Kalpa Publications in Computing
Volume 3, 2017, Pages 138–156
RV-CuBES 2017. An International Workshop on
Competitions, Usability, Benchmarks, Evaluation,
and Standardisation for Runtime Verification Tools

R2U2: Tool Overview*

Kristin Y. Rozier[1] and Johann Schumann[2]

[1] Iowa State University, Ames, IA, USA, KYRozier@iastate.edu
[2] SGT, Inc., NASA Ames, Moffett Field, CA, USA, Johann.M.Schumann@nasa.gov

Abstract

R2U2 (Realizable, Responsive, Unobtrusive Unit) is an extensible framework for runtime System Health Management (SHM) of cyber-physical systems. R2U2 can be run in hardware (e.g., FPGAs), or software; can monitor hardware, software, or a combination of the two; and can analyze a range of different types of system requirements during runtime. An R2U2 requirement is specified utilizing a hierarchical combination of building blocks: temporal formula runtime observers (in LTL or MTL), Bayesian networks, sensor filters, and Boolean testers. Importantly, the framework is extensible; it is designed to enable definitions of new building blocks in combination with the core structure. Originally deployed on Unmanned Aerial Systems (UAS), R2U2 is designed to run on a wide range of embedded platforms, from autonomous systems like rovers, satellites, and robots, to human-assistive ground systems and cockpits.

R2U2 is named after the requirements it satisfies; while the exact requirements vary by platform and mission, the ability to formally reason about Realizability, Responsiveness, and Unobtrusiveness is necessary for flight certifiability, safety-critical system assurance, and achievement of technology readiness levels for target systems. Realizability ensures that R2U2 is sufficiently expressive to encapsulate meaningful runtime requirements while maintaining adaptability to run on different platforms, transition between different mission stages, and update quickly between missions. Responsiveness entails continuously monitoring the system under test, real-time reasoning, reporting intermediate status, and as-early-as-possible requirements evaluations. Unobtrusiveness ensures compliance with the crucial properties of the target architecture: functionality, certifiability, timing, tolerances, cost, or other constraints.

1 Introduction

The need for formal reasoning about safety-critical systems is widely recognized, and design-time verification techniques, such as symbolic model checking [21], have had major impacts on the development of real-life systems, particularly in the aerospace industry [3, 34, 13, 36, 8, 12, 6, 20], in part because they enable automated checking against specifications written in an intuitive

*The R2U2 homepage is http://temporallogic.org/research/R2U2/. Work supported in part by NASA Autonomy Operating System (AOS) 8042-018286_NASA NNX14AN61A, NASA ECF NNX16AR57G, and NSF CAREER Award CNS-1552934.

language like Linear Temporal Logic (LTL). For today's complex, cyber-physical systems, exhaustive verification is not achievable for all subsystems; we also need to carry practical, automated checks forward to runtime verification. For intelligent and autonomous systems, we need to go a step further, to provide System Health Management (SHM). Autonomous systems are only capable of effective self-governing if they can reliably sense their own faults and respond to failures and uncertain environmental conditions. Providing SHM for systems that are safety-critical, or that must meet standards for flight certification, is particularly challenging due to the constraints placed on on-board technologies.

R2U2, the Realizable, Responsive, Unobtrusive Unit, is named after the requirements it is designed to uphold. These requirements are defined by the FAA, NASA, and regulatory bodies; for example flight-certifiability in many countries depends on FAA DO-178B [15], DO-178C [18], DO-333 [17], and DO-254 [16]. While the exact requirements for Realizability, Responsiveness, and Unobtrusiveness vary with the system platform, TRL (Technology Readiness Level), degree of safety-criticality, and other factors, the central goal of R2U2's design is to provide a capability for system health management that is sufficiently adaptable and extensible as to operate effectively and performably while respecting those requirements. For example, Unobtrusiveness for the already-flight-certified Swift UAS dictated that R2U2 needed to maintain a read-only interface with the system bus [19, 7], whereas for NASA's Autonomy Operating System (AOS), R2U2 can publish messages to the system bus without violating Unobtrusiveness requirements.

Responsiveness R2U2 continuously monitors adherence to the safety requirements of the target system in real time. Changes in the validity of monitored requirements are detected within a tight and a priori known time bound. Uniquely, R2U2 reports *both* the satisfaction and violation of requirements *as early as possible*, e.g., within one time step of the information required to determine compliance becoming available, as well as updates on the intermediate status of each requirement at every time step. Responsive requirement monitoring enables responsive inputs required for real-time prognostics and intelligent decision-making, enabling mitigation of any problems encountered to avoid damage to the target system and its environment. Intermediate updates ensure liveness of the R2U2 subsystem and allow for time-triggered processing of R2U2 outputs, in addition to event-triggered, e.g., by the success or failure to meet a requirement.

Realizability System health management with R2U2 becomes realizable via the plug-and-play architecture, expressive specification format, and generic interface suitable for a wide variety of target systems. Each requirement checked by R2U2 is specified by encoding an *observation tree*, a hierarchical combination of building blocks: temporal formula runtime observers (in LTL or MTL), Bayesian networks, sensor filters, Boolean testers. Because R2U2 has both hardware and software implementations, R2U2 is adaptable to many different architectures and platforms, including operating on an FPGA, in software on the microcontroller, in software on the flight computer, or in both hardware and software in a hybrid cyber-physical system. Different implementations enable scalability, extensibility, and specialization for reasoning with different software and sensor variables, with different timing guarantees. Our design is reconfigurable so that temporal logic observers can be updated without a lengthy re-compilation process and can be used both during testing of the target system and after deployment. This feature enables R2U2 to check different observation trees during different mission stages, such as take-off, approach, measurement/work, return, and landing.

Unobtrusiveness Our multi-architecture, multi-platform design enables effective runtime

verification while respecting crucial properties of (possibly autonomous) target systems, including functionality (not change behavior), certifiability (provide timing guarantees and enable safety cases), timing (not interfere with timing guarantees), and tolerances (not exhaust constraints for size, weight, power, telemetry bandwidth, software overhead). The adaptability of the R2U2 observation trees and the ability to re-use configurations for standard, or commercial-off-the-shelf (COTS) components ease compliance with development-time and budget constraints.

1.1 Key Features of R2U2

R2U2 specification format:

1. **Signal Processing**: Preparation of sensor readings
   - **Filtering**: processing of incoming data
   - **Discretization**: generation of Boolean outputs
   - **Prognostics**: prediction of component life
2. **Temporal Logic (TL) Observers**: Efficient temporal reasoning
   (a) **Asynchronous**: output $\langle t, \{0, 1\}\rangle$
   (b) **Synchronous**: output $\langle t, \{0, 1, ?\}\rangle$
   - **Logics**: MTL, pt-MTL, Mission-time LTL
   - **Variables**: Booleans (from system bus), sensor filter outputs
3. **Bayes Nets**: Efficient decision making
   - **Variables**: outputs of TL observers, sensor filters, Booleans
   - **Output**: most-likely status + probability

Figure 1: R2U2 system health management framework in a nutshell (t is time) [19, 30, 23].

