Innovations in Systems and Software Engineering (2022) 18:567–580
https://doi.org/10.1007/s11334-021-00407-5

S.I. : DETECT 2020

Integrating runtime verification into an automated UAS traffic management system

Abigail Hammer[1] · Matthew Cauwels[1] · Benjamin Hertz[1] · Phillip H. Jones[1] · Kristin Y. Rozier[1]

Received: 15 December 2020 / Accepted: 21 June 2021 / Published online: 14 July 2021

Abstract
Unmanned Aerial Systems (UAS) are quickly integrating into the National Air Space. Doing so safely is a pressing concern, as the US alone has over 1.5 million registered small (under 55 pounds) UAS and the FAA projects further rapid expansion. This drives the need for an intelligent, automated system for UAS Traffic Management (UTM). Even more than for manned aircraft, UTM must integrate runtime checks to ensure system safety, at the very least to make up for the lack of humans on-board to employ the common-sense safety checks ingrained into the culture of human aviation. We overview a candidate automated, intelligent UTM system and propose multiple integration points for runtime verification to ensure that each part of the UTM adheres to safety requirements during operation. We write, validate, and present patterns for formal requirements across multiple subsystems of this UTM framework. We incorporate specifications that use set aggregation as a way of raising their abstraction from single sensors to sets of sensors, which allow us to monitor for system requirement violations with smaller specifications. After encoding our requirements as flight-certifiable runtime observers in the R2U2 RV engine, we execute them in simulation across multiple real-life test flights supplemented with simulated data to cover additional cases that did not occur in flight. Lessons learned accompany an analysis of the efficacy and performance of RV integration into the UTM framework.

Keywords UAS · UTM · Runtime verification · R2U2

1 Introduction

The Federal Aviation Administration (FAA) forecasts Unmanned Aerial System (UAS) numbers to continue to "expand rapidly" over the next 20 years with over 90% of the growth from consumer-grade or professional-grade (non -model) UAS used for commercial or research purposes [9]. Given the considerable traffic this will generate and the pressing concern for safe integration into the National Air Space (NAS), additional traffic management is required on top of current safety regulations [10,18,23]. A recent candidate for an intelligent, automated UAS Traffic Management (UTM) system addresses these concerns [34].

One important consideration in such an automated system is how, and where, to integrate checks *during system operation* that continuously monitor for violations of system safety requirements, e.g., due to unexpected environmental conditions or other scenarios that could not be predicted and tested for during system design. This is especially critical given the automated nature of the systems involved: pilots and human ground controllers make numerous decisions in the control of commercial aircraft that serve as a foundation for their traffic management systems but are missing from UTM. For example, pilots regularly identify and dismiss off-nominal sensor readings and ground controllers operate

Supported by NSF CAREER Award CNS-1552934, NSF PFI:BIC grant CNS-1257011, and NASA Grant # 80NSSC21M0121.
Reproducibility artifacts: http://temporallogic.org/research/ISSE2021/.

✉ Abigail Hammer
arhammer@iastate.edu

Matthew Cauwels
mcauwels@iastate.edu

Benjamin Hertz
benhertz@iastate.edu

Phillip H. Jones
phjones@iastate.edu

Kristin Y. Rozier
kyrozier@iastate.edu

1 Iowa State University, Ames, USA

under unstated assumptions, such as that the flight plans of two aircraft should never contain unsafe overlaps.

Runtime Verification (RV) provides checks that cyber-physical systems adhere to their safety requirements during operation. However, much of the research into RV has focused on increasing expressivity of monitored properties and operational reach of RV engines. One example of this is incorporating first-order logic into a variety of temporal logic languages [4,8,11,14]. In any of these implementations, expressiveness comes with a tradeoff in complexity [5–7,11,13]. The on-board resources, overhead, operational delays, and intrusive system instrumentation required to run these tools are incompatible with flight certification [32]. In response, the Realizable, Responsive, Unobtrusive Unit (R2U2) was designed to monitor sufficiently expressive properties in real time, under hard resource constraints, with low-to-no overhead, and without system instrumentation that would violate flight certification [24]. Only three RV tools are flight-certifiable: R2U2, Lola [30], and Co-Pilot [22]; R2U2's flexible architecture was the easiest to adapt to our UTM system.

We examine the candidate UTM system [34], overviewing its design, implementation, and initial tests, e.g., with University of Iowa's (U of I's) Operational Performance Laboratory's (OPL's) Vapor 55 UAS flying over a small, nearby airspace. We map out three subsystems where RV could be embedded within this UTM framework: on-board the Vapor 55, on-board each Ground Control Station (GCS), and within the UTM's cloud-based framework. However, the big-gest bottleneck to the successful deployment of formal methods, like RV, is specification of the requirements under verification [26]. Building upon the runtime specification pattern categories of [26], we detail patterns for formal requirements specification across these subsystems and write, debug, and validate a covering set of temporal logic specifications.

Additionally, we detail a new syntactic extension for Mission-time Linear Temporal Logic that we call MLTL with set aggregation. In essence, we incorporate new syntax that allows subsets of atomic propositions to be grouped into a single atomic proposition for the set. Using these operators raises the level of abstraction of a specification from an individual signal to a set of signals, e.g., sets of UAS or flight plans, enabling easier validation for certain common specifications by reducing specification length and complexity, and enabling the underlying RV engine to employ optimizations when creating an observer for the specification. For example, if the intention of a specification is to trigger a "mode confusion" alert if a UAS ever has two flight modes enabled at the same time, we can use set aggregation syntax to signal to the RV engine that we do not care *which two* members of the set of modes are enabled, allowing for automated implementation optimizations taking advantage of lossy information strategies.

Using R2U2 to create runtime observers from this specification set, we deploy in simulation real-time RV over a set of real-life flight tests, expanding our data set to include realistic scenarios that were not able to be flown in real life. We examine the outputs from R2U2 and provide a roadmap for utilizing this data to robustify the UTM framework. Our case study details the process of RV integration for future adopters of systems like UTM.

Our contributions are as follows: (1) patterns useful for RV specifications across a real distributed UTM implementation; (2) introduce new syntax designating set aggregation in MLTL formulas that maintains context to be used in monitor optimizations; (3) an open set of RV benchmarks from real-world UAS/GCS telemetry data; (4) an extensive experimental evaluation (124 specifications) of a distributed RV implementation in real-time; and (5) lessons learned from distributed RV specifications validation and refinement for a UTM system.

The remainder of this paper is organized as follows. Section 2 gives background information on MLTL and R2U2. Section 3 overviews the candidate UTM framework. Our formal specifications fill Sect. 4, including specifications specific to the on-board UAS, the GCS, and the UTM's cloud-based framework. To inform future practitioners, we detail their organization, discuss coverage metrics, and exemplify each specification pattern we found useful in our study. We also formally present MLTL with set aggregation syntax and address the critical topic of specification validation and debugging. Section 5 describes our test scenario, evaluates the efficiency of MLTL with set aggregation, and graphs the outputs from R2U2 for specifications from six of our patterns. Section 6 concludes with lessons learned and next steps for RV integration into the future UTM system.

2 Preliminaries

2.1 Mission-time linear temporal logic

For all our specifications, our chosen language is Mission-time Linear Temporal Logic (MLTL) [17,24]. MLTL incorporates a closed interval over naturals $I = [a, b] (0 \leq a \leq b$ are natural numbers) time bounds over a set of bounded natural numbers on each temporal operator. Unlike Metric Temporal Logic (MTL) [2], open or half-open intervals over the natural domain are not necessary, as open and half-open intervals can be reduced to an equivalent closed bounded interval e.g., $(1, 2) = \emptyset$, $(1, 3) = [2, 2]$, $(1, 3] = [2, 3]$, etc [17].

Definition 1 (MLTL Syntax [17,24]) The syntax of an MLTL formula ϕ over a set of atomic propositions $\mathcal{AP}$ is recursively defined as:

$\phi ::= \mathsf{true} \mid \mathsf{false} \mid p \mid \neg\phi \mid \phi_1 \wedge \phi_2 \mid \phi_1 \vee \phi_2 \mid$
$\Box_I \phi \mid \Diamond_I \phi \mid \phi_1 \mathcal{U}_I \phi_2 \mid \phi_1 \mathcal{R}_I \phi_2$

where $p \in \mathcal{AP}$ is a Boolean atom (0/1), ϕ_1 and ϕ_2 are MLTL formulas, and I is a closed-bound interval $[lb, ub]$, where $lb \leq ub$.

