Model Predictive Runtime Verification for Cyber-Physical Systems with Real-Time Deadlines *

Pei Zhang[1] [0000-0001-9560-2175],
Alexis Aurandt[2] [0000-0003-2008-673X], Rohit Dureja**[3] [0000-0002-7152-8115],
Phillip H. Jones[2] [0000-0002-8220-7552], and Kristin Yvonne Rozier[2] [0000-0002-6718-2828]

[1] Google LLC, USA piz@google.com
[2] Iowa State University, USA {aurandt, phjones, kyrozier}@iastate.edu
[3] IBM Corporation, USA

Abstract. Cyber-physical systems often require fault detection of future events in order to mitigate the fault before failure occurs. Effective on-board runtime verification (RV) will need to not only determine the system's current state but also predict future faults by predetermined mitigation trigger deadlines. For example, if it takes three seconds to deploy the parachute of an Unmanned Aerial System (UAS), the deployment of the parachute must be triggered three seconds before it is needed to mitigate the impending crash. To allow for the detection of future faults by deadlines, we design a real-time Model Predictive Runtime Verification (MPRV) algorithm that uniquely uses current data traces whenever possible, predicting only the minimum horizon needed to make an on-deadline evaluation. Although MPRV is extensible to other RV engines, we deploy the algorithm on the R2U2 RV engine due to R2U2's resource-aware architecture, real-time guarantees, and deployment history. We demonstrate the utility of the MPRV algorithm through a quadcopter case study and evaluate the effectiveness of our implementation by conducting memory usage and runtime performance analysis in a resource-constrained FPGA environment.

1 Introduction

Modern cyber-physical systems such as Unmanned Aerial Systems (UAS), autonomous driving systems, and human-interactive robots require runtime verification (RV) to ensure that they uphold design requirements and react to unanticipated events during deployment [11]. On-board RV can detect violations of system requirements and trigger appropriate mitigation actions. For RV to be effective, some element of prediction is often required; waiting until sensor data confirms a fault occurred can limit mitigation options.

Predictive RV was first introduced in [36, 63], but these works focus only on the prediction of *untimed* properties. In [47], the idea of synthesizing online monitors to predict *timed* properties is introduced. However, the system model must be known and represented as a deterministic timed automaton, which is not compositional, resource-aware, or executable in real-time or in hardware. As a result, this approach is not scalable or applicable to many system models or applications. We introduce Model Predictive Runtime Verification (MPRV), which monitors *timed* properties but allows for integration with *any* user-defined model predictor.

In more recent work, several apply knowledge of the system to produce a model predictor [19, 23, 60, 61], while others address generating a model predictor when a model is not known a priori (i.e., black-box systems) or when a system is subject to perturbations [6–8, 24]. Others also take into consideration the uncertainty of the model predictor when considering the

* Supported by NSF:CPS Award 2038903. Artifacts at https://zenodo.org/record/8076503.
** Work performed as a graduate student at Iowa State University

supervisory controller's actions [38, 57, 60]. To the best of our knowledge, these tools are all restricted to software implementations and have not been deployed on resource-constrained, real-time systems. We prove resource and time constraints for the MPRV algorithm that demonstrate its adaptability to RV in resource-constrained environments with real-time requirements.

Cyber-physical systems often deploy with a small, finite set of possible mitigation actions should a fault occur, and each action typically has a known deployment time. For example, a UAS may have a parachute that takes three seconds to unfurl, but the parachute must be fully unfurled within two seconds after motor failure to be effective. If the parachute does not unfurl in time, an unsafe crash will occur. We can define a mitigation trigger deadline d as the deadline by which a mitigation decision needs to be determined. In this case, d is one second before motor failure occurs (i.e., $d = -1$). Note that d can change with the property being monitored or during the mission. With our example, the same mission property may require evaluation with an earlier d when the UAS is flying at low altitudes, but then with no d at all when the UAS is flying so low to the ground that a parachute would not be useful. We may monitor with a later d when the UAS is flying at a high altitude; in this case, there may even be time to rigorously confirm the motor failure without any predicted data.

