has a circuit-size complexity bounded by $11 \cdot m$, where m is the number of observers in AST(φ).*

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

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 reasoning is performed by a *health model*, modeled as a Bayesian network (BN) where the nodes correspond to discrete random variables. Fig. 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 [41]; for details of modeling and reasoning see [58].

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

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

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. 3 shows the execution sequences produced by both the asynchronous ($e^n \vDash \varphi_1$) and the synchronous ($e^n \vDash \widehat{\textbf{eval}}(\varphi_1)$) 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 $(\textbf{maybe}, 1)$, indicated by ⩫. The asynchronous observer will resolve this inconclusive output at time $n = 11$, by generating the tuple $(\textbf{false}, 1)$, 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:

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

$\sigma_{S_{B\uparrow}}$ 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,

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

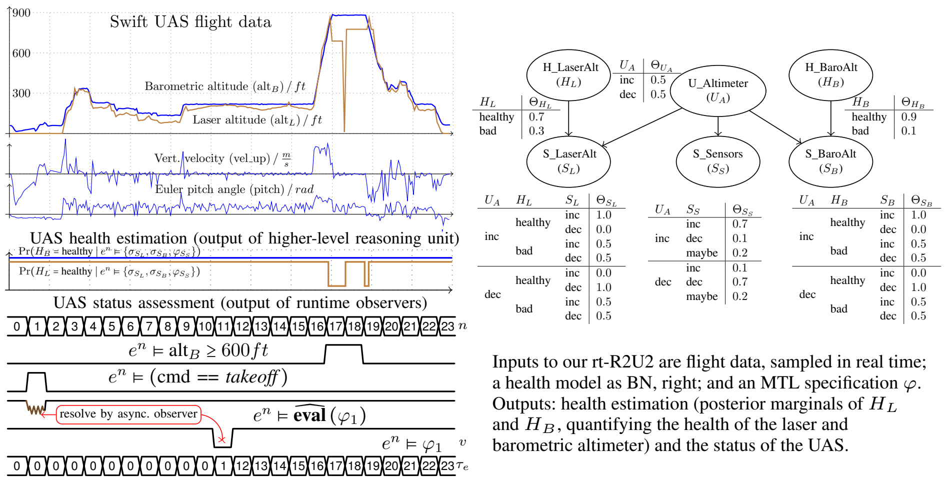


Fig. 3. Adding SHM to the Swift UAS

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 $\blacksquare_\tau \varphi$ observers as filters, by requiring that the pitch and the velocity signals exceed a threshold for multiple time steps.

$$\varphi_{S_{S\uparrow}} = \blacksquare_{10} \left(\text{pitch} \geq 5^\circ\right) \wedge \blacksquare_5 \left(\text{vel\_up} \geq 2\frac{m}{s}\right)$$
$$\varphi_{S_{S\downarrow}} = \blacksquare_{10} \left(\text{pitch} < 2^\circ\right) \wedge \blacksquare_5 \left(\text{vel\_up} \leq -2\frac{m}{s}\right)$$

Our real-time SHM analysis matched post-flight analysis by test engineers, including successfully pinpointing a laser altimeter failure, see Fig 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.

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.

References

1. Alur, R., Henzinger, T.A.: Real-time Logics: Complexity and Expressiveness. In: LICS. pp. 390–401. IEEE (1990)
2. 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)
3. Barre, B., Klein, M., Soucy-Boivin, M., Ollivier, P.A., Hallé, S.: MapReduce for parallel trace validation of LTL properties. In: RV. LNCS, vol. 7687, pp. 184–198. Springer (2012)
4. Barringer, H., Falcone, Y., Finkbeiner, B., Havelund, K., Lee, I., Pace, G.J., Rosu, G., Sokolsky, O., Tillmann, N. (eds.): Runtime Verification (RV), LNCS, vol. 6418. Springer (2010)
5. Basin, D., Klaedtke, F., Müller, S., Pfitzmann, B.: Runtime monitoring of metric first-order temporal properties. In: FSTTCS. pp. 49–60 (2008)
6. Basin, D., Klaedtke, F., Zălinescu, E.: Algorithms for monitoring real-time properties. In: RV. pp. 260–275. LNCS, Springer (2011)
7. Bauer, A., Leucker, M., Schallhart, C.: Comparing LTL semantics for runtime verification. J. Log. and Comp. 20, 651–674 (2010)
8. Bauer, A., Leucker, M., Schallhart, C.: Runtime verification for LTL and TLTL. ACM Trans. Softw. Eng. M. 20, 14:1–14:64 (2011)
9. Colombo, C., Pace, G., Abela, P.: Safer asynchronous runtime monitoring using compensations. FMSD 41, 269–294 (2012)
10. Darwiche, A.: Modeling and Reasoning with Bayesian Networks. Cambridge University Press, New York, NY, USA, 1st edn. (2009)
11. Divakaran, S., D'Souza, D., Mohan, M.R.: Conflict-tolerant real-time specifications in metric temporal logic. In: TIME. pp. 35–42 (2010)
12. Finkbeiner, B., Kuhtz, L.: Monitor circuits for LTL with bounded and unbounded future. In: RV, LNCS, vol. 5779, pp. 60–75. Springer (2009)
13. Fischmeister, S., Lam, P.: Time-aware instrumentation of embedded software. IEEE Trans. Ind. Informatics 6(4), 652–663 (2010)
14. Geilen, M.: An improved on-the-fly tableau construction for a real-time temporal logic. In: CAV. pp. 394–406 (2003)
15. Havelund, K.: Runtime verification of C programs. In: TestCom/FATES. LNCS, vol. 5047, pp. 7–22. Springer (2008)
16. 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)
17. 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)
18. Kleene, S.C.: Introduction to Metamathematics. North Holland (1996)
19. Kogge, P.M., Stone, H.S.: A parallel algorithm for the efficient solution of a general class of recurrence equations. IEEE Trans. Comput. 22(8), 786–793 (Aug 1973)
20. Kroening, D., Strichman, O.: Decision Procedures: An Algorithmic Point of View. Springer (2008)
21. Lichtenstein, O., Pnueli, A., Zuck, L.: The glory of the past. In: Logics of Programs, LNCS, vol. 193, pp. 196–218. Springer (1985)
22. 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 (2007)
23. Maler, O., Nickovic, D., Pnueli, A.: On synthesizing controllers from bounded-response properties. In: CAV. LNCS, vol. 4590, pp. 95–107. Springer (2007)
24. Maler, O., Nickovic, D., Pnueli, A.: Checking temporal properties of discrete, timed and continuous behaviors. In: Pillars of Comp. Science. pp. 475–505. Springer (2008)
25. Pike, L., Niller, S., Wegmann, N.: Runtime verification for ultra-critical systems. In: RV. pp. 310–324. LNCS, Springer (2011)
26. Reinbacher, T., Függer, M., Brauer, J.: Real-time runtime verification on chip. In: RV. LNCS, vol. 7687, pp. 110–125. Springer (2012)
27. Schumann, J., Mbaya, T., Mengshoel, O., Pipatsrisawat, K., Srivastava, A., Choi, A., Darwiche, A.: Software health management with Bayesian Networks. Innovations in Systems and SW Engineering 9(4), 271–292 (2013)