For any two MLTL formulas ϕ_1 and ϕ_2, $\phi_1 \equiv \phi_2$ if and only if they are semantically equivalent. Since MLTL is derived from linear temporal logic (LTL), many of the semantics are the same: $\mathsf{false} \equiv \neg\mathsf{true}$, $\phi_1 \vee \phi_2 \equiv \neg(\neg\phi_1 \wedge \neg\phi_2)$, $\neg(\phi_1 \mathcal{U}_I \phi_2) \equiv (\neg\phi_1 \mathcal{R}_I \neg\phi_2)$ and $\neg\Diamond_I\phi \equiv \Box_I\neg\phi$. MLTL keeps the standard operator equivalences from LTL as well, including $(\Diamond_I\phi) \equiv (\mathsf{true}\mathcal{U}_I\phi)$, $(\Box_I\phi) \equiv (\mathsf{false}\mathcal{R}_I\phi)$, and $(\phi_1\mathcal{R}_I\phi_2) \equiv (\neg(\neg\phi_1\mathcal{U}_I\neg\phi_2))$. The only notable difference is that MLTL discards LTL's next ($\mathcal{X}$) operator, as it is semantically equivalent to $\Box_{[1,1]}\phi$ [17]. A position $\pi[i]$ in a trace π, where $(i \geq 0)$ is an assignment over $2^{\mathcal{AP}}$; $|\pi|$ represents the length of π.

Definition 2 (MLTL Semantics [17,24]) The satisfaction of an MLTL formula ϕ, over a set of propositions $\mathcal{AP}$, by a computation/trace π starting from position i (denoted as $\pi, i \models \phi$) is recursively defined as:

- $\pi, i \models p$ iff $p \in \pi[i]$,
- $\pi, i \models \neg\phi$ iff $\pi, i \not\models \phi$,
- $\pi, i \models \phi_1 \wedge \phi_2$ iff $\pi, i \models \phi_1$ and $\pi, i \models \phi_2$,
- $\pi, i \models \phi_1 \mathcal{U}_{[lb,ub]} \phi_2$ iff $|\pi| > lb$ and, there exists $i \in [lb, ub]$, $i < |\pi|$ such that $\pi, i \models \phi_2$ and for every $j \in [lb, ub]$, $j < i$ it holds that $\pi, j \models \phi_1$.

2.2 Realizable responsive unobtrusive unit (R2U2)

Our R2U2 instrumentation uses two of that tool's main architectural layers: (1) signal processing and (2) temporal logic monitors. R2U2 has implementations in hardware (FPGAs), C++, and C; we choose the latter for embedding in the UTM. R2U2's architecture details appear in a tool overview [27], with additional details from past case studies in [12,16,19,24,28,31].

R2U2 reads relevant sensor readings off the main system bus, then passes them through lightweight, real-time atomic checkers that filter and discretize the sensor readings. Checks like "altitude > 1,000 ft" transform signals into Boolean atomics, i.e., true or false, that populate the atomic propositions in temporal logic formulas. Each MLTL formula encodes directly into an observer embedded on the target platform. The hierarchical tree of inputs, filters, atomic checkers, and temporal logic formulas comprise an *R2U2 specification observation tree*, an example of which can be seen in Sect. 5. Redundant branches of the tree can be combined through a pre-flight optimization step for efficiency and to reduce encoding size.

R2U2 encodes specifications in a variety of temporal logical syntaxes (e.g., LTL, MTL, MLTL) which are then compiled and constructed into one or more observation trees, which allow for re-use of similar sub-formulas within separate specifications. For example, suppose R2U2 is implemented on a fixed-wing UAS and has two separate specifications: (1) the UAS's landing gear will be stowed when it is above 1,000 ft, and (2) the UAS's speed will be within 300mph to 400mph when above 1,000 ft. Since both of these specifications require the altimeter reading to exceed 1,000 ft, a single Boolean operator can be passed to both temporal logic observers.

3 UTM system definition

In parallel with NASA's third UTM Technical Capability Level [20], a hybrid university-industrial team proposed an intelligent, centralized UTM for low-altitude urban environments to coordinate UAS traffic in a safe and efficient way [34]. A high-level diagram of the proposed UTM system appears in Fig. 1.

Ground Control Stations (GCS) connect to the UTM Cloud Server and upload their proposed flight plan for approval. The UTM Cloud server performs pre-flight plan conflict detection using a dynamic geofencing algorithm [36]. The UTM then notifies the GCS if the flight plan is rejected or approved. If rejected, the GCS should submit a new flight plan until one is approved. When approved, the GCS streams the UAS's telemetry data to the server, which then performs an en route conflict prediction. If an en route conflict is predicted, the server will alert all GCS involved in the conflict, so that they may have enough time to submit a new flight plan and perform an avoidance maneuver.

There are many challenges to overcome before such a UTM would be incorporated into the NAS [3,25]. For example, an ongoing research question is how to handle uncooperative and hostile UAS in the UTM's airspace. One assumption of this UTM is that all UAS are non-hostile, i.e., no UAS is purposefully flying an unapproved flight plan. However, this UTM was designed to receive telemetry data from anyone who connects to it, regardless of flight plan status. While the details of how to maintain communication with both cooperative and uncooperative UAS are still ongoing research [33], RV can be used within this UTM to alert the operator to the presence of uncooperative UAS.

Another ongoing research question for UTMs is whether low-altitude airspace should be structured, e.g., with similar traffic patterns and rules as ground transportation [15]. Regardless of which approach is used, RV can be incorporated to alert users of dangerous or undesirable circumstances. For example, this UTM was developed for unstructured airspace, so it has more general operating range

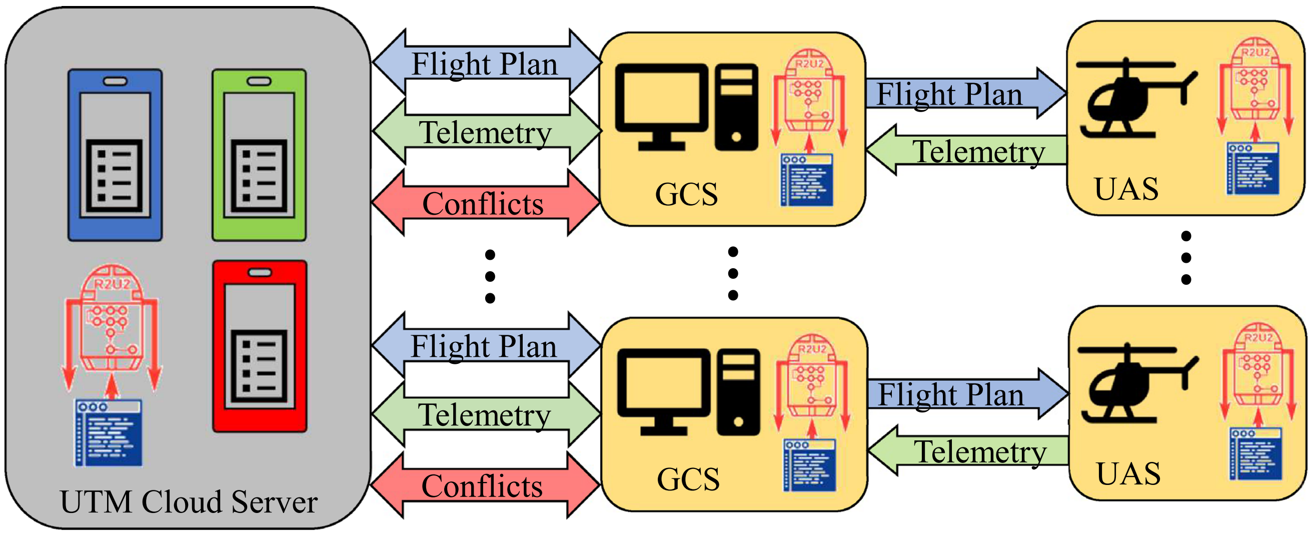


Fig. 1 An overview of the NSF funded cloud-based UTM [34]

Table 1 Selected output signals from the UAS

Signal	Description	Units
`Pos{N,E,D}`	Relative positional vector (North, East, Downward) from the home point.	{m, m, m}
`Lat, Lon, Alt`	GPS coordinate positions.	{DD, DD, MSL}
`Roll, Pitch, Yaw`	Euler angles of the UAS.	{deg, deg, deg}
`P, Q, R`	Euler angle-rates of the UAS.	{deg/s, deg/s, deg/s}
`Vel{N,E,D}`	Velocity vector of the UAS.	{m/s, m/s, m/s}
`Acc{N,E,D}`	Acceleration vector of the UAS.	{m/s^2, m/s^2, m/s^2}
`Temp, TempE{1,2}`	Temperature of the air and motors.	C
`Pres`	Atmospheric pressure.	hPa
`Phase`	Set of strings corresponding to preset phases of flight.	{<undefined>, Test actuators, Stationary, Hover, Cruise, Go to, Stop at, In flight, Landed}
`Subphase`	Set of strings corresponding to preset subphases of flight.	{Ready, Test, Takeoff, Manual, Waypoints, Home, Landing}
`FlightMode`	Set of strings corresponding to automatic and manual control.	{Automatic, Home}
`RPM`	RPM of the main motor.	–

Note that the Units of DD stands for Decimal Degrees and MSL stands for Mean Sea Level: a way to measure altitude

specifications, such as those that make sure that all UAS are within the UTM's airspace. Conversely, if a structured airspace was chosen, the structured ruleset can be formally verified using RV.