Specifications. In industrial systems, languages and formats for specification vary widely and are often tailored to specific applications. The question of how best to encapsulate specifications from real systems is an ongoing research question [23]. R2U2's Realizability requirement encapsulates the need for specifications that are cross-language, hierarchical, compositional, and extensible. Figure 1 summarizes R2U2 specifications, which combine two encodings for each linear-time temporal logic formula, which may be in one of several variants of LTL, with efficient (non-dynamic) Bayes Nets to provide diagnostic decision-making capabilities. R2U2 provides capabilities to perform signal data processing, discretization, and model-based prognostics [31]. Cyber-physical, autonomous systems often utilize hierarchical, multi-formalism specifications; see, e.g., [35].

As an example, Figure 2 displays a pictorial representation of an observation tree for determining if a fault has occurred in the fluxgate magnetometer during runtime. **We use this observation tree specification as a running example in Section 3.** From the manual, we know that there are five possible faults that can occur. We read the relevant sensor data from the fluxgate magnetometer and other on-board sensors useful in cross-checking it from

Health Nodes / Failure Modes	
H_FG	Magnetometer sensor
H_FC_RxUR	Receiver underrun
H_FC_RxOVR	Receiver overrun
H_FG_TxOVR	Transmitter overrun in sensor
H_FG_TxErr	Transmitter error in in sensor

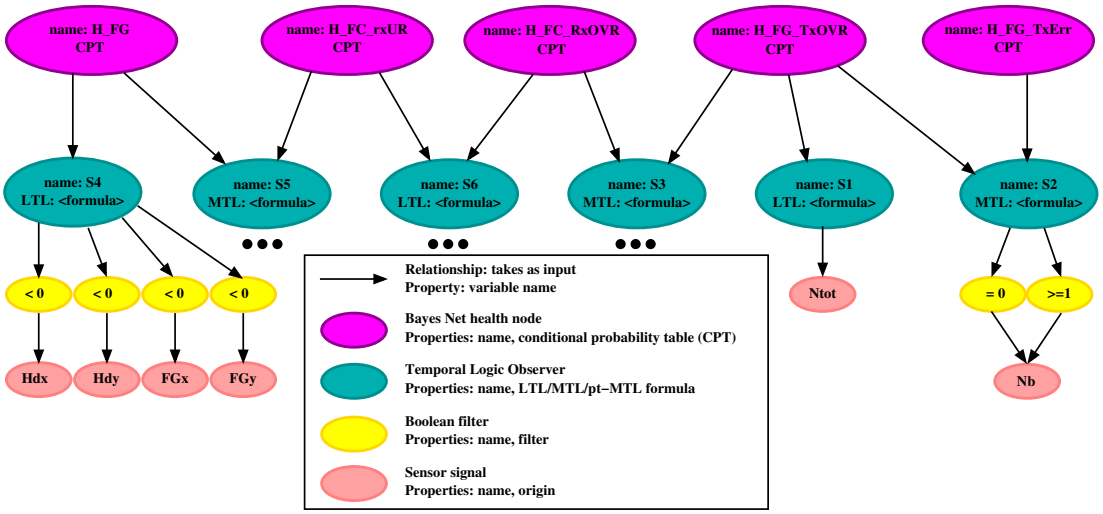

Figure 2: R2U2 configurations are sets of observation trees, like this one [23]. The possible failures a fluxgate magnetometer can suffer can be diagnosed by a Bayes Net with a health node corresponding to each type of failure. These nodes take as input the valuations from six temporal logic runtime observers; many failures require inputs from multiple temporal observers in order to make an accurate diagnosis [7]. The temporal logic observers reason about variables populated by Boolean testers over filtered sensor streams.

the system bus, filter this data, and then pass it through Boolean testers to supply variables to temporal logic formulas. We can write six relevant temporal logic specifications that we encode as runtime observers outputting statuses S1, ..., S6. The outputs from these runtime observers are inputs to five Bayesian health nodes, one for determining whether it is probabilistically likely that each possible fault has occurred. A health node may hierarchically depend on the output from more than one runtime sensor node and the runtime observers may supply temporal information to multiple health nodes.

Outputs. R2U2 has the ability to produce an output stream for each observation tree in one of several different formats: a 2-tuple of a Boolean valuation of a temporal logic formula observer paired with the time step of the verdict (for synchronous or asynchronous observers), a 2-tuple of most likely status and probability of that status from a Bayes Net, or something else (outputs from Boolean testers, or user-defined extensions). In future work we are examining other formats that would be useful as inputs to other subsystems that may functionally depend upon, or utilize in decision-making, the system's health.

Timing. R2U2 observation trees are adaptable to different timing of input sensor streams and output streams. Our algorithms are based on the temporal logic notion of time steps, which we have in practice resolved to ticks of the system clock, or seconds, though other frequencies

are also possible.

Uniquely, for every future-time temporal logic formula in an observation tree, R2U2 encodes *two* runtime observers. An asynchronous, or event-triggered observer reports both satisfaction and violation of the formula as early as possible, e.g., within the next time step after the information required to determine the verdict is known. Asynchronous observers also aggregate values, back-filling verdicts for many previous time steps at once. For example, a requirement that variable x must be true for two consecutive time steps cannot be resolved immediately if x is `true` at time step 0; if x is then `false` at time step 1, then at time step 1 we know the verdict of this formula for times 0 and 1 are both `false`. We pair each asynchronous observer with a three-valued, time-triggered, synchronous observer that outputs the status of the formula {`true`, `false`, `maybe`} at each time step. This provides an easy liveness check, serves as a simple sanity check for the asynchronous observers, and enables straightforward interfacing with time-triggered on-board systems utilizing the outputs of R2U2.

Platform. Both R2U2 temporal logic observers and Bayes Net decision nodes have both hardware and software implementations. One can run any combination of hardware, software, or cyber-physical implementations depending on the resources available and constraints for flight certifiability. Also, running multiple versions of the same algorithm on different platforms (hardware and software) is a useful sanity check, particularly for long-term robust execution in harsh environments, like space.

Targets. R2U2 monitors hardware, software, or a hybrid combination for cyber-physical systems. Observation trees can reason over a variety of types of input data: sensor streams, messages from message-passing operating systems (like ROS or NASA cFS/cFE), software variable values sent in one of these forms, or shared memory. The key feature enabling Unobtrusiveness is that one specific type of system instrumentation for gathering data is not required. The limiting factor in R2U2 execution has historically been bandwidth: we have had the monitoring targets limited by the UART connection in previous case studies [7].