28. 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: PHM (2013)
29. Tabakov, D., Rozier, K.Y., Vardi, M.Y.: Optimized temporal monitors for SystemC. Formal Methods in System Design 41(3), 236–268 (2012)
30. Thati, P., Roşu, G.: Monitoring Algorithms for Metric Temporal Logic specifications. ENTCS 113, 145–162 (2005)
31. Wegener, I.: The Complexity of Boolean Functions. Wiley-Teubner Series in Computer Science, John Wiley & Sons (1987), isbn: 3519021072

7 Appendix A – Proofs of Correctness

Theorem 5 (Correctness of the Observer for $\neg\,\varphi$). *For any execution sequence $\langle T_\varphi \rangle$, the observer stated in Algorithm 7 implements $e^n \vDash \neg\,\varphi$.*

Algorithm 7 Observer for $\neg\,\varphi$.

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

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

Theorem 6 **(Correctness of the Observer for $\text{◧}_\tau\, \varphi$).** *For any execution sequence $\langle T_\varphi \rangle$, the observer stated in Algorithm 8 implements $e^n \vDash \text{◧}_\tau\, \varphi$.*

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

1: At each new input T_φ:
2: $T_\xi \leftarrow T_\varphi$
3: **if** _|‾ transition of T_ξ 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_ξ 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:         $T_\xi \leftarrow (\llcorner\!\lrcorner, \llcorner\!\lrcorner)$
12:     **end if**
13: **end if**
14: **return** T_ξ

Proof. We first observe the equivalences

$$
\begin{aligned}
e^n \vDash \text{◧}_\tau\, \varphi &\Leftrightarrow e^n \vDash \square_{[0,\tau]}\, \varphi,\\
&\Leftrightarrow e^n \vDash \neg(\textbf{true}\,\mathcal{U}_{[0,\tau]}\, \neg\varphi),\\
&\Leftrightarrow \neg(\exists i(i \geq n) : (i - n \in [0,\tau] \land e^i \vDash \neg\varphi \land \forall j(n \leq j < i) :\ e^j \vDash \textbf{true})),\\
&\Leftrightarrow \neg(\exists i(i \geq n) : (i - n \in [0,\tau] \land e^i \vDash \neg\varphi \land \textbf{true})),\\
&\Leftrightarrow \neg(\exists i(i \geq n) : (i - n \in [0,\tau] \land e^i \vDash \neg\varphi)),\\
&\Leftrightarrow \forall i(i \geq n) : \neg(i - n \in [0,\tau] \land e^i \vDash \neg\varphi),\\
&\Leftrightarrow \forall i(i \geq n) : (\neg(i - n \in [0,\tau]) \lor e^i \vDash \varphi),\\
&\Leftrightarrow \forall i(i \geq n) : (i - n \in [0,\tau] \rightarrow e^i \vDash \varphi),\\
&\Leftrightarrow \forall i : (i \in [n, n+\tau] \rightarrow e^i \vDash \varphi).
\end{aligned}
$$

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

The theorem follows.

Theorem 7 (Correctness of the Observer for $\square_J\,\varphi$). *For any execution sequence $\langle T_\varphi \rangle$, the observer stated in Algorithm 9 implements $e^n \models \square_J\,\varphi$.*

Algorithm 9 Observer for $\square_J\,\varphi$.

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

Proof. We first observe the equivalences

$$
\begin{aligned}
& e^n \models \square_J\,\varphi \\
& \quad \Leftrightarrow e^n \models \neg\,(\mathbf{true}\,\mathcal{U}_{[\min(J),\max(J)]}\,\neg\,\varphi), \\
& \quad \Leftrightarrow \forall i (i \geq n) : (i - n \in [\min(J), \max(J)] \rightarrow e^i \models \varphi), \\
& \quad \Leftrightarrow \forall i : (i \in [n + \min(J), n + \max(J)] \rightarrow e^i \models \varphi).
\end{aligned} \tag{1}
$$

By Theorem 6 we have

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

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

$$
\begin{aligned}
& e^n \models \text{◧}_{\mathtt{dur}(J)}\,\varphi, \\
& \quad \Leftrightarrow \forall i : (i \in [n, n + \mathtt{dur}(J)] \rightarrow e^i \models \varphi), \\
& \quad \Leftrightarrow \forall i : (i \in [n, n + \max(J) - \min(J)] \rightarrow e^i \models \varphi).
\end{aligned} \tag{2}
$$

By the Equivalences 1 and 2 we observe that both $e^n \models \square_J\,\varphi$ and $e^n \models \text{◧}_\tau\,\varphi$ require that φ holds for an interval of length $\max(J) - \min(J) = \mathtt{dur}(J)$, however, $e^n \models \square_J\,\varphi$ requires that φ holds for an interval that is $\min(J)$ ahead (i.e., in the future) of $e^n \models \text{◧}_\tau\,\varphi$. Subtracting $\min(J)$ (equals to a shift into the past by $\min(J)$ time stamps) from

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

yields

$$
\begin{aligned}
& \forall i : (i - \min(J) \in [n, n + \max(J) - \min(J)] \rightarrow e^i \models \varphi), \\
& \quad \Leftrightarrow \forall i : (i \in [n + \min(J), n + \max(J) - \min(J) + \min(J)] \rightarrow e^i \models \varphi), \\
& \quad \Leftrightarrow \forall i : (i \in [n + \min(J), n + \max(J)] \rightarrow e^i \models \varphi), \\
& \quad \Leftrightarrow \square_J\,\varphi \qquad \text{(cf. Equation 1).}
\end{aligned}
$$