4 UTM runtime specifications

We first present the types of interfaces to each UTM subsystem, and follow with a formal definition of MLTL with set aggregation and highlight its usefulness within the GCS and UTM. Then we show how R2U2 can be implemented into each subsystem, to drive specification elicitation[1]. We conclude with a discussion of the techniques used to validate our specifications.

4.1 UTM Sub-system I/O

UAS The UAS follows a flight plan provided by the GCS and is responsible for collecting and streaming its telemetry data to the GCS. Real flight data from OPL's Vapor 55 UAS helicopter's [1] internal log provides the data used for anal-

[1] Note that the list presented is not a comprehensive list of all our specifications; the full list can be found at http://temporallogic.org/research/DETECT2020/.

ysis and evaluation. The subset we chose is based on which signals were most useful for performing RV; see Table 1.

For each UAS in the system, the number of inputs to an on-board R2U2 implementation remains constant over the entire run and is predetermined prior to runtime. This makes implementations of R2U2 equivalent across all UAS, meaning that the time spent creating specifications for an individual UAS remains constant. This is assuming all UAS in a system are the same class, e.g., all single-rotor helicopter-style UAS with similar parameters.

GCS The GCS has many responsibilities within the UTM system. It is responsible for: (1) submitting flight plans to the UTM; (2) directing and receiving telemetry data from an inflight UAS; (3) pre-processing and transmitting any telemetry data received from its UAS to the UTM; and (4) monitoring for any conflict alerts from the UTM. For our case study, we look only at implementing RV to monitor (1), (2), and (4). Due to limitations on the way the UTM's test data was produced, i.e., the Vapor 55 was only simulated during the UTM test, and because it would be identical to the UAS's R2U2 implementation, we omitted (3) from the GCS's R2U2 implementation.

A challenging aspect of the GCS is that the flight plan data must be continuously streamed to R2U2, since flight plans that are transmitted from the GCS to the UTM are not saved anywhere in R2U2's memory (see Table 2). This made formatting R2U2's inputs from the GCS challenging; in particular, the number of waypoints within a GCS's flight plan can vary. This led to Equation (1)'s total inputs from a GCS to R2U2:

$$NumTelem + NumFP + (NumWPsFP)(NumWP) \tag{1}$$

where *NumTelem* is the number of telemetry inputs, *NumFP* is the number of inputs from the flight plan, *NumWpsFP* is the number of signals associated with each waypoint, and *NumWP* is the number of waypoints within the flight plan. For our specific system, $NumTelem = 9$, $NumFP = 4$, $NumWpsFP = 5$, and *NumWP* varies between 3 and 10 waypoints.

This variance in the number of inputs from one GCS to another led us to develop specifications that validate across *all* instances of *NumWP*. We accomplished this by adjusting R2U2's pre-processing layer to iterate across a loop of all instances of one variable (say, `Phase`) and determine if at least one violates a certain property. In essence, this leads to a mapping of multiple input signals to a single Boolean atomic for R2U2's temporal logic monitors. Note that this is not a first-order logic extension of MLTL; rather, we found that this modification allowed us to reason over sets of signals in specifications while maintaining the light-weight and simple syntax of MLTL. This process, termed MLTL with set aggregation, is defined further in Sect. 4.2.

UTM Cloud Server Since the UTM is implemented as a cloud-based, centralized server, it is in charge of consolidated all transmitted data, comprised of flight plans, telemetry information, the states the UAS are in, and determining whether any two UAS will conflict. Like the instances of R2U2 for the GCS, the number of inputs for the UTM varies: once with the number of waypoints in a flight plan and again with the number of UAS. Thus, the total number of inputs to an instance of R2U2 for the UTM can be calculated by Equation (2):

$$NumID(NumTelem + NumFP) + \\ (NumWPsFP)(\sum_{i=0}^{NumID} NumWP[i]) \tag{2}$$

where *NumID* represents the total number of flight plan IDs in the UTM and $NumWP[i]$ is the specific number of waypoints for flight plan i. This can lead to a large number of inputs for R2U2, e.g., 20 UAS with 4 waypoints each would be 580 inputs.

Similar to the GCS, to get traction on such a large number of input signals, we utilized MLTL with set aggregation to develop specifications that iterate across all instances of UAS within the UTM. Again, we trade expressiveness for performance: we retain real-time performance guarantees but only promise R2U2 will immediately alert the UTM of a violation; it will not identify the specific UAS responsible.

4.2 MLTL with set aggregation

We express a subset of our GCS and UTM specifications using a modified version of MLTL, which we term MLTL with set aggregation. Ultimately, this abstraction allows us to conjunct (or disjunct) a set of atomic propositions together, prior to any temporal operator, to allow us to express specifications across sets of inputs in a simple and efficient manner. Using set aggregation, while not increasing the expressibility, preserves contexts for monitor optimizations, while providing a compact and efficient way to conjunct or disjunct specification together. While the definition of MLTL with set aggregation doesn't change the original semantics and presents itself as syntactic sugar, we find that there are benefits to the use of set aggregation. The syntax for our MLTL with set aggregation is as follows:

Definition 3 (MLTL w/ Set Aggregation Syntax [17,24]) The syntax of an MLTL formula ϕ over a set of atomic proposi-

Table 2 Input and output signals from the GCS to the UTM

Signal	GCS I/O	Description	Units
Telemetry signals			
`ID`	O	The flight plan ID of the telemetry transmission	Int
`Time`	O	The time stamp when the GCS transmits the telemetry to the UTM	UNIX
`wp{Lon,Lat,Alt}`	O	The longitude, latitude, and altitude of the waypoint the UAS is currently flying toward	DD/MSL
`Lon, Lat, Alt`	O	The UAS's measured longitude, latitude, and altitude	DD/MSL
`Vel`	O	The UAS's velocity measurement	m/s
`Ang`	O	The UAS's heading measurement	deg.
Flight Plan Signals			
`fp_ID`	I	The UTM's assigned flight plan ID for the approved flight plan	Int
`Status`	I	The UTM's response to the GCS's flight plan	{Approved, Rejected, Replaced}
`Start`	O	The start time of the flight plan	UNIX
`End`	O	The estimated end time of the flight plan	UNIX
`Phase`	O	The type of waypoint	{START, STOP, CRUISE, HOME}
`fp{Lon,Lat,Alt}`	O	The specific waypoint's longitude, latitude, and altitude.	DD/MSL
`Time_Filed`	O	The time stamp when the GCS transmitted the flight plan to the UTM.	UNIX
En route Conflict Signals			
`con_ID`	I	The conflicting UAS's flight plan ID	Int
`conStart`	I	The estimated start time of the predicted conflict	UNIX
`conEnd`	I	The estimated UNIX end time of the predicted conflict	UNIX
`con{Lon,Lat,Alt}`	I	The longitude, latitude, and altitude of the predicted an en route conflict	DD/MSL

Note that the Units of DD stands for Decimal Degrees and MSL stands for Mean Sea Level: a way to measure altitude

tions $\mathcal{AP}$ is recursively defined as:

$$\phi ::= \mathsf{true} \mid \mathsf{false} \mid p \mid \neg\phi \mid \phi_1 \wedge \phi_2 \mid \phi_1 \vee \phi_2 \mid \\ \wedge\{A\} \mid \vee\{A\} \mid \Box_I \phi \mid \Diamond_I \phi \mid \phi_1 \mathcal{U}_I \phi_2 \mid \phi_1 \mathcal{R}_I \phi_2$$

where $p \in \mathcal{AP}$ is a Boolean atom (0/1), $A \subseteq \mathcal{AP}$, ϕ_1 and ϕ_2 are MLTL formulas, and I is a closed-bound interval $[lb, ub]$, where $lb \leq ub$. Note that the new syntax, $\wedge\{A\}$ and $\vee\{A\}$, represent conjuncting and disjuncting all the propositions within A into a single proposition, respectively.

Definition 4 (MLTL w/ Set Aggregation Semantics [17,24]) The semantics of MLTL with set aggregation follow exactly MLTL's original semantics in Definition 2, with the addition of two operators:

- $\pi, i \models \wedge\{A\}$ iff for all $p \in A$, where $A \subseteq \mathcal{AP}$, $p \in \pi[i]$ (in other words, $\wedge_{p \in A} p$)
- $\pi, i \models \vee\{A\}$ iff there exists a $p \in A$, where $A \subseteq \mathcal{AP}$, $p \in \pi[i]$ (in other words, $\vee_{p \in A} p$).

Since $\wedge\{A\}$ and $\vee\{A\}$ are a conjunction or disjunction of all elements within the set A, they do not modify MLTL correctness nor increase its time or space complexity.