Unique Implementation Elements. R2U2's efficient algorithms for encoding and reasoning about observation trees have several implementation elements that are unique among runtime verification tools:

- **Hardware temporal logic encoding without automata.** We encode temporal logic formulas as parallel-execution circuits rather than the more standard method of translating them into automata. To perform the arithmetic operations on time stamps required by asynchronous observers, we map registers and flags to circuits that can store information, such as flip-flops.
- **Reconfiguration without resynthesis.** Our hardware implementation design provides the tremendous advantages of not having to be resynthesized between missions (because changing the TL monitors requires only, e.g., sending new binaries representing the formulas to the FPGA), and being able to run in parallel with the software and sensor systems R2U2 monitors with no overhead because we are not generating, e.g., additional threads or parallel programs to perform the monitoring.
- **Monitoring without software instrumentation.** We perform software monitoring *without software instrumentation* through language-independent channels such as passing software values over the system bus or accessing shared memory. This enables monitoring

of software that cannot be modified, is ITAR, or is otherwise restricted/closed-source, provided the software publishes some record R2U2 can read.

- **Aggregated output streams.** The output streams for asynchronous observers aggregate results from multiple time steps. Our aggregation function repeatedly replaces two consecutive elements in our verdict queue by the latter tuple when their verdicts match. Formally, let $\langle T_\varphi \rangle$ be the 2-tuple output stream evaluating formula φ such that for every generated output tuple T we have that $T.v \in \{\textbf{true}, \textbf{false}\}$ is the verdict and $T.\tau_e \in [0, n]$ is the time step corresponding to that verdict. Then aggregation replaces two consecutive elements $\langle T^i_\varphi \rangle, \langle T^{i+1}_\varphi \rangle$ in $\langle T_\varphi \rangle$ by $\langle T^{i+1}_\varphi \rangle$ iff $\langle T^i_\varphi \rangle.v = \langle T^{i+1}_\varphi \rangle.v$.

- **Shared connection queues.** Elements of observation trees pass data through *shared connection queues*; a shared connection queue resembles a FIFO queue with multiple read pointers and one write pointer pointing to the "last" queue element. However, the queues implement aggregation efficiently by modifying the time step of the "last" queue element if the new element's verdict is the same, leaving the queue size unchanged. This helps us to prove bounds on queue size. Also, verdicts for past time steps may or may not affect changes to the output queue, depending on their current relevance. Consequences include that some common queue functions, such as checking for emptiness or fullness, are implemented differently than for FIFO queues.

1.2 Runtime Verification Problems Solved by R2U2

When is R2U2 a suitable option for system health management? If all of the following conditions hold, R2U2 may be of use.

- System requirements can be described by an observation tree using any hierarchical combination of R2U2's many building blocks:
  - sensor filters
  - Boolean testers
  - LTL (which we translate into "Mission Time LTL" [19])
  - MTL (or pt-MTL)
  - Bayes Net
  - User-defined building block
- Data required for input to observation trees is available from any of the following:
  - system bus
  - shared memory
  - an external interface (e.g., UART)
- R2U2 may be installed on some flight computer or on-board hardware, e.g., FPGA, without violating flight certifiability or resource usage constraints.

The remainder of this paper is organized as follows. Section 2 overviews the R2U2 framework, including high-level details of the architecture and tool chain. a brief history, the architectural variants of R2U2, and our plans for public dissemination of these. Section 3 exemplifies runtime system health management using R2U2, demonstrating its key features and providing tips for usability. Section 4 overviews some case studies and missions that have utilized R2U2 for real-life applications. Section 5 concludes and lists on-going and future developments while Section 6 thanks the people and programs who have supported this work.

2 The R2U2 Framework

2.1 Overview of R2U2 Model Elements

R2U2 "models" are actually configurations of its building blocks; we call one such configuration an observation tree due to its compositional structure. The building blocks can be temporal logic formulas, Bayesian networks, or specifications of signal-preprocessing and filtering. These models can be designed in a modular and hierarchical manner to enable the designer to easily express properties containing temporal, model-based, and probabilistic aspects.

As a runtime verification tool, which also can perform diagnostic reasoning, R2U2 shares many commonalities with Fault Detection and Diagnosis (FDD) systems. In their simplest variant, FDD systems use Boolean conditions to detect off-nominal conditions or violated properties. Additional expressiveness can be achieved by using model-based diagnosis, temporal reasoning, or probabilistic reasoning. Figure 3 shows this abstraction space and lists a number of prominent related systems.

We designed our R2U2 framework to incorporate temporal logic (LTL and MTL) reasoning with temporal observer pairs, probabilistic reasoning using (static) Bayesian networks, as well as model-based components like Kalman filters or prognostics engines.

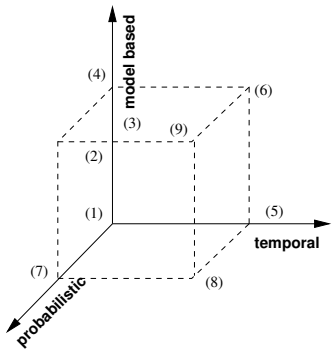

Figure 3: [33] Abstraction space for SHM along the dimensions of temporal, model-based, and probabilistic reasoning. In this figure, (1) corresponds to Boolean conditions, (2) to paradigms similar to QSi/TEAMS [14], (3) to Livingstone [2], (4) to HyDe [9], (5) to temporal logic, (6) to FACT/TFPG [10], (7) to (static) Bayesian networks (BN), (8) to dynamic BN, and (9) to our R2U2 framework.

2.2 Architecture

R2U2 reasons over (filtered) data streams; these can be from sensors, actuators, or the flight software. Depending on the system's Unobtrusiveness requirements, R2U2 can use a read-only (serial) interface (Figure 4), or can output data on the system's current health. The standard output from R2U2 is a set of the outputs for each observation tree at every time step. The output from one observation tree is the output from the observer at the root of the

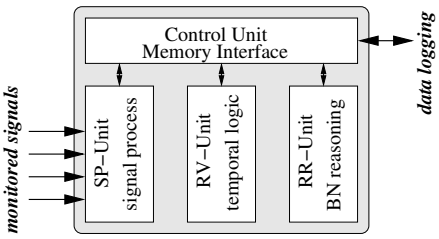

Figure 4: [30] Hardware R2U2 implementation

tree, which is most often a 2-tuple of the Boolean verdict from a temporal logic observer paired with the time step of that verdict, or a 2-tuple from a Bayes Net containing the most likely health status and the estimated probability for that status.

Signals from the flight computer software and communication buses are filtered and discretized in the signal processing (SP) unit to obtain streams of propositional variables. The runtime verification (RV) and runtime reasoning (RR) units comprise the health management hardware: the RV unit monitors TL properties using pairs of synchronous and asynchronous observers defined in [19]. By compiling every TL formula (and its subformulas) into a pair of asynchronous, or event-driven, and synchronous, or time-driven, observers, we can better enable intermediate evaluation of the status of the formula for better decision making. After the TL formulas have been evaluated, the results are transferred to the RR subsystem, where the compiled Bayesian network is evaluated to yield the posterior marginals of the health model.

R2U2 continuously monitors multiple signals during runtime with minimal instrumentation of the flight software, which is essential for UNOBTRUSIVENESS: altering safety-critical software or hardware components can cause difficulties in maintaining flight certification.