Since Algorithm 9 instantiates a $\text{◧}_\tau\,\varphi$ 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 8 (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 $e^n \vDash \varphi \wedge \psi$.*

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

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

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 [49]. 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 $T_\varphi.v, T_\psi.v \in \{\textbf{true}, \textbf{false}\}$. Depending on the state of the synchronization queues, we distinguish the following cases:

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 $T_\varphi.v = T_\psi.v =$ **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

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

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

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 1 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

	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)	$(\textbf{false}, \max(T_\varphi.\tau_e, T_\psi.\tau_e))$	5
2	0	1			0	time stamp of φ	$(\textbf{false}, T_\varphi.\tau_e))$	7
3	1	0			0	time stamp of ψ	$(\textbf{false}, T_\psi.\tau_e))$	9
4	1	1			1	min(time stamp φ, time stamp ψ)	$(\textbf{true}, \min(T_\varphi.\tau_e, T_\psi.\tau_e))$	3
5	0	0	0	1	0	time stamp of φ	$(\textbf{false}, T_\varphi.\tau_e))$	7
6	0	1			0	time stamp of φ	$(\textbf{false}, T_\varphi.\tau_e))$	7
7	1	0			?	-	$(\textvisiblespace, \textvisiblespace)$	11
8	1	1			?	-	$(\textvisiblespace, \textvisiblespace)$	11
9	0	0	1	0	0	time stamp of ψ	$(\textbf{false}, T_\psi.\tau_e))$	9
10	0	1			?	-	$(\textvisiblespace, \textvisiblespace)$	11
11	1	0			0	time stamp of ψ	$(\textbf{false}, T_\psi.\tau_e))$	9
12	1	1			?	-	$(\textvisiblespace, \textvisiblespace)$	11
13	0	0	1	1	?	-	$(\textvisiblespace, \textvisiblespace)$	11
14	0	1			?	-	$(\textvisiblespace, \textvisiblespace)$	11
15	1	0			?	-	$(\textvisiblespace, \textvisiblespace)$	11
16	1	1			?	-	$(\textvisiblespace, \textvisiblespace)$	11

Table 1. 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 $q_\psi = ()$.

non-empty (see #2 in Table 1), the expected output is $a \wedge b$ = **true** $\wedge$ **false** = **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 1), the expected output is $a \wedge b$ = **maybe** $\wedge$ **true** = **maybe**.

It remains to be shown that Algorithm 10 generates these outputs. We study the column "Outputs of Algorithm 10" of Table 1, 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 1), 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 1. 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 1.
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 1. In the special case that both $T_\varphi.v$ and $T_\psi.v$ are **false** (#1 in Table 1), 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 1.

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 **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 1 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 1), 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.

It remains to be shown that Algorithm 10 generates these time stamps. Again, we study the column "Outputs of Algorithm 10" of Table 1, 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 1), 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.

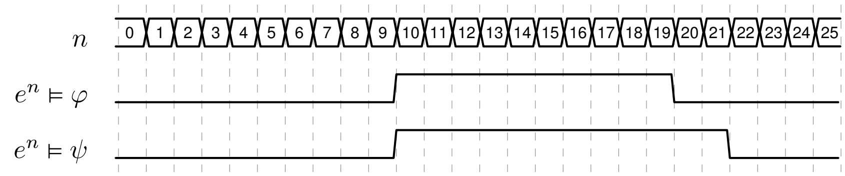
Theorem 9 (Correctness of the Observer for $\varphi \mathcal{U}_J \psi$). *For any two execution sequences $\langle T_\varphi \rangle$, $\langle T_\psi \rangle$, the observer stated in Algorithm 12 implements $e^n \models \varphi \mathcal{U}_J \psi$.*

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_φ, T_ψ) in lockstep mode:
if _|‾ transition of T_φ occurs **then**
    $m_{\uparrow\varphi} \leftarrow \tau_e - 1$
    $m_{pre} \leftarrow -\infty$
end if
if ‾|_ transition of T_φ occurs and T_ψ holds **then**
    $T_\varphi.v$, $p \leftarrow$ **true**, **true**
    $m_{\downarrow\varphi} \leftarrow \tau_e$
end if
if T_φ holds **then**
    if T_ψ holds **then**
        if $(m_{\uparrow\varphi} + \min(J) < \tau_e)$ holds **then**
            $m_{pre} \leftarrow \tau_e$
            return (**true**, $\tau_e - \min(J)$)
        else if p holds **then**
            return (**false**, $m_{\downarrow\varphi}$)
        end if
    else if $(m_{pre} + \mathtt{dur}(J) \le \tau_e)$ holds **then**
        return (**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 (**false**, τ_e)
end if
return $(\llcorner\!\lrcorner, \llcorner\!\lrcorner)$

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 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[4]: $((\textbf{false}, 9), (\textbf{false}, 9))$, $((\textbf{true}, 19), (\textbf{true}, 19))$, $((\textbf{false}, 21), (\textbf{true}, 21))$,$((\textbf{false}, 25), (\textbf{false}, 25))$. The observer will produce the following outputs:

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

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 $(\textbf{false}, 21)$ 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 $((\textbf{false}, 21), (\textbf{true}, 21))$ into $((\textbf{false}, 20), (\textbf{true}, 20))$ (3.1) and $((\textbf{false}, 21), (\textbf{true}, 21))$ (3.2) and calculate:

3.1 For input $((\textbf{false}, 20), (\textbf{true}, 20))$, the observer in Algorithm 12 returns $(\textbf{true}, 18)$.
3.2 For input $((\textbf{false}, 21), (\textbf{true}, 21))$, the observer in Algorithm 12 returns $(\textbf{false}, 21)$.

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

If: The observer for $\varphi \mathcal{U}_J \psi$ returns $(\textbf{true}, n)$ 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 $(\textbf{true}, n)$.

If we show the correctness of both statements, Theorem 9 holds. We will start by proving the following lemma that helps us to simplify the proof of Theorem 9.

Lemma 1 (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* $n' \leq n + \max(J)$.