MPRV answers the following question: *by a given mitigation trigger deadline d, what is the two-valued verdict (*true *or* false*) of a given timed property φ based on maximum real data?* **No other tool or algorithm answers this question** in an implementation independent, real-time, online setting, much less in a resource-aware, on-board embeddable fashion, such as on an FPGA. Additionally, the deadline d may vary across different properties or in one property across different operational modes (e.g., UAS flying at low altitudes versus high altitudes); therefore, a modular, generalizable algorithm is required. Effective mitigation triggering depends upon making the most accurate decision by the deadline d. In most cases, real data is more accurate than predicted data (assuming the RV engine is also monitoring for sensor malfunctions); therefore, MPRV uniquely uses real data up to time d with the minimal necessary predicted data to make a decision by d.

As the NASA Lunar Gateway team's 2021 survey confirms [20], only two RV engines run in real-time on embedded platforms, R2U2 [48, 50] and LOLA [3, 21], but neither of them provides prediction. Many RV designers have recognized the need for hardware implementations suitable for real-time embedded systems [30, 32, 34, 39, 43, 46, 55, 56, 58]; however, R2U2 is the only one that has a currently-maintained implementation, evaluates both true and false verdicts (versus only property failure) over a future-time logic without evaluation delay, and monitors formulas (versus analysis requiring software instrumentation). R2U2 also has a history of being deployed on several mission-critical, resource-constrained, flight-certified systems [5, 17, 31, 34]. We exemplify the R2U2 extension and leave the LOLA extension to future work. We choose Mission-time Linear Temporal Logic (MLTL) for property specification as it intuitively captures our quadcopter case study's requirements, is a timed logic, and R2U2 encodes it natively. We also introduce a new MLTL semantics that incorporates deadlines to accurately represent our MPRV specifications. However, MPRV is *extensible* to other logics and RV engines.

Our contributions include **(1)** MLTL semantics with deadline (Definition 10), **(2)** a formal definition of MPRV (Definition 13), **(3)** a generic algorithm for MPRV and proof of its correctness (Algorithm 1 and Theorem 1), **(4)** a specialized algorithm for implementing MPRV utilizing the R2U2 framework (Algorithm 2) and analysis of its worst-case execution time and memory usage (Section 3.3), **(5)** application in a quadcopter case study (Section 4),

and **(6)** memory usage and runtime performance analysis of MPRV in a resource-constrained FPGA environment (Section 5).

2 Preliminaries

2.1 Mission-Time Linear Temporal Logic (MLTL) [37, 48]

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

$$\varphi ::= \mathsf{true} \mid \mathsf{false} \mid p \mid \neg\varphi \mid \varphi_1 \wedge \varphi_2 \mid \varphi_1 \vee \varphi_2 \mid \Box_I \varphi \mid \Diamond_I \varphi \mid \varphi_1 \, \mathcal{U}_I \, \varphi_2 \mid \varphi_1 \, \mathcal{R}_I \, \varphi_2$$

where $p \in \mathcal{AP}$ is an atom and φ_1, φ_2 are MLTL formulas. I is a closed interval $[lb,ub]$ where $lb \leq ub$ and $lb,ub \in \mathbb{N}_0$, or simply $[ub]$ if $lb = 0$.