One benefit of set aggregation is specification debugging [26]. Often specifications are complex to read and write; this approach allows for easier validation of specifications. Set aggregation allows users to recognize patterns in specifications and provides a way that is easier to write, use, interpret, and debug for later verification. For example, we recognized that for a singular waypoint, if a value is violated, it does not matter which one, just that the UAS's flight plan has encountered an error. We were able to group similar signals (latitude, longitude, altitude, and phase) for waypoints in a flight plan to determine if one violated a given safety threshold. Essentially, we raised our level of abstraction from reasoning about an individual signal's properties to reasoning about a set of similar signal's properties. While we recognize that this level of abstraction removed some potentially useful data (e.g., knowing which waypoint in the flight plan violated a given safety property), we found that it was often more useful to know that *at least one* waypoint violated a safety property. Instead of creating multiple, complex specifications to conjunct together, it becomes possible to write only one that is easier to validate with set aggregation. However, we also recognize that there are some cases where set aggregation might not be useful, and defaulting back to conjunction is necessary. An example of this would be the `Time_Filed`. If a UAS is sending multiple flight plans, and only one of those has an incorrect time, knowing which flight plan is invalid would be necessary, hence why the use of Set Aggregation would be unnecessary.

Additionally, this approach returns a verdict faster through savings in computation time. This allows us to return a useful safety verdict (e.g., that the UAS's flight plan was bad) faster, which would then allow the GCS or UTM to take a mitigating action earlier. MLTL with set aggregation also decreases memory overhead by cutting down on the amount of intermediate values stored within an R2U2 instance. In essence, we used a single specification across a set of signals, rather than one specification per signal. This drastically reduced the number of specifications encoded in the UTM (on average by a factor of *NumID*) and we found modest reductions within the GCS for specifications involving the UAS's flight plan. We recognize that this level of abstraction is not useful for all specifications; rather, we found that a subset of safety specifications within our distributed system (particularly the GCS and the UTM) benefit from this approach.

4.3 Coverage of real-world specification types

To help organize our specifications, each one is categorized into one of six labels: (1) operating range, (2) sensor bounds, (3) rates of change, (4) control sequences, (5) physical model relationships, and (6) inter-sensor relationships. These categories resemble those of [26,35], though we add a level of granularity to several for ease of organization.

Operating Range Every sensor to, and variable within, a given system has an expected *operating range*. Should it fall below or exceed a given threshold, this may indicate a hazardous system state. For example, the proposed centralized UTM will cover a predefined airspace. Should a UAS stray beyond these operating limits of the UTM, an alert will be sent to the UTM operator to inform the corresponding UAS's GCS that they are reaching or exceeding a safety threshold of the system.

Sensor Bounds Sensors and variables also have well defined *bounds* on the values they can return. For example, a UAS should never see latitude values that are meaningless (e.g., latitude measurements less than $-90°$ or greater than $90°$). These types of specifications may be used in conjunction with *Operating Range* specifications to help diagnose whether there is a user error (accidentally operating outside their airspace) or hardware failure (sensor returning bad data to the system).

Rates of Change Additionally, sensor's and variable's *rate of change* may also be bounded. For example, a UAS will have some maximum change in velocity between any two telemetry transmissions. Should it exceed this value, it may indicate that the UAS's transmission rate varied (e.g., a dropped transmission). Additionally, one could monitor to make sure there is change between two consecutive sensor measurements, or that the amount of variance between sensor measurements is not skewed in one direction or another, which could mean the UAS is under a cyber-attack, such as GPS spoofing.

Control Sequences Because this system follows a rigorous series of stages, several specifications monitor that the system is adhering to its specified *control sequence*. For example, the intended sequence of states for the UTM is to: (1) receive a flight plan from a GCS, (2) approve or reject the flight plan, (3) if approved, issue the GCS a corresponding flight plan ID, and (4) the GCS transmits the telemetry data of the UAS with the corresponding flight plan ID. Many different hazardous situations can be made by removing or rearranging this intended sequence; thus, monitoring for any out-of-order sequences can help alert the system or the user to execute a mitigating action.

Physical Model Relationships In many systems, there exist *physical relationships* between one or more combinations of sensors and actuators when commanding the system. For example, if a UAS is commanded to accelerate, the motors should respond accordingly to execute that command. These types of relationships can detect sensor calibration errors and ensure that sensors agree about the system's overall state.

Inter-sensor Relationships To help diagnose failures, some systems may be able to invoke specifications that use multiple sensors, either of the same or different type, to measure common values. For example, the relationship between barometric pressure (obtained from an on-board barometer) and altitude (obtained from the GPS) allows for more than one way to measure altitude. RV can use these types of specifications to determine if both sensors agree. If they do not agree, then polling, or other system health management techniques, could be used to determine the faulty sensor and switch the primary source for the UAS altitude measurements.

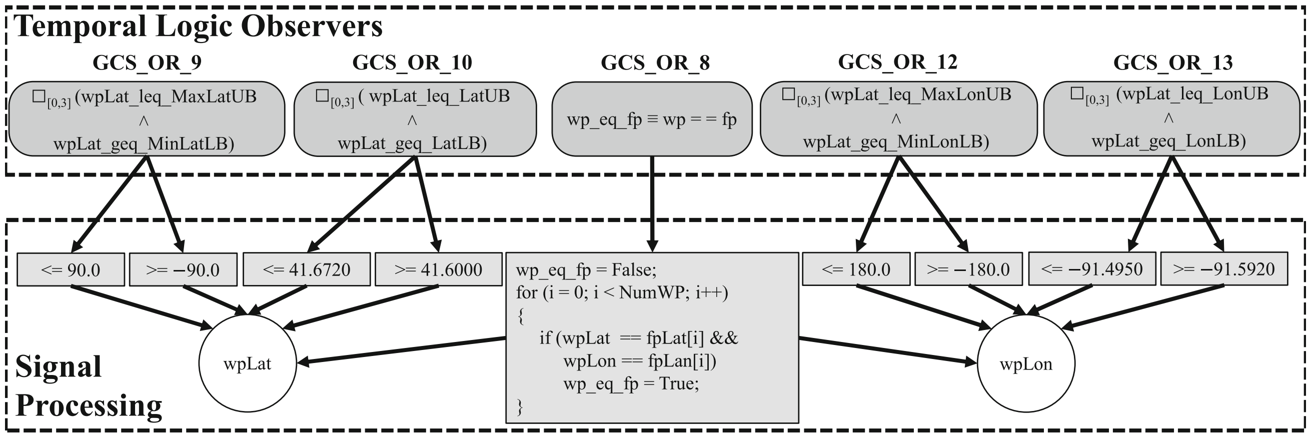
4.4 Specification validation

Because specification creation is a circular process [26], we chose to validate our list of RV specifications in a variety of ways. The first was a Matlab-based approach where we incorporated logged data for each subsystem into Matlab and validated the ways in which the Boolean atomics

Table 3 UAS, GCS, and UTM Specifications Investigated

Name	Description	MLTL specification
UAS_RC_8	The difference between two consecutive pressure `Pres` readings cannot exceed a maximum rate of climb `MaxPrevPres`	$\neg(\Box_{[0,3]}\neg(Pres\_leq\_MaxPrevPres \wedge Pres\_geq\_MinPrevPres))$
UAS_IS_1	Since the altimeter and the barometer both derive the air pressure, the error between these two measurements of pressure will be less than the `MaxPresErr` and greater than `MinPresErr`	$(PresDiff\_lt\_MaxPresErr) \wedge (PresDiff\_gt\_MinPresErr)$
GCS_CS_7	The reference latitude `LatWP` and longitude `LonWP` will be contained within the set of waypoints given in the flight plan	$wpLonLat\_eq\_fpLonLat$
GCS_PM_2	If a telemetry stream is reporting that the UAS's heading `Ang` is between 90° and 180°, then, if the UAS's velocity `Vel` is greater than 0 m/s, the UAS's latitude `Lat` should be decreasing while its longitude `Lon` should be increasing	$\neg(Ang\_eq\_Quad4 \wedge Vel\_gt\_Zero) \vee (Lat\_leq\_PrevLat \wedge Lon\_geq\_PrevLon)$
UTM_OR_11	Every UAS's position will be bounded within the given airspace All latitude `Lat` will be bounded between (41.6000°,41.6720°)	$\Box_{[0,3]}(Lat\_leq\_LatUB \wedge Lat\_geq\_LatLB)$
UTM_SB_3	Every UAS's position will exist on Earth GPS coordinates. All latitude `Lat` measurements will be bounded by (−90°,90°) degrees	$\Box_{[0,3]}(Lat\_leq\_MaxLatUB \wedge Lat\_geq\_MinLatLB)$

Each specification shown in the above table is a variable name with syntax chosen by its respective equation. The naming convention follows the trend of a signal_comparison_(signal or constant). The comparisons are less than (lt), less than or equal (leq), equal (eq), not equal(neq), greater than (gt), and greater than or equal (geq), some of which are demonstrated in the table. Also note that there is an implied $\Box$ operator outside all of the specifications due to the *stream-based* nature of R2U2 runtime observers. That is: R2U2 outputs a stream of verdicts indicating whether each specification holds starting at every discretized execution time stamp. Formally, $\forall i$, R2U2 gives a verdict as to whether $\pi, i \models \varphi$ in the form of a stream $\langle i, verdict \rangle$. So, even the purely propositional formulas are still asserting that a relationship holds, e.g., throughout a flight, and the temporal form are asserting something similar; their temporal operators are just preserving activity within a tight temporal window holds throughout the flight



```
wp_eq_fp = False;
for (i = 0; i < NumWP; i++)
{
    if (wpLat  == fpLat[i] &&
      wpLon == fpLan[i])
      wp_eq_fp = True;
}
```

Fig. 2 A small observation tree from the GCS's R2U2 implementation. Two sensor values, `wpLat` and `wpLon`, are inputs to the signal processing layer, which pre-processes them into Boolean atomics for the temporal logic observers, where the specifications are encoded

were created. The second was by uploading our MLTL runtime specifications for each individual subsystem into an open-source MLTL satisfiability checker [17] to perform specification debugging via checking each specification, its negation, and the conjunction of all specifications for satisfiability [29]. The third way these specifications were validated was by running the pre-recorded data into the R2U2 tool chain and checking to see if the specification held true over the system trace. If it did, we injected faults into the pre-recorded data and monitored R2U2's output to see if it correctly detected the faults. Of the list of 124 specifications we made for the UAS, GCS, and UTM, Table 3 presents six specifications that we feel encapsulate interesting properties about each subsystem.