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

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

$$
\begin{aligned}
& e^n \models \varphi\,\mathcal{U}_J\,\psi \\
& \Leftrightarrow \exists i(i \geq n) : (i - n \in J \wedge e^i \models \psi \wedge \forall j(n \leq j < i) :\ e^j \models \varphi), \\
& \Leftrightarrow \exists i(i \geq n) : (i - n \in [\min(J), \max(J)] \wedge e^i \models \psi \wedge \forall j(n \leq j < i) :\ e^j \models \varphi), \\
& \Leftrightarrow \exists i(i \geq n) : (i \in [n + \min(J), n + \max(J)] \wedge e^i \models \psi \wedge \forall j(n \leq j < i) :\ e^j \models \varphi).
\end{aligned} \tag{3}
$$

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 3 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 3 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 3 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 $(\llcorner\!\lrcorner, \llcorner\!\lrcorner)$ on line 28.

Case (ii): $n + \min(J) \leq i \leq n + \max(J)$, since $i \in [n + \min(J), n + \max(J)]$, Equation 3 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 3 does not hold. The algorithm immediatelly returns $(\textbf{false}, \tau_e)$ in line 26 (since T_φ does not hold). For the latter, Equation 3 holds, and the algorithm returns $(\textbf{true}, \tau_e - \min(J))$ 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 $i \notin [n + \min(J), n + \max(J)]$, Equation 3 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 9. We first show that:

If: The observer for $\varphi\,\mathcal{U}_J\,\psi$ returns $(\textbf{true}, n)$ 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 $(\textbf{true}, n)$. We observe that Algorithm 12 may return tuples T where $T.v = \textbf{true}$ either in line 14 (case (i)) or in line 24 (case (ii)).

Case (i): the observer returns $(\textbf{true}, \tau_e - \min(J))$ 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

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

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

$$e^{\tau_e - \min(J)} \vDash \varphi \mathcal{U}_J \psi \Leftrightarrow (\min(J) \in [\min(J), \max(J)] \wedge e^{\tau_e} \vDash \psi \wedge \textbf{true}), \\ \Leftrightarrow \textbf{true} \wedge e^{\tau_e} \vDash \psi \wedge \textbf{true} \Leftrightarrow e^{\tau_e} \vDash \psi.$$

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

Case (ii): the observer returns $(\textbf{true}, n)$ 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 $e^n \vDash \varphi \mathcal{U}_J \psi$ (we substitute $\min(J) = 0$ and $i = \tau_e$):

$$\begin{aligned} &e^n \vDash \varphi \mathcal{U}_{[0,\max(J)]} \psi \\ &\Leftrightarrow \exists i (i \geq n) : (i - n \in J \wedge e^i \vDash \psi \wedge \forall j (n \leq j < i) : \ e^j \vDash \textbf{false}), \\ &\Leftrightarrow \exists i (i \geq n) : (\tau_e - n \in [0, \max(J)] \wedge e^{\tau_e} \vDash \psi \wedge \forall j (n \leq j < \tau_e) : \ e^j \vDash \textbf{false}), \end{aligned} \quad (4)$$

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 4 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 4 holds. We arrive at

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

Thus, in case $\min(J) = 0$ and $e^{\tau_e} \nvDash \varphi$, the truth value of $e^{\tau_e} = \varphi \mathcal{U}_J \psi$ is equal to $e^{\tau_e} \vDash \psi$. 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 $(\textbf{true}, n)$ if* $e^n \vDash \varphi \mathcal{U}_J \psi$ holds.

To complete the proof it remains to be shown that:

Only If: $e^n \vDash \varphi \mathcal{U}_J \psi$ holds if the observer for $\varphi \mathcal{U}_J \psi$ returns $(\textbf{true}, n)$. 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 $e^0 \nvDash \varphi$ and $e^0 \nvDash \psi$. By substituting into the definition of $e^n \vDash \varphi \mathcal{U}_J \psi$ we get

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

By our assumption $e^0 \nvDash \varphi$, $\forall j (0 \leq j < i) : \ e^j \vDash \varphi$ 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) $e^0 \nvDash \varphi \mathcal{U}_J \psi$ since $0 \in [0, \max(J)]$ holds, however, $e^0 \nvDash \psi$. We observe that the algorithm returns $(\textbf{false}, 0)$ for this case in line 24.
(b) $e^0 \nvDash \varphi \mathcal{U}_J \psi$ 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 (ii): assume T_φ does not hold and T_ψ holds. We have that $e^0 \nvDash \varphi$ and $e^0 \vDash \psi$. For analogous arguments as in case (i), we distinguish the two cases (a) $\min(J) = 0$ and (b) $\min(J) > 0$.

(a) $e^0 \vDash \varphi \mathcal{U}_J \psi$ since $0 \in [0, \max(J)]$ holds and $e^0 \nvDash \psi$. We observe that the algorithm returns $(\textbf{true}, 0)$ for this case in line 24.
(b) $e^0 \nvDash \varphi \mathcal{U}_J \psi$ 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 $e^0 \vDash \varphi$ and $e^0 \nvDash \psi$. 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) $e^0 \nvDash \varphi \mathcal{U}_J \psi$ since with $i \in [0, 0]$ we have $e^0 \nvDash \psi$. 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 $e^0 \vDash \varphi$ and $\min(J) > 0$, the validity of $e^0 \vDash \varphi \mathcal{U}_J \psi$ cannot be determined at time $n = 0$ as we can choose an arbitrary $i \in [\min(J), \max(J)]$ and need to test $e^i \vDash \psi$ and $\forall j (0 \le j < i):\ e^j \vDash \varphi$ at times $i > n$. The induction base follows by Lemma 1.

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 $e^0 \vDash \varphi$ and $e^0 \vDash \psi$. As in cases (i) and (ii), we distinguish the two cases (a) $\min(J) = 0$ and (b) $\min(J) > 0$.

(a) $e^0 \vDash \varphi \mathcal{U}_J \psi$ since with $i \in [0, \max(J)]$ we choose $i = 0$ we have $e^0 \vDash \psi$. 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 $e^0 \vDash \varphi$ and $\min(J) > 0$, the validity of $e^0 \vDash \varphi \mathcal{U}_J \psi$ cannot be determined at time $n = 0$. For analogous arguments as in case (iii.b), the induction base follows by Lemma 1.