Definition 2. *(MLTL Semantics)* The semantics of an MLTL formula over atomic propositions $\mathcal{AP}$ is interpreted over a bounded finite trace π. Every position $\pi(i)$ (where $i \geq 0$) is an assignment over $\in 2^{\mathcal{AP}}$. $|\pi|$ denotes the length of π (where $|\pi| < \infty$), and $\pi[m,n]$ denotes the trace segment $\pi(m)$,$\pi(m+1)$,...,$\pi(n)$. Given two MLTL formulas φ_1 and φ_2, $\varphi_1 \equiv \varphi_2$ denotes that they are semantically equivalent. MLTL keeps the standard operator equivalences from LTL, including $\mathsf{false} \equiv \neg\mathsf{true}$, $\varphi_1 \vee \varphi_2 \equiv \neg(\neg\varphi_1 \wedge \neg\varphi_2)$, $\neg(\varphi_1 \mathcal{U}_I \varphi_2) \equiv (\neg\varphi_1 \mathcal{R}_I \neg\varphi_2)$, $\neg\Diamond_I \varphi \equiv \Box_I \neg\varphi$, $\Diamond_I \varphi \equiv (\mathsf{true} \, \mathcal{U}_I \varphi)$, and $(\Box_I \varphi) \equiv (\mathsf{false} \, \mathcal{R}_I \varphi)$. (Notably, MLTL discards the next ($\mathcal{X}$) operator, which is essential in LTL, since $\mathcal{X}\varphi$ is semantically equivalent to $\Box_{[1,1]}\varphi$ [37]). We recursively define $\pi,i \models \varphi$ (trace π starting from time index $i \geq 0$ satisfies, or "models" MLTL formula φ) as

- $\pi,i \models \mathsf{true}$,
- $\pi,i \models p$ for $p \in \mathcal{AP}$ iff $p \in \pi(i)$,
- $\pi,i \models \neg\varphi$ iff $\pi,i \not\models \varphi$,
- $\pi,i \models \varphi_1 \wedge \varphi_2$ iff $\pi,i \models \varphi_1$ and $\pi,i \models \varphi_2$,
- $\pi,i \models \varphi_1 \, \mathcal{U}_{[lb,ub]} \, \varphi_2$ iff $|\pi| \geq i + lb$ and $\exists j \in [i+lb, i+ub]$ such that $\pi,j \models \varphi_2$ and for every $k < j$ and $k \in [i+lb, i+ub]$ we have $\pi,k \models \varphi_1$

Definition 3. *(MLTL Satisfiability [37])* Given an MLTL formula φ, the satisfiability problem asks whether there exists a finite trace π starting at time index i, such that $\pi,i \models \varphi$.

Propagation Delay. To evaluate the satisfiability of a future-time MLTL formula φ at time index i in a trace π, the RV engine may need to know the evaluation of atomic propositions at future time stamps, e.g., if $a \in \mathcal{AP}$ is false at time 0 in π, then we can evaluate that $\pi,0 \not\models \Box_{[0,5]} a$ at time 0. However, if a is always true in π, we cannot make the assertion that $\pi,0 \models \Box_{[0,5]} a$ until time 5. In this example, the best-case propagation delay of φ is 0, and the worst-case propagation delay is 5.

Definition 4. *(Propagation Delay)* Given an MLTL formula φ and trace π starting from time index $i \geq 0$, let k be the time stamp when the satisfiability of $\pi,i \models \varphi$ is determinable. The propagation delay of formula φ, denoted $\varphi.pd$ is the number of time stamps between i and k, i.e., $\varphi.pd = k - i$.

Definition 5. *(Best-case Propagation Delay)* The best-case propagation delay of an MLTL formula φ, denoted $\varphi.bpd$, is the minimum propagation delay required to determine the satisfiability of $\pi,i \models \varphi$, i.e., $\varphi.bpd = min(\varphi.pd)$.

Definition 6. *(Worst-case Propagation Delay)* The worst-case propagation delay of an MLTL formula φ, denoted $\varphi.wpd$, is the maximum propagation delay required to determine the satisfiability of $\pi,i \models \varphi$, i.e., $\varphi.wpd = max(\varphi.pd)$.