2.3 Tool Chain

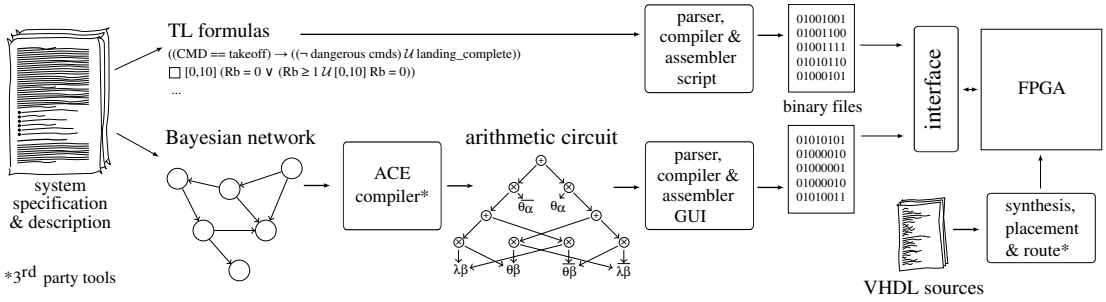

Figure 5: [29] R2U2 tool chain

The individual components of an R2U2 observation-tree model are specified using a simple language for the temporal observers and a spreadsheet-like method for the signal specifications. The Bayesian network is developed using a graphical tool like SamIam.[1] Input-output specifications must be set up in a textual form. Our current tool chain does not include a GUI-based development environment; this is future work.

The individual parts of the R2U2 specification are then processed by our tool chain (Figure 5): the temporal formulas are compiled into binary code to be executed by the temporal processors. The Bayesian network is converted into an Arithmetic Circuit [5], which allows R2U2 to perform efficient and time-bounded probabilistic reasoning. The specification of the signal preprocessing generates VHDL configurations for our FPGA implementation or customized C code.

2.4 History

R2U2 began as a collaboration at NASA Ames Research Center in 2013. It resulted from a combination of inspirations including the need for a system health management capability that

[1] SamIam (Sensitivity Analysis, Modeling, Inference and More) http://reasoning.cs.ucla.edu/samiam/

could really fly, e.g., because it obeyed flight certification requirements; recent developments in temporal logic runtime verification; and the idea that intelligently fusing multiple different runtime reasoning capabilities could create a better SHM core than any of them running individually. R2U2 also was inspired by a NASA-developed Software Health Management system that used Bayesian networks [27]. The first, hardware, implementation of R2U2 was supported by a 2014 NARI Seedling Grant; initial work for that proposal, combined with the first funded results was published by Reinbacher, Rozier, and Schumann [19]. Ideas for the reconfigurable building blocks that comprise observation trees were published at the same time [32], and later extended to a journal version via folding in grant deliverables [33].

2.5 R2U2 Realizations

The two core building blocks of R2U2, temporal logic runtime observers and Bayesian reasoners, each have both a hardware and a software implementation. We implement the same algorithms in both hardware and software for several reasons:

- Having both hardware and software options enables R2U2 to meet the Unobtrusiveness requirement: some platforms allow software to send values to R2U2 while for others it would violate flight certification. Similarly, there may be limits on flight hardware.
- Realizability and scalability of the implementations stem from different options for accessing data with different timing guarantees. The hardware implementation has zero-overhead (versus the low-overhead software version), and offers tighter timing guarantees that are easier to bound versus a parallel software execution (potentially running on an isolated computer). Running R2U2 in software may avoid having to send the input software and sensor values through, e.g., a UART, to the hardware implementation but loosens the guarantees we can prove on the time bound for formula evaluation.
- In some cases, it may be optimal to monitor software with software and hardware with hardware, or vice versa. Considerations include ease of inputting data and outputting verdicts, and the availability of on-board implementation platforms with the capacity and authorization to execute R2U2.
- Who checks the checkers? By having multiple instances of R2U2 that are able to check each other, we increase robustness of the SHM observation trees. Our goal is to ensure that a verdict indicating a fault is triggered by an actual fault occurring, avoiding false positives caused by malfunctions of runtime verification tools.
- Hardware and software fail differently. If we have two implementations of R2U2 both checking the same observation trees, it is less likely that they will both suffer faults themselves and therefore fail to detect a fault they are monitoring for.

Hardware The (original) hardware version of R2U2 is implemented in VHDL that is compiled into an FPGA configuration; see Figure 6. The hardware implementation requires software that programs the gates to run TL observer binaries in a way that is demonstrably correct and complete, i.e., handles full future-time Metric Temporal Logic (MTL) and "Mission Time" Linear Temporal Logic (LTL) [19] observers (both synchronous and asynchronous), without restricting their size, e.g., length or number of variables. This software generates a single hardware description language (HDL) design to load on the FPGA to program it for SHM of a particular autonomous system. Our hardware implementation design provides the tremendous advantages of not having to be resynthesized between missions (because changing the TL monitors requires only sending new binaries representing the formulas to the FPGA), and being able to run in parallel with the software and sensor systems it monitors with no overhead because we are not

generating, e.g. additional threads or parallel programs to perform the monitoring.

To execute the TL monitors on the FPGA requires an optimized TL-to-binary translator. We can then prove real-time bounds on the evaluation of the monitors. Specifically, we can prove that each formula can be evaluated in one tick of the system clock. For our experiments, we use an Adapteva Parallella board [1] that provides a suitable FPGA and runs a Linux system for data logging and development.

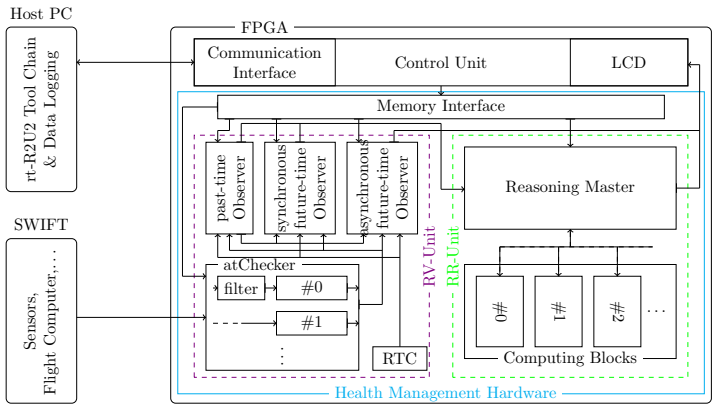

Figure 6: [7] Overview of the R2U2 hardware architecture on a Xilinx board. The main components are the RV-unit, which contains signal processing and processors for future (and past) time observers, the RR-unit for Bayes Net reasoning, and an interface and communication unit. The FPGA architecture running on the Parallella board combines both external interfaces into a single serial interface, communicating with a Linux process.