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

Induction Step $(n-1 \to n)$ with the induction hypothesis: assume that $e^{n-1} \vDash \varphi \mathcal{U}_J \psi$ if the observer for $\varphi \mathcal{U}_J \psi$ returns $(\textbf{true}, n-1)$ holds for $n - 1 \ge 0$. 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 = **true**, $m_{\downarrow\varphi} = n$, and $T_\varphi.v$ = **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 $e^n \vDash \varphi\,\mathcal{U}_J\,\psi$ 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 $e^n \nvDash \varphi\,\mathcal{U}_J\,\psi$ 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 $e^n \vDash \varphi\,\mathcal{U}_J\,\psi$ we have

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

and with the observation from above, $e^n \vDash \varphi\,\mathcal{U}_J\,\psi$ can only hold for an $i > \tau_e$. This implies that $e^n \vDash \varphi\,\mathcal{U}_J\,\psi$ 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 _/‾ 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, $e^n \vDash \varphi\,\mathcal{U}_J\,\psi$ cannot hold until a time stamp n that is equal to the time stamp of the _/‾ 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

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

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 $e^i \nvDash \psi$ and thus $e^n \nvDash \varphi\,\mathcal{U}_{[0,0]}\,\psi$. 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 $\uparrow$ 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 $\uparrow$ 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 $\uparrow$ transition of φ occurred at a time later than $n - \min(J)$. Then, the induction step follows by Lemma 1.

(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 $(\textbf{true}, n)$. 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 1.

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

The theorem follows.

8 Appendix B – Proofs of Complexity Results

Theorem 10 (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* AST(φ)*, 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 AST(φ); the weight on the edges is the value computed by **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 AST(φ) 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 11 (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 $\text{AST}(\varphi)$, d the depth of AST(φ), and $n \in \mathbb{N}_0$ the time stamp it is executed.*

Proof. As shown in [50] 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 $\text{Add}(\langle a \rangle, \langle b \rangle, c)$ 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 $\text{Add}(\langle a \rangle, \langle \overline{b} \rangle, 1)$, 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 [51, 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.

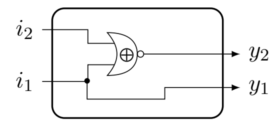
For a given MTL specification φ, we can determine its depth d by the number of nodes of the longest-path in the parse tree of φ. We then arrive at an asymptotic time complexity of $\mathcal{O}(d \cdot \log_2 \log_2 \max(p,n))$ for an asynchronous observer for φ.

The theorem follows.

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

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

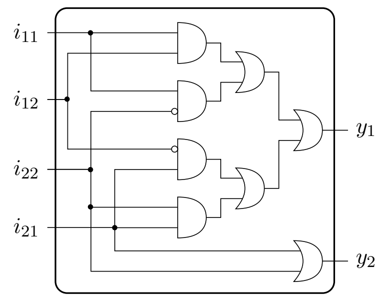
two-input gates: 1



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

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

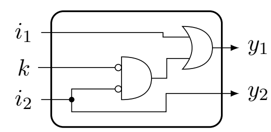
two-input gates: 8



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

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

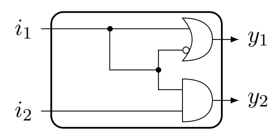
two-input gates: 2



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

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

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

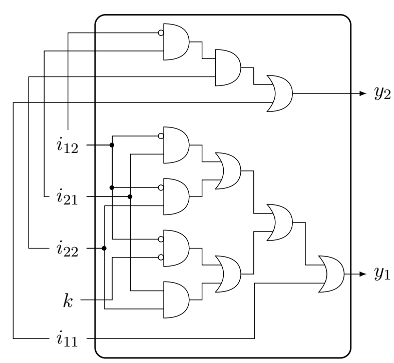


Fig. 4. Mapping of synchronous MTL observers to circuits of two-input gates.

Theorem 12 (Circuit-Size Complexity of Synchronous Observers). *For a given MTL formula φ, the circuit to monitor $\widehat{\textbf{eval}}\,(\varphi)$ 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 [62] bounded by $11 \cdot m$, where m is the number of observers in AST(φ). This statement holds, if we can show that any of the synchronous observers for $\widehat{\textbf{eval}}\,(\neg\,\varphi)$, $\widehat{\textbf{eval}}\,(\varphi \wedge \psi)$, $\widehat{\textbf{eval}}\,(◧_{\tau}\,\varphi)$, $\widehat{\textbf{eval}}\,(\square_J\,\varphi)$, and $\widehat{\textbf{eval}}\,(\varphi\,\mathcal{U}_J\,\psi)$ can be built with at most 11 two-input gates. The circuits in Figure 4 show that any of the synchronous observers can be built with at most 11 two-input gates.

The theorem follows.

Theorem 13 (Circuit-Depth Complexity of Synchronous Observers). *For a given MTL formula φ, the circuit to monitor* $\widehat{\textbf{eval}}\,(\varphi)$ *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 [62] 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 4 we can observe the depth of these circuits:

1. $\widehat{\textbf{eval}}\,(\neg\,\varphi)$ circuit depth: 1
2. $\widehat{\textbf{eval}}\,(\varphi\,\wedge\,\psi)$ circuit depth: 3
3. $\widehat{\textbf{eval}}\,(\blacksquare\!\!\square_{\tau}\,\varphi)$ circuit depth: 2
4. $\widehat{\textbf{eval}}\,(\square_{J}\,\varphi)$ circuit depth: 1
5. $\widehat{\textbf{eval}}\,(\varphi\,\mathcal{U}_{J}\,\psi)$ circuit depth: 4

The theorem follows.

9 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. 2 summarizes the relevant signals required to understand the simulation traces.

Discussion of the simulation trace in Fig. 5. 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 $e^{2619} \vDash \sigma_{S_{B\uparrow}}$ holds. Similarly, the other atomic propositions (as shown in Fig. 5) of the specification are evaluated to: $e^{2619} \nvDash \sigma_{S_{B\downarrow}}$, $e^{2619} \vDash \sigma_{S_{L\uparrow}}$, and $e^{2619} \nvDash \sigma_{S_{L\downarrow}}$. Additionally (not explicitly mentioned in Fig. 5) the example uses the atomic propositions: $\sigma_{S_{P\uparrow}} = (\textsf{pitch} \geq 5°)$ and $\sigma_{S_{P\downarrow}} = (\textsf{pitch} \leq 2°)$ to monitor a significant up-/down pitching of the UAS (from the IMU sensors). $\sigma_{S_{V\uparrow}} = (\textsf{vel\_up} \geq 2\frac{m}{s})$ and $\sigma_{S_{V\downarrow}} = (\textsf{vel\_up} < -2\frac{m}{s})$ to monitor a significant up/down velocity of the UAS (from the IMU sensors). $\sigma_{S_{ct}} = (\textsf{cmd} == \text{takeoff})$ and $\sigma_{S_{cl}} = (\textsf{cmd} == \text{land})$ 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 $e^{2619} \vDash \sigma_{S_{P\uparrow}}$, $e^{2619} \nvDash \sigma_{S_{P\downarrow}}$, $e^{2619} \nvDash \sigma_{S_{ct}}$, $e^{2619} \nvDash \sigma_{S_{cl}}$, and $e^{2619} \vDash \sigma_{S_h}$.