5 Evaluation

The UTM test scenario consists of 20 UAS interacting with the UTM – OPL's Vapor 55 hardware-in-the-loop simulation and our 19 physics-based simulated flights – with the goal of testing the UTM's conflict detection logic. Of the 20 flight plans, 18 were conflict-free, one was designed to create a pre-departure conflict, and one deviated from the pre-approved flight plan, creating an en-route conflict. During the 42 minute test, the UTM correctly detected and alerted both GCSs of the en-route conflict, with OPL's GCS submitting a new, conflict-free flight plan en-route.

Although we intended to have R2U2 embedded into the UTM system for this test, in practice this would have required enhancements to core functionalities and improving the networking capabilities of the UTM. However, all test data was recorded and put to use offline in refining our specifications and implementations of R2U2 into each subsystem. We argue that since R2U2 has previously been embedded and used in several successful aerospace applications [12,16,19,24,31], our offline, real-time simulations of this embedding performs representatively to an actual implementation. Note we plan to incorporate R2U2 into the UTM system for the next test.

R2U2 was hosted on a Ubuntu 18.04 LTS operating system on an Intel Core i7-4810MQ CPU with a 2.80GHz clock and 16GB of RAM. Each subsystem of R2U2 was run independently, i.e., each subsystem was run with its own instance of R2U2 across its own input and no cross-platform communication was performed. Figure 2 shows an example of how specifications are encoded into R2U2's observation trees.

5.1 Analysis of MLTL with set aggregation

We analyzed the performance increase obtained from abstracting several of our specifications using MLTL with set aggregation. To demonstrate the potential benefits, we examined the worst case scenario: suppose there is a set of atomics $A = \{a_0, a_1, \ldots a_n\}$, where n is an arbitrary integer. Now let us suppose we perform a simplistic temporal logic formula, say $\Box_{[lb,ub]}$ onto either each atomic individually (original MLTL specifications) or onto the entire set of atomics (MLTL with set aggregation specification). In the original MLTL syntax, each atomic correlates to one temporal logic formula, i.e., there are n number of temporal formulas of the form $\Box_{[lb,ub]}a_i$, where $i \leq n$. Contrasting this to the new syntax for MLTL with set aggregations, there is only one temporal formula of the form $\Box_{[lb,ub]} \wedge_{i=1}^{n} a_i$.

The use of MLTL with set aggregation provides an additional feature: specification debugging. While not extending the expressiblity of MLTL, the use of set aggregation eases readability of specifications which allows for easier validation. An example of this can be provided using the specification OR_GCS_10. This specification states that the longitude reference waypoint (`wpLon`) for any UAS will be bounded between `Lon_LB` and `Lon_UB`, the longitude's lower and upper bounds respectively. Writing in MLTL, the specification is as follows, for a UAS with n waypoints:

$[(\text{wpLon}_1 \leq \text{Lon\_UB}) \wedge (\text{wpLon}_1 \geq \text{Lon\_LB})] \wedge [(\text{wpLon}_2 \leq \text{Lon\_UB}) \wedge (\text{wpLon}_2 \geq \text{Lon\_LB})] \wedge [(\text{wpLon}_3 \leq \text{Lon\_UB}) \wedge (\text{wpLon}_3 \geq \text{Lon\_LB})] \wedge \cdots \wedge [(\text{wpLon}_{NumWPs} \leq \text{Lon\_UB}) \wedge (\text{wpLon}_{NumWPs} \geq \text{Lon\_LB})]$.

For a UAS, with up to 10 waypoints, the specification rapidly becomes complicated. While correct and possible to validate, debugging and performing validating a specification of this sort is tedious. The use of MLTL with set aggregation eases reading and provides a simpler representation of the same formula:

$\wedge_{i=1}^{NumWPs} [(\text{wpLon}_i \leq \text{Lon\_UB}) \wedge (\text{wpLon}_i \geq \text{Lon\_LB})]$.

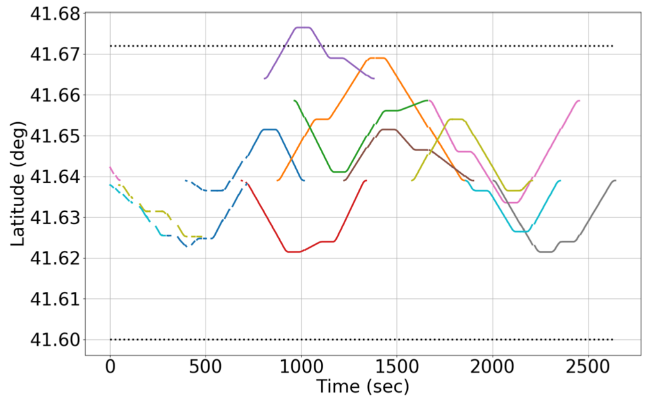
MLTL with set aggregation was manually implemented for performance analysis in comparison to MLTL. The GCS uses set aggregation in 13 of its 30 specifications. Since the GCS's specifications that use set aggregation involved the flight plan, not using set aggregation would have resulted in $17+13(\mathit{NumWPs})$ number of specifications, where NumWPs varies from 3 to 10 waypoints (but averaged around 4 waypoints). For the UTM, the savings were even greater, since nearly all of its specifications (40 out of 42) use set aggregation. This allowed the UTM to keep at a constant 42 specifications, compared to the $2 + 40(\mathit{NumUAS})$ number of specifications it would have had without set aggregation.

The reduction of the number of specifications correlates to a reduced memory usage as well. While part of this was a result of reducing the overall number of specifications by conjuncting specifications, which can be done in MLTL without set aggregation, it was also due to implementing the conjunction or disjunction of sets in the signal processing layer. Since each atomic is a bit and a time stamp, by decreasing the number of atomics, we decreased memory usage. We found that in the best case, there was a reduction of 42.9% in memory, while the worst case gave a reduction of 42.1% savings. These savings transfer to performance as well. The atomic conversions for these tests were implemented manually using Matlab, with varying parameters. Using a range of one to all of the UTM Operating Range Specifications, and changing the number of specification used, and which were used per run, we found that the reduction in runtime decreased, on average by 40%.

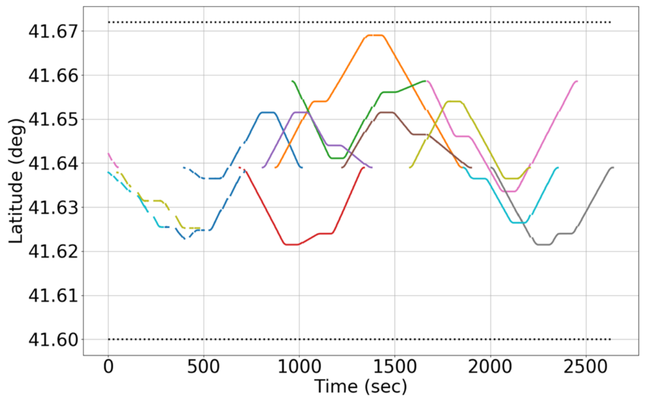
While beyond the scope of this paper, potential implementations are planned for the future and theorized to maintain or improve performance.

5.2 Specification evaluation

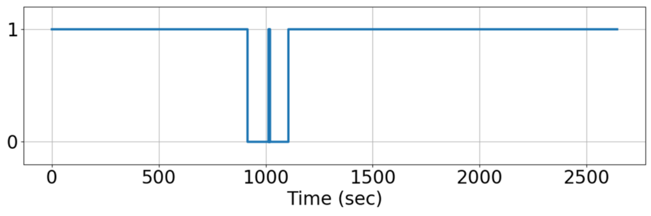
Operating Range As seen from Fig. 3, the UTM's R2U2 monitors and reports if the operating range bounds are satisfied for all of UAS's latitude measurements. As the original test data was fault-free, we injected a fault, which revealed a sudden spike in R2U2's output during the injected fault. This corresponds to a dropped transmission in the original data. Thus, we refine our specification to include an overarching $\Box_{[0,3]}$ operator, which acts as a sliding window temporal filter, to suppress such output bouncing.