Software The software versions of R2U2 have an architecture that is similar to the hardware version; the temporal observers are implemented as special-purpose processors that are emulated in software (currently C or C++). This modular design enables us to re-use our tool chain and observation trees for different R2U2 configurations on-board different systems. The following are example configurations:

- A stand-alone R2U2 version that interfaces to Matlab[2] and Octave[3]. The interface provides a function to compile an R2U2 observation tree from within Matlab or Octave, and a function to execute R2U2 on a time-series of vectors, stored as a matrix. This version is mainly used for model development, debugging, and post-mortem analysis.
- A message-passing version of R2U2 is integrated into the AOS (Autonomy Operating System) [11], which is built on top of NASA's cFE/cFS (core Flight Executive/core Flight System) software[4]. During flight, the R2U2 application monitors messages on the software bus that contain sensor and software data and publishes outputs to the message bus to update other apps on the current system health. R2U2 can be ported easily to other cFS/cFE-based systems.
- A parallel version of R2U2 running on a 16 or 64 core Epiphany[5] chip [28] on a Parallella board [1] executes independent temporal formula observers on different cores in parallel

[2] http://mathworks.com
[3] http://octave.org
[4] NASA Core Flight System https://cfs.gsfc.nasa.gov/
[5] http://epiphany.org

to the Bayes Net execution and a parallelized version of the signal processing. A direct-memory-access-based technique allows R2U2 to access values of relevant variables of the monitored software (e.g., flight controller) without the need for software instrumentation and with minimal overhead.

Hybrid versions of R2U2, where some R2U2 components are executed on the FPGA and others are implemented in software can be set up and have been used in various case studies.

2.6 Current Status and How to Obtain R2U2

R2U2 is an innovative, low-TRL framework for runtime verification and system health management of advanced air and space technologies [4]. The hardware and software implementations of R2U2 are currently research prototypes. While they correctly and scalably implemented the capabilities required for all of our case studies, the implementations were not extensively validated to be complete, correct for all possible configurations, and exactly matching the algorithms we have defined and proved correct, e.g., in [19]. Hardware and software implementations are currently undergoing extensive revisions to help close this gap. Most of the R2U2 modules are generic and can be used on all platforms. Interfaces that obtain input data from the system and that assemble and deliver the results of the temporal observers and the Bayesian networks must be customized for each application. Also, additional AT signal processing filters must be implemented from scratch as needed. By design, the proper R2U2 observation tree models must be set up individually for each new application; this is currently a manual specification process.

All versions of R2U2 have been developed at, or with funding support from, NASA and are currently not freely available. We are currently working on making R2U2 into an Open Source product. We anticipate that all code, models, and documentation will be freely available for download in the future. We will pave the way for transfer of our revised R2U2 framework to NASA, academia, and other industrial and research autonomous platform developers with thorough documentation and a usable and as-automated-as-possible interface. We anticipate the publication of the *R2U2 User's Guide* as a NASA Technical Memorandum in coordination with the open-source release.

3 Examples

In this section, we briefly describe some examples from published case studies that illustrate key features of R2U2.

3.1 Temporal Monitoring to Detect Sensor Failures

In [7], we developed R2U2 observation trees that are able to detect communication errors between major computer components of a UAS. The driving example was a NASA flight test where the Swift UAS was grounded for 48 hours as system engineers worked to diagnose an unexpected problem with that disrupted vital data transmissions to the ground. During flight, the data from the fluxgate magnetometer (FG), which measures strength and direction of the Earth's magnetic field, is transferred via a serial link to the Common Payload System (CPS). A subtle configuration mismatch caused internal buffer overflows, resulting in an increasing number of corrupted packets sent to the CPS. In that case study, we input to R2U2 the original data as recorded by the Swift UAS; R2U2 was able to diagnose this problem in real-time,

and could have avoided the costly delay in flight testing. Below, we show a subset of relevant temporal formulas (see also Figure 2).
S1: The FG packet transmission rate N^R_{tot} is approximately 64 per second: $63 \leq N^R_{tot} \leq 66$.
S2: The number of bad packets N^R_b is low, no more than one bad packet every 30 seconds:

$$\Box_{[0,30]}((N^R_b = 0) \vee ((N^R_b \geq 1)\ \mathcal{U}_{[0,30]}(N^R_b = 0))).$$

S3: The bad packet rate N^R_b does not appear to be increasing; we do not see a pattern of three bad packets within a short period of time:

$$\neg(\Diamond_{[0,30]}(N^R_b \geq 2) \wedge \Diamond_{[0,100]}(N^R_b \geq 3)).$$

S5: We have a subformula Eul that states if the UAS is moving (Euler rates of pitch p, roll r, and yaw y are above the tolerance thresholds $\theta = 0.05$) then the fluxgate magnetometer should also register movement above its threshold $\theta_{FG} = 0.005$. The formula states that this subformula should not be false more than three times within 100 seconds of each other.

$$Eul := ((|p| > \theta) \vee (|r| > \theta) \vee (|y| > \theta)) \rightarrow ((|FG_x| > \theta_{FG}) \vee (|FG_y| > \theta_{FG}) \vee (|FG_z| > \theta_{FG}));$$

$$\neg(\neg Eul \wedge (\Diamond_{[2,100]}(\neg Eul \wedge \Diamond_{[2,100]}\neg Eul))).$$

3.2 Temporal Monitoring of a UAS AutoPilot

In most R2U2 configurations, we distinguish three different kinds of system safety requirements [33]: value checks (V), relationship requirements (R), and flight rules (F). Below, we give a number of examples for each of these categories.
V1: The maximal safe climb and descent rate V_z of the UAS is limited by its design and engine characteristics:

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

V2: The maximal angle of attack α is limited by design characteristics as:

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

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

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

V4: The battery voltage U_{batt} and current I_{batt} must remain within certain bounds during the entire flight. Furthermore, no more than 50A should be drawn from the battery for more than 30 consecutive seconds in order to avoid battery overheating:

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

R1: Pitching up (i.e., increasing α from the current setting α_0) for a sustained period of time (more than 20 seconds) should result in a positive change in altitude, measured by a positive vertical speed V_z. This increase in vertical speed should occur within two seconds after pitch-up:

$$\Box([\Box_{[0,20s]}(\alpha > \alpha_0)] \rightarrow [\Diamond_{[0,2s]}(V_z > 0)]).$$

We can refine this relationship to only hold if the engine has enough power (as measured by the electrical current to the engine I_{eng}) to cause the aircraft to actually climb:

$$\square([\square_{[0,20s]}((\alpha > \alpha_0) \wedge (I_{eng} > 30A))] \rightarrow [\diamond_{[0,2s]}(V_z > 0)]).$$

Similarly, we can define a rule for descent:

$$\square([\square_{[0,20s]}((\alpha < \alpha_0) \vee (I_{eng} < 10A))] \rightarrow [\diamond_{[0,2s]}(V_z < 0)]).$$

R2: Whenever the UAS is in the air, its indicated airspeed (V_{IAS}) must be greater than its stall speed V_S. The UAS is considered to be air-bound when its altitude alt is larger than that of the runway alt_0:[6]

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

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

$$\square([\square_{[0,5s]}(V_z > 0)] \rightarrow [\diamond_{[0,2s]}(\Delta_{alt_b} > \theta)]).$$

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

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

F2: After receiving the landing command, touchdown needs to take place within 40 seconds, unless the link (lnk) is lost. The status of the link is denoted by s_{lnk}. In a lost-link situation, the aircraft should reach a loitering altitude around 425ft within 20 seconds:

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

F3: The default flight mode is to "stay on the move." The UAS should not loiter in one place for more than a minute unless it receives the loiter command (which may not ever happen during a mission). Let *sector_crossing* be a Boolean variable, which is true if the UAS crosses the boundary between the small subdivision of the airspace in which the UAS is currently located and another subdivision. After receiving the loiter command, the UAS should stay in the same sector, at an altitude between 400ft and 450ft until it receives a landing command. The UAS has 30 seconds to reach loitering position:

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

[6] Here, we assume that the altitude of the runway is always lower than that of the flying aircraft.