Discussion of the simulation trace in Fig. 6. Initially, all inputs to the altimeter health model indicate an increasing altitude (i.e., $e^n \vDash \sigma_{S_{B\uparrow}}$, $e^n \vDash \sigma_{S_{L\uparrow}}$, and $e^n \vDash \varphi_{S_{S\uparrow}}$). The posterior marginals Pr(baro_alt=OK | $\pi \vDash \{\sigma_{S_{B\uparrow}}, \sigma_{S_{L\uparrow}}, \varphi_{S_{S\uparrow}}\}$), and Pr(baro_alt=BAD | $\pi \vDash \{\sigma_{S_{B\uparrow}}, \sigma_{S_{L\uparrow}}, \varphi_{S_{S\uparrow}}\}$), and Pr(laser_alt=OK | $\pi \vDash \{\sigma_{S_{B\uparrow}}, \sigma_{S_{L\uparrow}}, \varphi_{S_{S\uparrow}}\}$), and Pr(laser_alt=BAD | $\pi \vDash \{\sigma_{S_{B\uparrow}}, \sigma_{S_{L\uparrow}}, \varphi_{S_{S\uparrow}}\}$), 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, $e^n \nvDash \sigma_{S_{L\downarrow}}$ holds and indicates a *decrease* in altitude, while the other inputs to the altimeter health model disagree and indicate an *increase* (i.e., $e^n \vDash \sigma_{S_{B\uparrow}}$, $e^n \vDash \varphi_{S_{S\uparrow}}$ 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 | $\pi \vDash \{\sigma_{S_{B\uparrow}}, \sigma_{S_{L\uparrow}}, \varphi_{S_{S\uparrow}}\}$)), while the belief in a healthy altimeter reading remains high (signal Pr(baro_alt=OK | $\pi \vDash \{\sigma_{S_{B\uparrow}}, \sigma_{S_{L\uparrow}}, \varphi_{S_{S\uparrow}}\}$)).

Table 2. Interpretation of the simulation signals in Fig. 5 and Fig. 6: *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
↪ maybe	set if output of the observer $\widehat{\textbf{eval}}\,(\varphi_1)$ is **maybe**, cleared otherwise.
↪ value	set if output of the observer $\widehat{\textbf{eval}}\,(\varphi_1)$ is **true**, cleared if $\widehat{\textbf{eval}}\,(\varphi_1)$ is **false**
phi_1 (async)	output of asynchronous observer for flight rule φ_1
↪ value	set if verdict of the output tuple $T_{\varphi_1}.\mathsf{v}$ = **true**, cleared otherwise.
↪ time	time stamp of the output tuple ($T_{\varphi_1}.\tau_e$)
phi_2 (sync)	output of synchronous observer for flight rule φ_2
↪ maybe	set if output of the observer $\widehat{\textbf{eval}}\,(\varphi_2)$ is **maybe**, cleared otherwise.
↪ value	set if output of the observer $\widehat{\textbf{eval}}\,(\varphi_2)$ is **true**, cleared if $\widehat{\textbf{eval}}\,(\varphi_2)$ is **false**
phi_2 (async)	output of asynchronous observer for flight rule φ_2
↪ value	verdict of the output tuple ($T_{\varphi_2}.\mathsf{v}$)
↪ time	time stamp of the output tuple ($T_{\varphi_2}.\tau_e$)
phi_S_s_up (async)	output of asynchronous observer for $\varphi_{S_{s\uparrow}}$
↪ value	set if verdict of the output tuple $T_{\varphi_{S_{s\uparrow}}}.\mathsf{v}$ = **true**, cleared otherwise.
↪ time	time stamp of the output tuple ($T_{\varphi_{S_{s\uparrow}}}.\tau_e$)
phi_S_s_down (async)	output of asynchronous observer for $\varphi_{S_{s\downarrow}}$
↪ value	set if verdict of the output tuple $T_{\varphi_{S_{s\downarrow}}}.\mathsf{v}$ = **true**, cleared otherwise.
↪ 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. 6)	
Pr(baro_alt=OK | $pi \models \varphi$)	posterior marginal (likelihood of a good barometric altimeter reading)
Pr(baro_alt=BAD | $pi \models \varphi$)	posterior marginal (likelihood of a bad barometric altimeter reading)
Pr(laser_alt=OK | $pi \models \varphi$)	posterior marginal (likelihood of a good laser altimeter reading)
Pr(laser_alt=BAD | $pi \models \varphi$)	posterior marginal (likelihood of a bad laser altimeter reading)