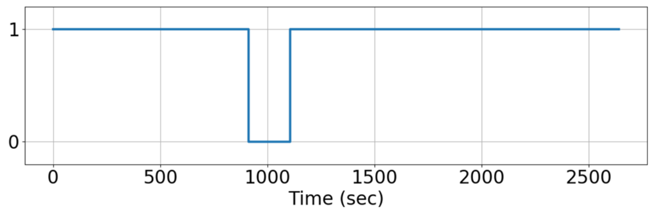
(a) OR_UTM_11: Modified latitude trace



(b) OR_UTM_11: Unmodified latitude trace

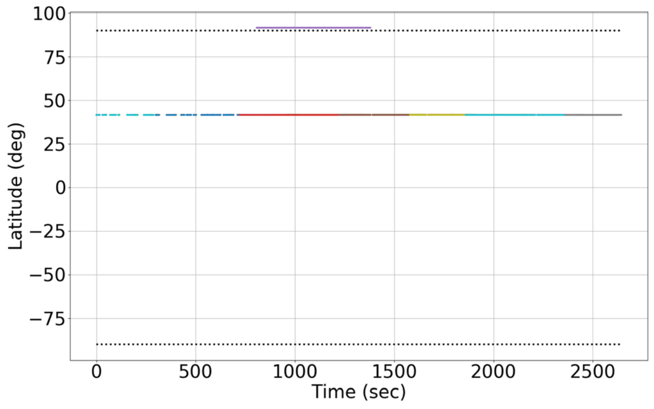


(c) $(Lat \leq LatUB) \wedge (Lat \geq LatLB)$

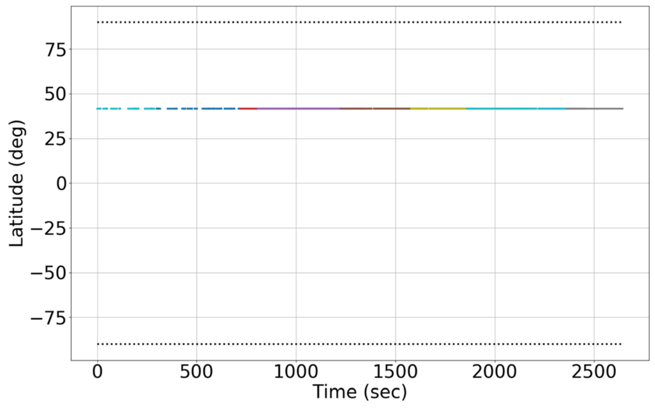


(d) $\Box_{[0,3]}((Lat \leq LatUB) \wedge (Lat \geq LatLB))$

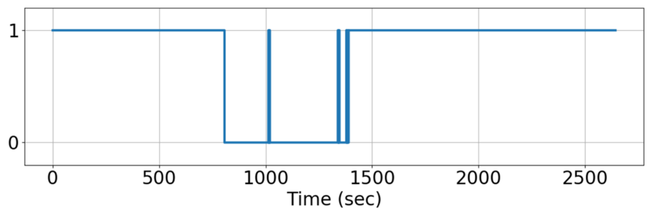
Fig. 3 Two instances of the UTM's R2U2 monitoring: a modified run **a** where one UAS (purple) temporarily exceeds the operating range bounds, and an unmodified run **b** where all UAS lie within the operating range (dashed lines). Both fault-injected runs show R2U2 identifies the corresponding violation of the specification; however, the output of the purely Boolean formula **c** bounces due to a missed telemetry transmission. To avoid a false positive, due to missing data, we add a temporal logic filter **d** that monitors for multiple subsequent nominal data sequences



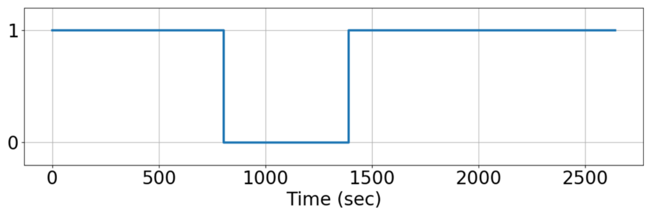
(a) SB_UTM_3: Modified latitude trace



(b) SB_UTM_3: Unmodified latitude trace

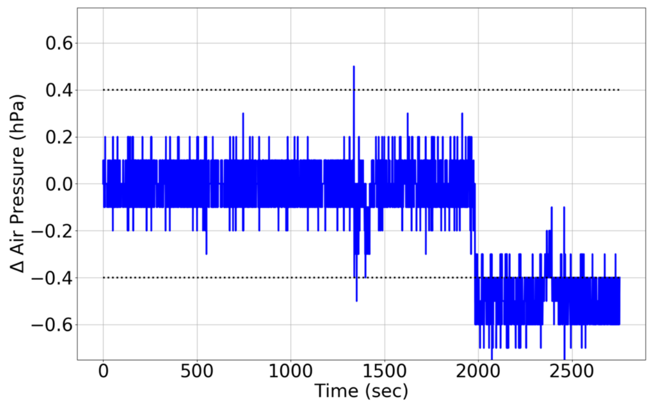


(c) $(Lat \leq MaxLatUB) \wedge$
$(Lat \geq MinLatLB)$

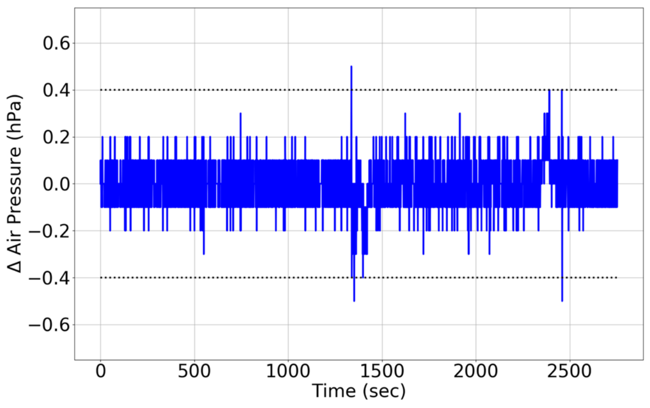


(d) $\Box_{[0,3]}((Lat \leq MaxLatUB) \wedge$
$(Lat \geq MinLatLB))$

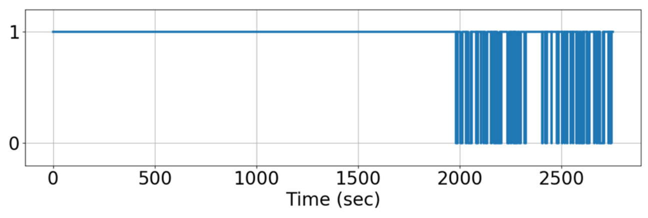
Fig. 4 Like Fig. 3, the top graphs show modified (**a**) and unmodified (**b**) input traces. Similarly, dropped telemetry transmissions cause output bouncing (**c**), so a $\Box_{[0,3]}$ filter is applied (**d**)



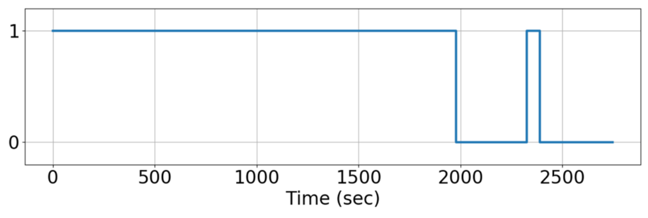
(a) RC_UAS_8: Modified air pressure trace



(b) RC_UAS_8: Unmodified air pressure trace



(c) $\neg\square_{[0,3]}\neg(Pres \leq MaxPrevPres \wedge Pres \geq MinPrevPres)$



(d) $\square_{[0,3]}((Pres \leq MaxPrevPres)\wedge(Pres \geq MinPrevPres))\wedge \neg\square_{[0,3]}\neg((Pres \leq MaxPrevPres) \wedge (Pres \geq MinPrevPres))$

Fig. 5 Two instances of the UAS's R2U2 monitoring: (**a**) a modified trace where we injected a shift in the air pressure's rate of change, and (**b**) an unmodified trace where a few anomalies exceed the pressure rate of change bounds (dashed lines). Both outputs of the fault-injected run from R2U2 are shown; however, the output of the original formula (**c**) bounces due to noisy input jumping back within the margins. To remove this bouncing, we added another $\square_{[0,3]}$ filter (**d**) to keep the current state until all outliers are filtered and the state has unquestionably changed

Sensor Bounds Similar to Figs. 3 and 4 shows the UTM's R2U2 monitoring and reporting if any of the UAS's latitude measurements exceed the sensor bound threshold of $(-90°, 90°)$. Similarly, the original data was fault free, so we injected a fault into one of the UAS's latitude measurements. Again, testing revealed transmission losses, so we added a $\square_{[0,3]}$ filter to suppress any false positives triggered by missing data.

Rates of Change The pressure recorded by a UAS's on-board barometer changes as it ascends and descends. Thus, we developed a specification to monitor change in pressure: the difference between two consecutive pressure readings are limited to ± 0.4 hPa (derived from the maximum rates of climb and descent [21]). Unlike our other specifications, Fig. 5 shows that we needed to include a conjunction of two $\square_{[0,3]}$ filters to remove all output bouncing: one filters outlying violating verdicts and one filters outlying satisfying verdicts.