3.3 Temporal Software Health Monitoring

Due to the unobtrusive nature of its observers, R2U2 can utilize signals from both hardware and software in determining system health, combining hardware sensor values with operating-system specific signals, like memory usage, CPU load, free space in the file system, etc. For example, we can reason about temporal properties on a message-based embedded software system with logging to an on-board file system. In contrast to the requirements stated above, this flight rule specifically concerns properties of the flight software:
SW1: All messages sent from the guidance, navigation, and control (GN&C) component to the actuators must be logged into the on-board file system (FS). Logging has to occur before the message is removed from the queue.

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

3.4 Root-cause Analysis with R2U2 Bayesian Networks

Because temporal logic observers alone often cannot uniquely identify faults, we combine them with Bayesian networks for disambiguation and root-cause analysis.

3.4.1 Disambiguation of Flight Computer Failures

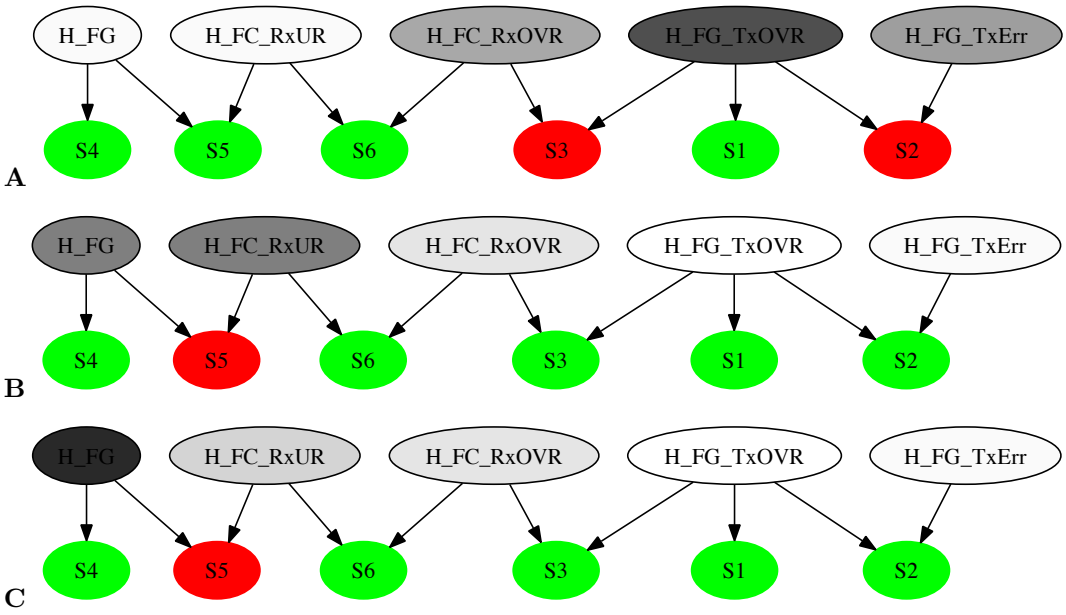

Figure 7: [7] Posterior probabilities of the health nodes for different fluxgate magnetometer failure conditions in the observation tree from Figure 2.

The six temporal observers $S_1, \ldots, S_6$ discussed in Figure 2 and Section 3.1 efficiently diagnose symptoms of a faulty fluxgate magnetometer. These observers alone are not sufficient to fully pinpoint the actual failure; different failures could trigger these temporal observers in different combinations. R2U2 therefore combines them with a Bayesian network to do root

cause analysis and to disambiguate failure modes. In Figure 7, the nodes in the lower row receive, as evidence, the results of each specification S_i (see Section 3.1). The Bayesian network produces posterior marginals of the health nodes for the various failure modes. When exercised under different failure scenarios, a clear distinction of the root causes can be obtained in most cases (Figure 7A,C). Figure 7B, shows, however, that no distinction can be made to distinguish a fault in the sensor itself (H_FG) from a receiver underrun error in the flight computer (H_FC_RxUR). A disambiguation of these failure modes can be easily achieved by the use of priors in the Bayesian network: a sensor failure is more likely to occur than that specific transmission failure.

3.4.2 Disambiguation of Sensor Failures

Maintaining an accurate measurement of the altitude of an aircraft is absolutely essential as sensor failures can lead to catastrophic crashes. In a UAS, where no redundant sensors are available, R2U2 can perform diagnostic reasoning based on sensors that measure related effects. Figure 8A shows an R2U2 Bayesian network that estimates the health of a barometric altimeter, which measures the altitude above sea-level, and a laser altimeter, which measures the altitude above ground. As inputs, we do not use the actual measurements but the trends (going-up, going-down) and also use (noisy) information from the inertial navigation unit about the sign of the vertical aircraft speed S_S. This figure also shows the conditional probability tables (CPT), including priors on the lower reliability of the laser altimeter as compared to the barometer. Figure 8B shows actual flight data from a flight, where the laser altimeter (brown line) failed. The lower panel shows the health of the barometric altimeter (blue) and the laser altimeter (brown) as determined by R2U2 using a combination of temporal logic observers and the Bayesian network in Figure 8A.

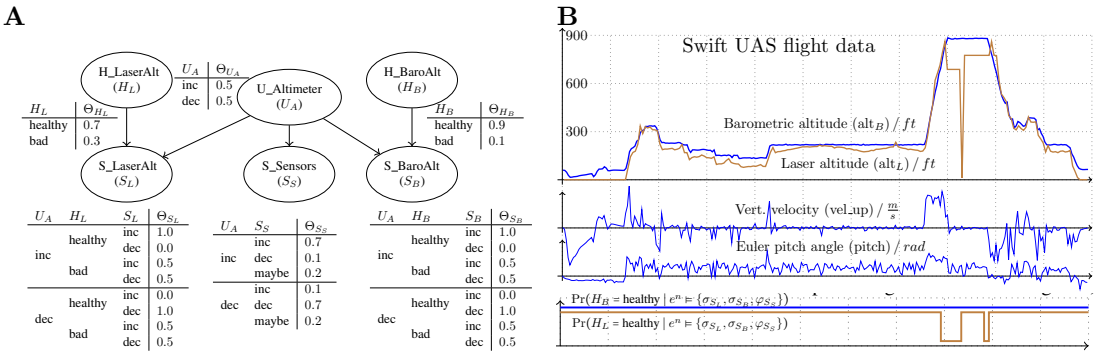

Figure 8: [19] **A:** Bayesian network for altimeter health; **B:** Flight data of the NASA Swift

4 Applications of R2U2

R2U2 has been both used in case studies and deployed in real-world applications for UAS, from the 13-foot wingspan all-electric Swift UAS to the 3.75-foot wingspan DragonEye to the S1000 octocopter. Table 1 gives an overview of the major applications. In 2016 we began branching out into space applications. In the future, we plan to deploy R2U2 on a host of other platforms, including rovers, satellites, landers, cockpit displays, and other robotic and human-assistive systems, in both air and space.