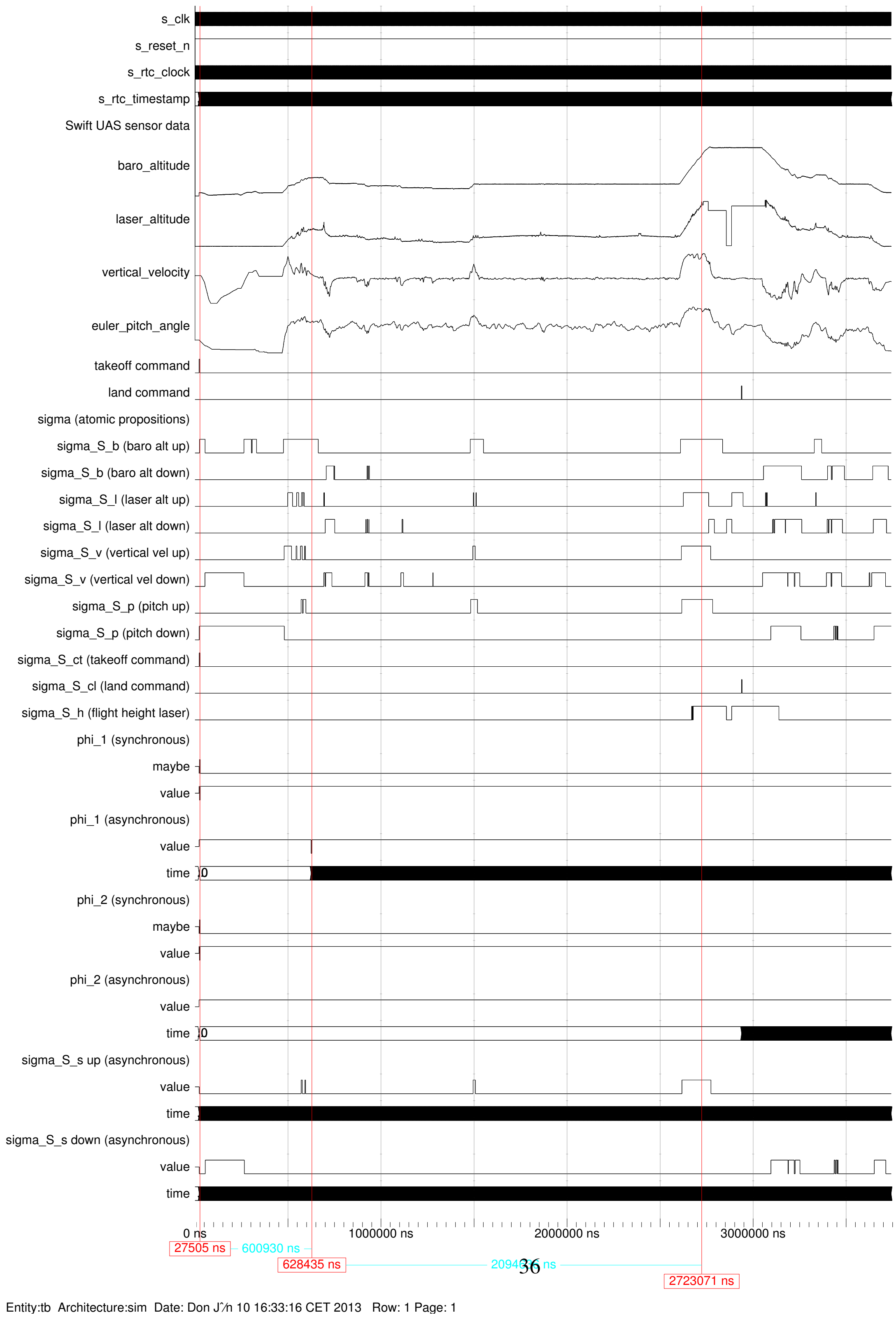


Fig. 5. Hardware simulation traces for a complete test-flight data of the Swift UAS.

10 Appendix D – Hardware Platform

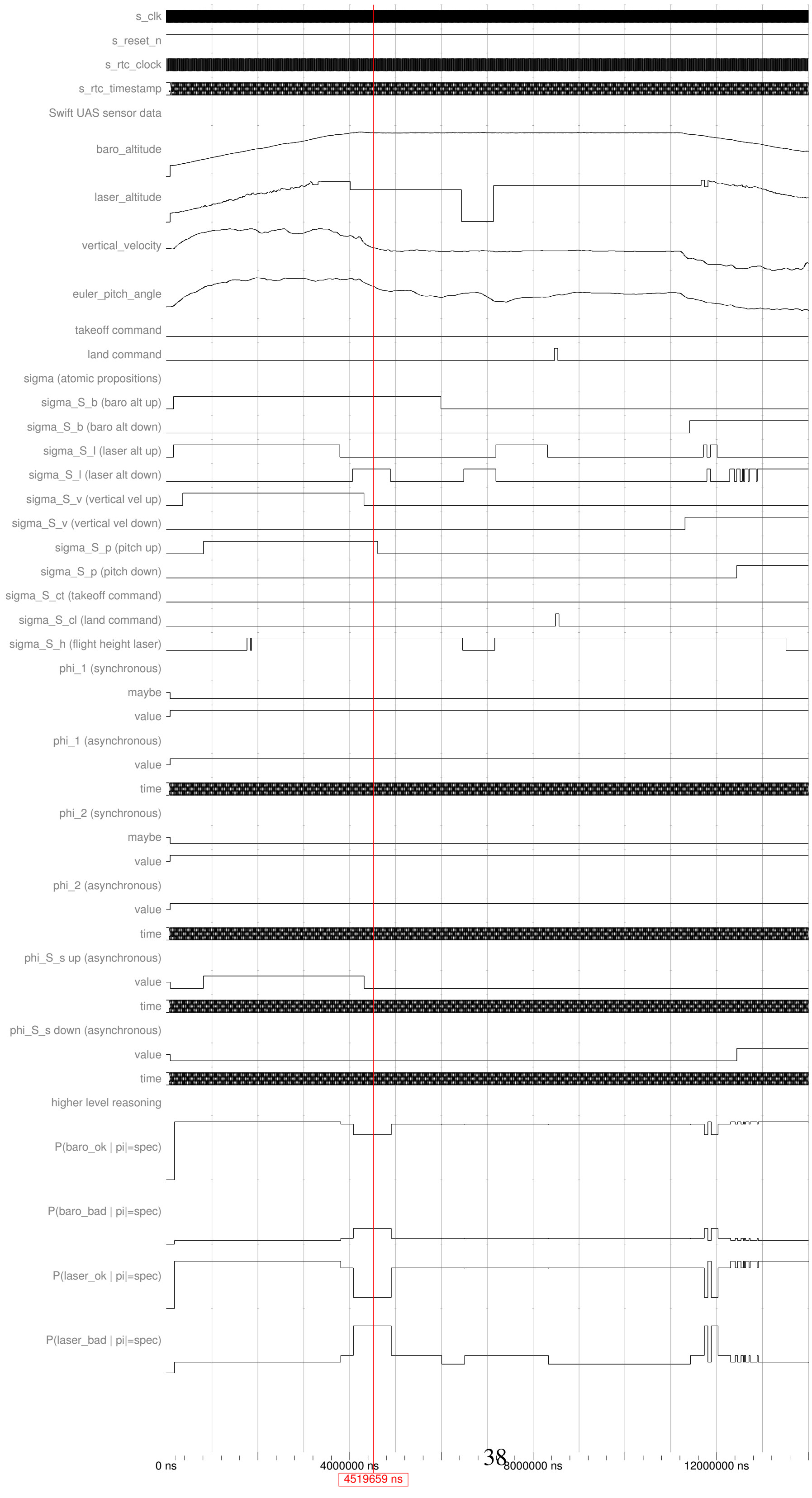


Fig. 6. A section (laser altimeter outage) of the simulation traces with health assessment.