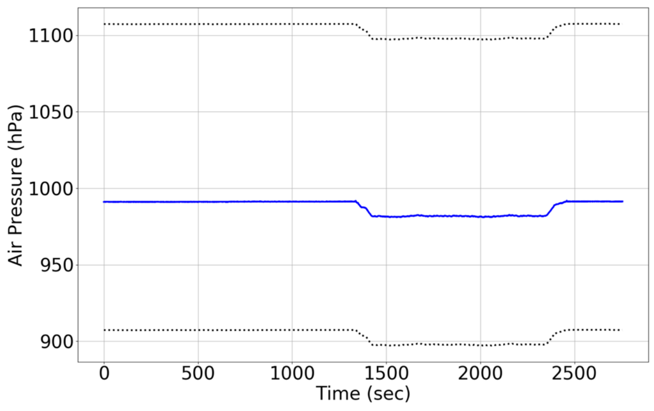
Control Sequence The UTM's test scenario included one UAS deviating from its pre-approved flight plan. Figure 7 shows R2U2 correctly detecting this real-world deviation in real time.

Inter-sensor Relationship The difference between the barometer's and GPS's pressure should be bounded within acceptable error. A comparison of the two sensors can help diagnose sensor failures (see [21] for more details). For example, Fig. 6 shows a side-by-side comparison of two pressure traces: an unmodified and a modified version with a fault injected from $t = 1500$ to $t = 1750$.

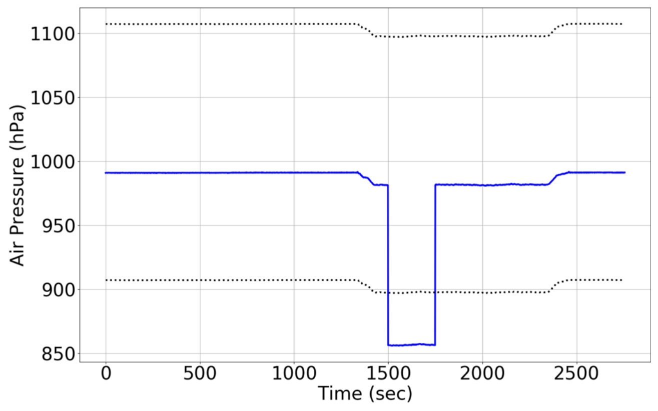
Physical Model Relationship As shown in Fig. 8, when a UAS's heading is between 90° and 180° and its velocity is non-zero, then the UAS's latitude should be decreasing while its longitude is increasing.

5.3 Lessons learned

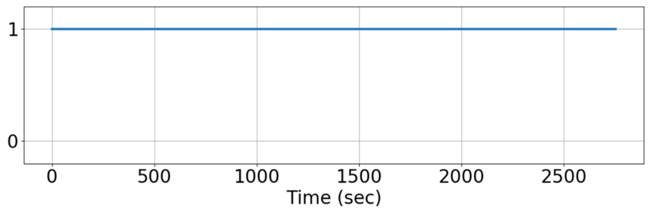
Many of our specifications are rather simplistic, e.g., $\square_{[0,3]}(\varphi_1 \wedge \varphi_2)$; however, their simplicity allows for easy validation and verification. They are easy to validate through discussion with system designers. Additionally, we used temporal filters, e.g., the $\square_{[0,3]}$ sliding window filter, extensively to mitigate false-positives. As false-positives can cause mistrust of the RV monitor, we built our specifications to err on the side of missing a fault. As seen in Sect. 5, if R2U2 sent a fault



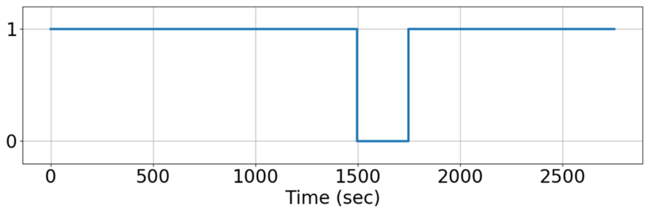
(a) IS_UAS_1: Unmodified pressure sensor trace



(b) IS_UAS_1: Modified pressure sensor trace



(c)
$\Box_{[0,3]}((Pres < MaxPresErr) \wedge (Pres > MinPresErr))$



(d)
$\Box_{[0,3]}((Pres < MaxPresErr) \wedge (Pres > MinPresErr))$

Fig. 6 Two instances of the UAS's R2U2 monitoring: an unmodified run (a) where the pressure from the barometer remains within the error margins of the GPS's calculated atmospheric pressure (dashed lines), and a modified run (**b**) where the same data was injected with a fault by subtracting 100hPa from the barometer's atmospheric pressure reading. R2U2's output (**c**) acknowledges the error-free trace of (**a**), and (**d**) shows that R2U2 detects the violation from (**b**)

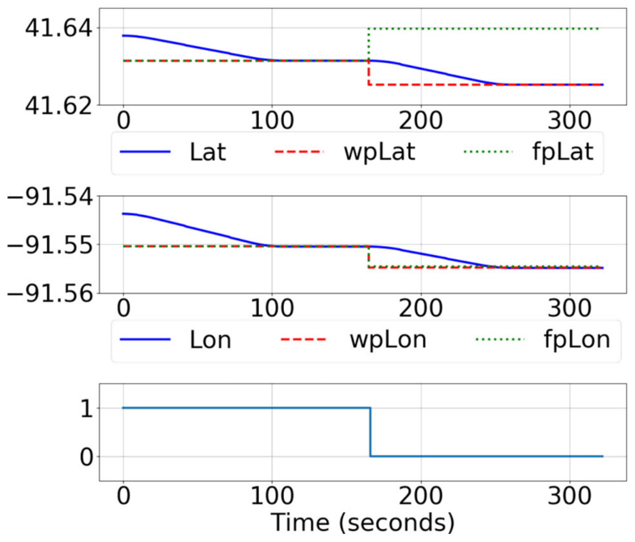


Fig. 7 The latitude (top) and longitude (middle) traces for an adversarial UAS, showing that the GCS is commanding it to a different waypoint (red, dashed line) instead of one from its approved flight plan (green, dotted line). Corresponding to the violation of CS_GCS_7 (Table 3), R2U2's output (bottom) shows it successfully detects this real-world fault

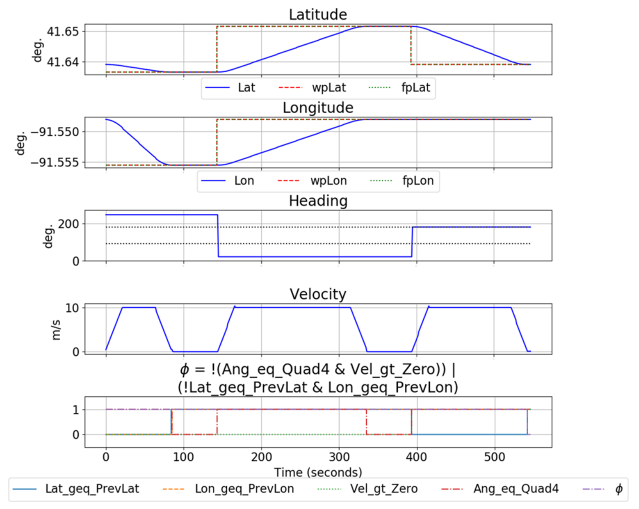


Fig. 8 Single instance of R2U2 on a simulated UAS showing the latitude, longitude, heading, and velocity. With assumptions of a relationship between heading and trajectory and the UAS operating in North America, then a relationship between velocity, heading, and position can be verified

alert, the fault was clear for the human operators receiving the alert. Many of our specifications encapsulate intuitive bounds and relationships for sensor values and variables that humans implicitly assume about a given system, e.g., latitude coordinates are bounded between $(-90°, 90°)$ and that events cannot end before they start. These "common-sense" specifications are often overlooked, yet they catch real faults, e.g., from variable overflow and underflow, sensor or wiring failures, and excessive noise. Our coverage categorization for specifications allowed us to enumerate many such sanity checks about the UTM system, which helped us achieve a reasonable covering set of specifications for the UTM's three sub-systems. In practice, this lead to R2U2 identifying a real-life fault where a data-translation error caused the UTM to register flight plans that ended before they started. Such an error would be obvious to human controllers but automated systems require RV to flag this impossibility. We also find that the use of set aggregation, while not providing MLTL with more expressive syntax, allows for easier debugging of specifications due to readability. MLTL with set aggregation maintains contexts to be used in monitor optimizations, while potentially maintaining the same performance. Future work is aimed toward creating automated tools for specification elicitation and implementation of set aggregation into R2U2.