Definition 7. *(Propagation Delay Semantics)* The best- and worst-case propagation delay for an MLTL formula φ is structurally defined as follows:

$$\bullet\ \varphi \in \mathcal{AP}: \begin{cases} \varphi.wpd = 0 \\ \varphi.bpd = 0 \end{cases} \qquad \bullet\ \varphi = \neg\psi: \begin{cases} \varphi.wpd = \psi.wpd \\ \varphi.bpd = \psi.bpd \end{cases}$$

$$\bullet\ \varphi = \Box_{[lb,ub]}\psi \text{ or } \varphi = \Diamond_{[lb,ub]}\psi: \begin{cases} \varphi.wpd = \psi.wpd + ub \\ \varphi.bpd = \psi.bpd + lb \end{cases}$$

$$\bullet\ \varphi = \varphi_1 \vee \varphi_2 \text{ or } \varphi = \varphi_1 \wedge \varphi_2: \begin{cases} \varphi.wpd = max(\varphi_1.wpd, \varphi_2.wpd) \\ \varphi.bpd = min(\varphi_1.bpd, \varphi_2.bpd) \end{cases}$$

$$\bullet\ \varphi = \varphi_1 \mathcal{U}_{[lb,ub]} \varphi_2 \text{ or } \varphi = \varphi_1 \mathcal{R}_{[lb,ub]} \varphi_2: \begin{cases} \varphi.wpd = max(\varphi_1.wpd, \varphi_2.wpd) + ub \\ \varphi.bpd = min(\varphi_1.bpd, \varphi_2.bpd) + lb \end{cases}$$

where $\mathcal{AP}$ is the set of atomic propositions and lb and ub stand for the lower and upper bounds of an interval, respectively.

2.2 Abstract Syntax Tree

R2U2 [34, 48, 50, 51] decomposes the formula φ into subformulas using a parse tree. It computes the satisfaction of every subformula from the bottom up and propagates the verification results to the root-level formula φ. R2U2 uses optimized automatic code generation to synthesize asynchronous (event-triggered) observers that output *execution sequences* over finite time stamps.

Definition 8. *(Execution Sequence for Asynchronous Observers [48])* An execution sequence for an MLTL formula φ, denoted $\langle T_\varphi \rangle$, over trace π is a sequence of tuples $T_\varphi = (v, \tau)$, where $\tau \in \mathbb{N}_0$ is a time index and $v \in \{\mathsf{true}, \mathsf{false}\}$ is a verdict.

We use an integer superscript to access a particular tuple in $\langle T_\varphi \rangle$, e.g., T_φ^0 is the first tuple in $\langle T_\varphi \rangle$. Elements in T_φ are referenced as $T_\varphi.\tau$ and $T_\varphi.v$. We say T_φ holds if $T_\varphi.v$ is true, and T_φ does not hold if $T_\varphi.v$ is false. For a given execution sequence $\langle T_\varphi \rangle = T_\varphi^0, T_\varphi^1, T_\varphi^2, T_\varphi^3, \ldots$, the tuple accessed by T_φ^n corresponds to a section of satisfaction of φ such that $T_\varphi^n.v$ is true if and only if $\forall i \in [T_\varphi^{n-1}.\tau + 1, T_\varphi^n.\tau]$, we have $\pi, i \models \varphi$. Similarly, $T_\varphi^n.v$ is false if there $\exists i \in [T_\varphi^{n-1}.\tau + 1, T_\varphi^n.\tau]$ such that $\pi, i \not\models \varphi$. We say an execution sequence tuple T_φ is *produced* by the observer for φ when $T_\varphi.\tau \geq i$ and $T_\varphi.v \in \{\mathsf{true}, \mathsf{false}\}$ according to $\pi, i \models \varphi$.

Abstract syntax tree construction. A compiler parses the user-specified MLTL formula into an Abstract Syntax Tree (AST) of subformulas, where each node in the AST handles one MLTL operator. Every node explicitly exposes the logical connection between the subformulas. R2U2 automatically synthesizes a runtime observer for every non-leaf node in the tree (i.e., for every MLTL operator) that takes an input execution sequence from child nodes and produces an output execution sequence for the parent node. R2U2 determines the satisfaction of the MLTL formula on an input trace by evaluating the output of the observers from the leaf nodes to the root of the tree. Fig. 1 shows the AST for MLTL formula $\varphi = (\Box_{[0,2]} a0) \wedge a1$, and Fig. 2 shows the corresponding compiled instructions. The leaf nodes are atomic proposition load operators that output an execution sequence that combines the values of the atomic propositions in the trace π with time index

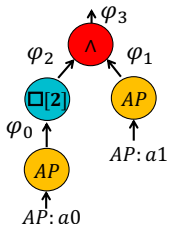

Fig. 1. AST for $\varphi = (\Box_{[0,2]} a0) \wedge a1$

τ. The output sequence of the root node corresponds to the verification result of the MLTL specification φ at every time stamp.