Domain	Properties	Platform(s)	Description	Ref
UAS	sensor	NASA Swift UAS	altimeter data	[19]
UAS	payload	NASA Swift UAS	fluxgate magnetometer data	[7]
UAS	autopilot	NASA DragonEye	mixed sensor data	[32, 33]
UAS	variety	NASA Swift+DragonEye	mixed sensor data	[24]
UAS	prognostics		battery prognostics	[31]
UAS	security	UAS simulation	GPS spoofing, hijacking	[29]
UAS	tool demo	UAS simulation	general demo, GPS spoofing	[30]
Space	autonomy	small satellites, landers	sanity checks facilitating autonomous operation, specifications for space systems	[22]
UAS	autonomy	S1000 octocopter	general SHM for NASA AOS	[11]

Table 1: Applications of R2U2

5 Conclusions

R2U2 fills a unique gap in the field of runtime verification. By coming at the problem from the system's perspective and asking, what questions do we need answered to manage this system's health, and how can we address them within this system's constraints (both physical and regulatory), R2U2 addresses an outstanding need for on-board analysis. Our perspective contrasts with more traditional verification paradigms that carry design-time specifications for individual components through to runtime, e.g., by instrumenting software to add code annotations designating how specific code blocks should behave, or by translating design-time architecture specifications in LTL to automata-based runtime monitors.

We arrive at an adaptive and extensible specification formalism of an observation tree to address each question about runtime system health. The compositional building blocks that make up R2U2's observation trees are inspired by design-time-to-runtime technologies for efficiently translating different types of specifications into real-time checks. We expect the efficacy and utility of R2U2's observation trees to continue to grow as these technologies advance and we add more building blocks. R2U2 stands at the forefront of runtime verification for cyber-physical systems as observation trees implement specifications that seamlessly mix inputs from both hardware and software in a way that is amenable to flight certification.

5.1 Future Work

We are currently investigating embedding R2U2 into a wider variety of platforms, with a particular focus on space applications: SmallSats, rovers, robotic platforms, and autonomous spacecraft. These present different challenges than the UAS platforms we have investigated previously from many perspectives, from low-level details such as the implementation platforms, to high-level questions of what system health management means for each such system.

Another frontier for R2U2 is integration into larger avionics systems. We are investigating integration of R2U2 on-board reasoning with automated subsystems for NextGen automated air traffic control to enable reasoning about compliance with air traffic regulations on-board. Furthermore, we plan to build on the real-time adaptability of the R2U2 framework to enhance dynamic reasoning capabilities in glass cockpits and intelligent co-pilot systems.

To enable more optimized system health management over a variety of platforms, we plan to rigorously evaluate algorithmic and implementation trade-offs. For example, in the current MTL encoding, we translate formulas into a normal form before synthesizing an R2U2 configuration to check if they are satisfied during runtime; for LTL satisfiability checking, we previously showed that formula translation has a substantial influence on performance [26], and that we can

achieve significant, up to exponentially better performance, through better formula encoding [25]. We plan to investigate whether similar results are possible for runtime encodings of MTL. Other areas for investigation include functional formula patterns [23] for VHDL, hardware implementation choices such as queue configurations, and a rigorous experimental evaluation of the hardware and software implementations to quantify the space for better specialization of R2U2 configurations on future systems.

6 Acknowledgments

Many colleagues and students have contributed to the design, implementation, testing, and applications of R2U2 on a variety of platforms over the years since its inception in 2013. We thank the following scientists for their contributions: Bijan Choobineh, Johannes Geist, Corey Ippolito, Stefan Jaksic, Phillip Jones, Chetan S. Kulkarni, Eddy Mazmanian, Patrick Moosbrugger, Quoc-Sang Phan, Thomas Reinbacher, Indranil Roychoudhury, Iyal Suresh, Joseph Zambreno, Pei Zhang.

R2U2, along with its different versions, extensions, applications, and integration interfaces with a variety of other tools and platforms have been funded by the following grants. We thank these programs for their support:

2014–2015 NASA Aeronautics Research Institute (NARI) Seedling Phase I Grant *Intelligent Hardware-Enabled Sensor and Software Safety and Health Management for Autonomous UAS* (PI: Rozier; Co-I: Schumann)

2015–2018 NASA NNX14AN61A *NASA Autonomy Operating System (AOS) for UAVs* (Co-Is: Rozier, Schumann)

2016–2021 NSF CAREER Award CNS-1552934 *CAREER: Theoretical Foundations of the UAS in the NAS Problem (Unmanned Aerial Systems in the National Air Space* (PI: Rozier)

2016–2019 NASA ECF NNX16AR57G *Multi-Platform, Multi-Architecture Runtime Verification of Autonomous Space Systems* (PI: Rozier)

References

[1] Adapteva. The parallella board. https://www.parallella.org/board.

[2] National Aeronautics and Space Administration (NASA). Livingstone2 software. http://ti.arc.nasa.gov/opensource/projects/livingstone2/, 2007. open source.

[3] R. Anderson, P. Beame, S. Burns, W. Chan, F. Modugno, D. Notkin, and J. D. Reese. Model checking large software specifications. *IEEE Transactions on Software Engineering*, 24:156–166, 1996.

[4] M. G. Ballin, W. Cotton, and P. Kopardekar. Share the sky: Concepts and technologies that will shape future airspace use. In *11th AIAA Aviation Technology, Integration, and Operations (ATIO) Conference, including the AIAA Balloon Systems Conference and 19th AIAA Lighter-Than*, page 6864, 2011.

[5] A. Darwiche. *Modeling and Reasoning with Bayesian Networks*. Cambridge University Press, 1st edition, 2009.

[6] M. Gario, A. Cimatti, C. Mattarei, S. Tonetta, and K. Y. Rozier. Model checking at scale: Automated air traffic control design space exploration. In *Proceedings of 28th International Conference on Computer Aided Verification (CAV 2016)*, volume 9780 of *LNCS*, pages 3–22. Springer, 2016.

[7] J. Geist, K. Y. Rozier, and J. Schumann. Runtime Observer Pairs and Bayesian Network Reasoners On-board FPGAs: Flight-Certifiable System Health Management for Embedded Systems. In

Proceedings of the 14th International Conference on Runtime Verification (RV14), volume 8734, pages 215–230. Springer-Verlag, September 2014.