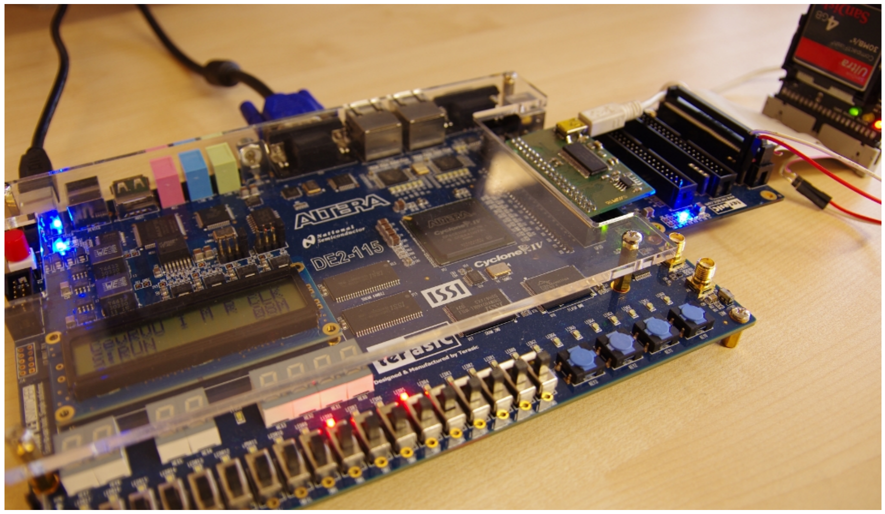


Fig. 7. FPGA target board (Altera DE2-115)

References

32. Alur, R., Henzinger, T.A.: Real-time Logics: Complexity and Expressiveness. In: LICS. pp. 390–401. IEEE (1990)
33. 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)
34. Barre, B., Klein, M., Soucy-Boivin, M., Ollivier, P.A., Hallé, S.: MapReduce for parallel trace validation of LTL properties. In: RV. LNCS, vol. 7687, pp. 184–198. Springer (2012)
35. Barringer, H., Falcone, Y., Finkbeiner, B., Havelund, K., Lee, I., Pace, G.J., Rosu, G., Sokolsky, O., Tillmann, N. (eds.): Runtime Verification (RV), LNCS, vol. 6418. Springer (2010)
36. Basin, D., Klaedtke, F., Müller, S., Pfitzmann, B.: Runtime monitoring of metric first-order temporal properties. In: FSTTCS. pp. 49–60 (2008)
37. Basin, D., Klaedtke, F., Zălinescu, E.: Algorithms for monitoring real-time properties. In: RV. pp. 260–275. LNCS, Springer (2011)
38. Bauer, A., Leucker, M., Schallhart, C.: Comparing LTL semantics for runtime verification. J. Log. and Comp. 20, 651–674 (2010)
39. Bauer, A., Leucker, M., Schallhart, C.: Runtime verification for LTL and TLTL. ACM Trans. Softw. Eng. M. 20, 14:1–14:64 (2011)
40. Colombo, C., Pace, G., Abela, P.: Safer asynchronous runtime monitoring using compensations. FMSD 41, 269–294 (2012)
41. Darwiche, A.: Modeling and Reasoning with Bayesian Networks. Cambridge University Press, New York, NY, USA, 1st edn. (2009)
42. Divakaran, S., D'Souza, D., Mohan, M.R.: Conflict-tolerant real-time specifications in metric temporal logic. In: TIME. pp. 35–42 (2010)
43. Finkbeiner, B., Kuhtz, L.: Monitor circuits for LTL with bounded and unbounded future. In: RV, LNCS, vol. 5779, pp. 60–75. Springer (2009)
44. Fischmeister, S., Lam, P.: Time-aware instrumentation of embedded software. IEEE Trans. Ind. Informatics 6(4), 652–663 (2010)
45. Geilen, M.: An improved on-the-fly tableau construction for a real-time temporal logic. In: CAV. pp. 394–406 (2003)
46. Havelund, K.: Runtime verification of C programs. In: TestCom/FATES. LNCS, vol. 5047, pp. 7–22. Springer (2008)
47. 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)

48. 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)
49. Kleene, S.C.: Introduction to Metamathematics. North Holland (1996)
50. Kogge, P.M., Stone, H.S.: A parallel algorithm for the efficient solution of a general class of recurrence equations. IEEE Trans. Comput. 22(8), 786–793 (Aug 1973)
51. Kroening, D., Strichman, O.: Decision Procedures: An Algorithmic Point of View. Springer (2008)
52. Lichtenstein, O., Pnueli, A., Zuck, L.: The glory of the past. In: Logics of Programs, LNCS, vol. 193, pp. 196–218. Springer (1985)
53. 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 (2007)
54. Maler, O., Nickovic, D., Pnueli, A.: On synthesizing controllers from bounded-response properties. In: CAV. LNCS, vol. 4590, pp. 95–107. Springer (2007)
55. Maler, O., Nickovic, D., Pnueli, A.: Checking temporal properties of discrete, timed and continuous behaviors. In: Pillars of Comp. Science. pp. 475–505. Springer (2008)
56. Pike, L., Niller, S., Wegmann, N.: Runtime verification for ultra-critical systems. In: RV. pp. 310–324. LNCS, Springer (2011)
57. Reinbacher, T., Függer, M., Brauer, J.: Real-time runtime verification on chip. In: RV. LNCS, vol. 7687, pp. 110–125. Springer (2012)
58. Schumann, J., Mbaya, T., Mengshoel, O., Pipatsrisawat, K., Srivastava, A., Choi, A., Darwiche, A.: Software health management with Bayesian Networks. Innovations in Systems and SW Engineering 9(4), 271–292 (2013)
59. 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: PHM (2013)
60. Tabakov, D., Rozier, K.Y., Vardi, M.Y.: Optimized temporal monitors for SystemC. Formal Methods in System Design 41(3), 236–268 (2012)
61. Thati, P., Roşu, G.: Monitoring Algorithms for Metric Temporal Logic specifications. ENTCS 113, 145–162 (2005)
62. Wegener, I.: The Complexity of Boolean Functions. Wiley-Teubner Series in Computer Science, John Wiley & Sons (1987), isbn: 3519021072