6 Conclusion

Before UAS can integrate into the NAS, we need to establish a provably safe, intelligent, and automated UTM system. To help facilitate this, we have integrated the state-of-the-art runtime verification tool R2U2 across the three different layers of an actual UTM implementation: on-board the individual UAS, in conjunction with each operator's GCS, and embedded into a centralized, cloud-based UTM server. By validating and releasing over 100 runtime MLTL specifications, two sets of recorded traces from test flights of a real-life UTM implementation, and the results of checking those formulas, we contribute a large benchmark suite. This suite is useful for verification of the algorithms and implementations of future RV tools, providing both nominal and faulty traces and realistic sensor noise and outlier readings that challenge RV engines. Additionally, we exemplify the real-world challenges of implementing RV into a centralized, high-traffic UTM. We demonstrate real-time performance of our newly developed MLTL with set aggregation, where we conjunct or disjunct *sets* of atomic propositions to create a higher level of specification abstraction. These specifications verify that a property holds *for all* UAS or *at least one* member of a set, which allows for a system to respond to a failed verdict faster. For our implementation on a GCS, 13 out of 30 specifications use set aggregation; for the UTM, 40 out of 42 use set aggregation. This extension of MLTL is part of the first step of distributed system health management: a system must know something is wrong (i.e., at least one node is violating a safety specification) so that it can alert an operator and/or perform some mitigating action. When refining our specification set, we found sensor noise and outliers triggered false positives and that a simple $\Box_{[0,3]}$ around each critical sensor check eliminated these while only slightly delaying the trigger of actual faults. Of our 124 specifications, two-thirds contain this construct. This modification can be automatically inserted into specifications for real-life systems where false positives cannot be tolerated. Our specification patterns and efficient encoding into R2U2 provide a basis for future work in more automated specification elicitation and embedding into distributed systems. Though we verified a short (42 minute) relatively small real-life system (26, 33-64, and 634 sensor inputs for the UAS, GCS, and UTM, respectively) we still found it hard to manually write a sufficiently covering set of specifications to catch faults in this distributed system, like that flight plan end times cannot proceed flight plan start times. To ensure we did not miss covering unstated assumptions, we used coverage metrics to brainstorm our list of 124 specifications: variable coverage (every variable appears in at least one specification) and pattern coverage (specifications follow each pattern from [26]). Our experience informs an on-going project to enable more automated specification elicitation.

Acknowledgements The authors would like to acknowledge Brian Kempa and Christopher Johannsen at Iowa State University for their assistance regarding MLTL with set aggregation.

References

1. AeroViroment: VAPOR All-electric Helicopter UAS (2019) https://www.avinc.com/uas/view/vapor-vtol. Last Accessed on Dec. 17
2. Alur R, Henzinger TA (1993) Real-time logics: complexity and expressiveness. Inf Comput 104(1):35–77
3. Aweiss AS, Owens BD, Rios JL, Homola JR, Mohlenbrink CP (2018) UAS traffic management national campaign II. In: 2018 AIAA SciTech, pp 1–16
4. Basin D, Dardinier T, Heimes L, Krstic S, Raszyk M, Schneider J, Traytel D (2020) A formally verified. https://doi.org/10.1007/978-3-030-51074-9_25
5. Basin D, Klaedtke F, Muller S, Salinescu E (2015) Monitoring metric first-order temporal properties. J Assoc Comput Mach. https://doi.org/10.1145/2699444
6. Ben-David S, Chechik M, Gurfinkel A, Uchitel S (2011) Cssl: A logic for specifying conditional scenarios. In: SIGSOFT/FSE 2011—Proceedings of the 19th ACM SIGSOFT symposium on foundations of software engineering, Association for Computing Machinery, pp 37–47. https://doi.org/10.1145/2025113.2025123
7. Cornell University: Symbolic LTLf Synthesis (2017). https://doi.org/10.24963/ijcai.2017/189
8. Cornell University (2019) First-Order vs. Second-Order Encodings for LTLf-to-Automata Translation, vol abs/1901.06108

9. Federal Aviation Administration (FAA) (2019) FAA Aerospace Forecast—Fiscal Years 2019–2039. Online:https://www.faa.gov/data_research/aviation/aerospace_forecasts/media/FY2019-39_FAA_Aerospace_Forecast.pdf
10. Federal Aviation Administration (FAA) (2020) Unmanned Aerial Systems (UAS). Online: https://www.faa.gov/uas/
11. Finkbeiner B, Gieseking M, Hecking-Harbusch J (2020) AdamMC: a model checker for petri nets with transits against flow-LTL. Comput Aided Verif. https://doi.org/10.1007/978-3-030-53291-8_5
12. Geist J, Rozier KY, Schumann J (2014) Runtime observer pairs and Bayesian network reasoners on-board fpgas: flight-certifiable system health management for embedded systems. In: RV14, vol 8734, pp 215–230. Springer
13. Havelund K, Peled D (2018) Runtime verification: from propositional to first-order temporal logic. In: Proceedings of 18th international conference, RV 2018, Limassol, Cyprus, November 10–13, 2018, pp 90–112. Springer International Publishing. https://doi.org/10.1007/978-3-030-03769-7_7
14. Havelund K, Peled D (2019) An extension of LTL with rules and its application to runtime verification, pp 239–255. Springer, Cham. https://doi.org/10.1007/978-3-030-32079-9_14
15. Hunter G, Wei P (2019) Service-oriented separation assurance for small uas traffic management. In: INCS19, pp 1–11
16. Kempa B, Zhang P, Jones PH, Zambreno J, Rozier KY (2020) Embedding online runtime verification for fault disambiguation on robonaut2. In FORMATS, LNCS. Springer
17. Li J, Vardi MY, Rozier KY (2019) Satisfiability checking for mission-time LTL. In: CAV, LNCS, vol 11562, pp 3–22. Springer, New York
18. Martin N (2020) FAA, NASA conclude Drone traffic mgmt pilot program's second phase; pamela whitely quoted. Online:https://www.executivegov.com/2020/11/faa-nasa-conclude-drone-traffic-mgmt-pilot-programs-second-phase-pamela-whitley-quoted/
19. Moosbrugger P, Rozier KY, Schumann J (2017) R2U2: Monitoring and diagnosis of security threats for unmanned aerial systems. FMSD pp 1–31
20. NASA: Unmanned Aircraft System (UAS) Traffic Management (UTM) (2020) https://utm.arc.nasa.gov/index.shtml. Last Accessed on Mar. 12
21. NASA: Earth atmosphere model (2015) Online: https://www.grc.nasa.gov/WWW/K-12/airplane/atmosmet.html
22. Pike L, Wegmann N, Niller S, Goodloe A (2013) Copilot: monitoring embedded systems. Innovat Syst Softw Eng 9(4):235–255
23. Reichmann K (2020) FAA, NASA UAS demonstrations mark end of UTM pilot program. Online:https://www.aviationtoday.com/2020/11/18/faa-nasa-uas-demonstrations-mark-end-utm-pilot-program/
24. Reinbacher T, Rozier KY, Schumann J (2014) Temporal-logic based runtime observer pairs for system health management of real-time systems. In TACAS, pp 357–372
25. Rios J, Mulfinger D, Homola J, Venkatesan P (2016) NASA UAS traffic management national campaign: Operations across Six UAS Test Sites. In DASC, pp 1–6
26. Rozier KY (2016) Specification: the biggest bottleneck in formal methods and autonomy, 9971st edn. Springer, pp 8–26
27. Rozier KY, Schumann J (2017) R2U2: Tool overview. RV-CUBES, 3rd edn. Kalpa Publications, Seattle, WA, USA, pp 138–156
28. Rozier KY, Schumann J, Ippolito C (2015) Intelligent hardware-enabled sensor and software safety and health management for autonomous UAS. Tech. Rep. 20150021506, NASA Ames Research Center, Moffett Field, CA 94035, USA
29. Rozier KY, Vardi MY (2010) LTL satisfiability checking. Int J Softw Tools Technol Transfer 12(2):123–137
30. Schirmer S (2016) Runtime monitoring with lola. Master's thesis, Saarland University
31. Schumann J, Rozier KY, Reinbacher T, Mengshoel OJ, Mbaya T, Ippolito C (2015) Towards real-time, on-board, hardware-supported sensor and software health management for unmanned aerial systems. IJPHM 6(1):1–27
32. The international conference on runtime verification. Online: https://www.runtime-verification.org/ (2001–present)
33. Wargo CA, Glaneuski J, Hunter G, DiFelici J, Blumer T, Hasson D, Carros P, Kerczewski RJ (2018) Ubiquitous surveillance notional architecture for system-wide daa capabilities in the nas. In 2018 IEEE Aerospace Conference, pp 1–14
34. Wei P, Atkins EM, Hunter G, Rozier KY, Schnell T (2017) Pre-departure dynamic geofencing, en-route traffic alerting, emergency landing and contingency management for intelligent low-altitude airspace UAS Traffic Management. Online: https://www.nsf.gov/awardsearch/showAward?AWD_ID=1718420
35. Zhao Y, Rozier KY (2014) Formal specification and verification of a coordination protocol for an automated air traffic control system. Science of Computer Programming, P 96
36. Zhu G, Wei P (2016) Low-altitude UAS traffic coordination with dynamic geofencing. In 16th AIAA Aviation Technology, Integration, and Operations Conference