[8] A. Groce, K. Havelund, G. Holzmann, R. Joshi, and RG. Xu. Establishing flight software reliability: Testing, model checking, constraint-solving, monitoring and learning. *Annals of Mathematics and Artificial Intelligence*, 70(4):315–349, 2014.

[9] D. Hall, S. Narasimhan, L. Brownston, A. Patterson-Hine, K. Goebel, and S. Poll. The HyDE diagnostics system. http://ti.arc.nasa.gov/tech/dash/diagnostics-and-prognostics/hyde-diagnostics/.

[10] L. Howard. An Algorithm for Diagnostic Reasoning Using TFPG Models in Embedded Real-Time Applications. In *Proc. AUTOTESTCON 2001*, pages 978–987, 2001.

[11] M. Lowry, A. Bajwa, P. Quach, G. Karsai, K. Y. Rozier, and S. Rayadurgam. Autonomy Operating System for UAVs. Online: https://nari.arc.nasa.gov/sites/default/files/attachments/15%29%20Mike%20Lowry%20SAEApril19-2017.Final_.pdf, April 2017.

[12] C. Mattarei, A. Cimatti, M. Gario, S. Tonetta, and K. Y. Rozier. Comparing different functional allocations in automated air traffic control design. In *Proceedings of Formal Methods in Computer-Aided Design (FMCAD 2015)*. IEEE/ACM, 2015.

[13] C. Muñoz, V. Carreño, and G. Dowek. Formal analysis of the operational concept for the Small Aircraft Transportation System. In *Rigorous Engineering of Fault-Tolerant Systems*, volume 4157 of *LNCS*, pages 306–325. Springer, 2006.

[14] QSi. Teams designer. http://www.teamqsi.com/products/teams-designer/.

[15] Radio Technical Commission for Aeronautics (RTCA). DO-178B: Software Considerations in Airborne Systems and Equipment Certification, 1992.

[16] Radio Technical Commission for Aeronautics (RTCA). DO-254: Design Assurance Guidance for Airborne Electronic Hardware, April 2000.

[17] Radio Technical Commission for Aeronautics (RTCA). DO-333: Formal Methods Supplement to DO-178C and DO-278A. Technical report, December 2011.

[18] Radio Technical Commission for Aeronautics (RTCA). DO-178C/ED-12C: Software Considerations in Airborne Systems and Equipment Certification, 2012.

[19] T. Reinbacher, K. Y. Rozier, and J. Schumann. Temporal-logic based runtime observer pairs for system health management of real-time systems. In *Proceedings of the 20th International Conference on Tools and Algorithms for the Construction and Analysis of Systems (TACAS)*, volume 8413 of *Lecture Notes in Computer Science (LNCS)*, pages 357–372. Springer-Verlag, April 2014.

[20] Kristin Y. Rozier Rohit Dureja, Eric W. D. Rozier. A case study in safety, security, and availability of wireless-enabled aircraft communication networks. In *Proceedings of te 17th AIAA Aviation Technology, Integration, and Operations Conference (AVIATION)*. American Institute of Aeronautics and Astronautics, June 2017.

[21] K. Y. Rozier. Linear Temporal Logic Symbolic Model Checking. *Computer Science Review Journal*, 5(2):163–203, May 2011.

[22] K. Y. Rozier. R2U2 in Space: System and Software Health Management for Small Satellites. In *Spacecraft Flight Software Workshop (FSW)*, December 2016. https://www.youtube.com/watch?v=OAgQFuEGSi8.

[23] K. Y. Rozier. Specification: The biggest bottleneck in formal methods and autonomy. In *Proceedings of 8th Working Conference on Verified Software: Theories, Tools, and Experiments (VSTTE 2016)*, volume 9971 of *LNCS*, pages 1–19, Toronto, ON, Canada, July 2016. Springer-Verlag.

[24] K. Y. Rozier, J. Schumann, and C. Ippolito. Intelligent Hardware-Enabled Sensor and Software Safety and Health Management for Autonomous UAS. Technical Memorandum NASA/TM-2015-218817, NASA, NASA Ames Research Center, Moffett Field, CA 94035, USA, May 2015.

[25] K. Y. Rozier and M. Y. Vardi. A multi-encoding approach for LTL symbolic satisfiability checking.

In *17th International Symposium on Formal Methods (FM2011)*, volume 6664 of *Lecture Notes in Computer Science (LNCS)*, pages 417–431. Springer-Verlag, 2011.

[26] K.Y. Rozier and M.Y. Vardi. LTL satisfiability checking. *International Journal on Software Tools for Technology Transfer (STTT)*, 12(2):123–137, March 2010.

[27] J. Schumann, T. Mbaya, O. Mengshoel, K. Pipatsrisawat, A. Srivastava, A. Choi, and A. Darwiche. Software health management with Bayesian Networks. *Innovations in Systems and Software Engineering*, 9(4):219–233, 2013.

[28] J. Schumann and P. Moosbrugger. Unobtrusive Software and System Health Management with R2U2 on a parallel MIMD Coprocessor. In *Proceedings of the 2017 Annual Conference of the Prognostics and Health Management Society (PHM2017)*, 2017.

[29] J. Schumann, P. Moosbrugger, and K. Y. Rozier. R2U2: Monitoring and Diagnosis of Security Threats for Unmanned Aerial Systems. In *Proceedings of the 15th International Conference on Runtime Verification (RV15)*. Springer-Verlag, September 2015.

[30] J. Schumann, P. Moosbrugger, and K. Y. Rozier. Runtime analysis with R2U2: A tool exhibition report. In *Proceedings of the 16th International Conference on Runtime Verification (RV16)*, pages 504–509, 2016.

[31] J. Schumann, I. Roychoudhury, and C. Kulkarni. Diagnostic reasoning using prognostic information for unmanned aerial systems. In *Proceedings of the 2015 Annual Conference of the Prognostics and Health Management Society (PHM2015)*, 2015.

[32] J. Schumann, K. Y. Rozier, T. Reinbacher, O. J. Mengshoel, T. Mbaya, and C. Ippolito. Towards real-time, on-board, hardware-supported sensor and software health management for unmanned aerial systems. In *Proceedings of the 2013 Annual Conference of the Prognostics and Health Management Society (PHM2013)*, pages 381–401, October 2013.

[33] J. Schumann, K. Y. Rozier, T. Reinbacher, O. J. Mengshoel, T. Mbaya, and C. Ippolito. Towards real-time, on-board, hardware-supported sensor and software health management for unmanned aerial systems. *International Journal of Prognostics and Health Management (IJPHM)*, 6(1):1–27, June 2015.

[34] T. Sreemani and J. M. Atlee. Feasibility of model checking software requirements: A case study. In *Proc. Conference on Computer Assurance (COMPASS)*, pages 77–88. IEEE, 1996.

[35] M. Whalen, S. Rayadurgam, E. Ghassabani, A. Murugesan, O. Sokolsky, M. Heimdahl, and I. Lee. Hierarchical multi-formalism proofs of cyber-physical systems. In *Proc. Conference on Formal Methods and Models for System Design (MEMOCODE)*, pages 90–95. IEEE, 2015.

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