Abstract syntax tree evaluation. Let t_R be the current time stamp during runtime. Note that $t_R = 0$ indicates the start of execution. Table 1 shows the execution sequences generated by the observers when evaluating the MLTL formula $\varphi = (\Box_{[0,2]} a0) \wedge a1$ over the assignments shown in Fig. 3. The atomic proposition load operators capture hardware signals at the beginning of each time stamp (shown as event edges in Fig. 3) and output the corresponding execution sequence tuple. For example, at time stamp $t_R = 3$, $a1 = \mathsf{false}$ and $T_{\varphi_1} = (\mathsf{false}, 3)$. The verdicts in an execution sequence provide an evaluation of a future-time MLTL formula for every time stamp, sometimes by aggregating multiple consecutive verdicts from an input execution sequence. For example, at time stamp $t_R = 2$, $\langle T_{\varphi_0} \rangle = \langle (\mathsf{true}, 0), (\mathsf{true}, 1), (\mathsf{true}, 2) \rangle$ and $\langle T_{\varphi_2} \rangle = \langle (\mathsf{true}, 0) \rangle$; in other words, whether φ_2 holds at time index $\tau = 0$ cannot be known until $t_R = 2$. In this example, the trace π does not always satisfy φ; note that the tuple $(\mathsf{false}, 3)$ in T_φ (indicating that φ is false from time 2 to time 3) is produced at time stamp $t_R = 3$.

$$
\begin{aligned}
&Line0: \quad \varphi_0 \leftarrow load(a0) \\
&Line1: \quad \varphi_1 \leftarrow load(a1) \\
&Line2: \quad \varphi_2 \leftarrow \Box[0,2](\varphi_0) \\
&Line3: \quad \varphi_3 \leftarrow \wedge(\varphi_1, \varphi_2)
\end{aligned}
$$

Fig. 2. Instructions compiled from AST

MLTL \ t_R	0	1	2	3	4
$T_{\varphi_0}\,(\varphi_0 = load(a0))$	$(\top, 0)$	$(\top, 1)$	$(\top, 2)$	$(\top, 3)$	$(\top, 4)$
$T_{\varphi_1}\,(\varphi_1 = load(a1))$	$(\top, 0)$	$(\top, 1)$	$(\bot, 2)$	$(\bot, 3)$	$(\top, 4)$
$T_{\varphi_2}\,(\varphi_2 = \Box_{[0,2]}\varphi_0)$	-	-	$(\top, 0)$	$(\top, 1)$	$(\top, 2)$
$T_\varphi\,(\varphi = \varphi_1 \wedge \varphi_2)$	-	-	$(\top, 0)$	$(\top, 1)(\bot, 3)$	-

Table 1. Output of observers as an execution sequence ($\top \equiv \mathsf{true}$ and $\bot \equiv \mathsf{false}$) at time stamps t_R.

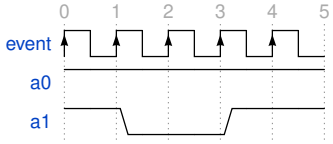

Fig. 3. Assignment to propositions $a0$ and $a1$ at event edges.

3 Model Predictive Runtime Verification (MPRV)

Overview. A high-level overview of MRPV is shown in Fig. 4. The future-time monitor utilizes current sensor data and model predictions to evaluate the satisfiability of formula φ by deadline d. The prediction's accuracy depends on the type of predictor and its modeling inaccuracies. We design MPRV generically; the user may choose *any* model predictor, weighing the trade-offs between accuracy and timing for the system-under-verification. The goal of MPRV is to make a decision on $\pi, i \models \varphi$ such that the supervisory controller can take the appropriate mitigation action by deadline d.

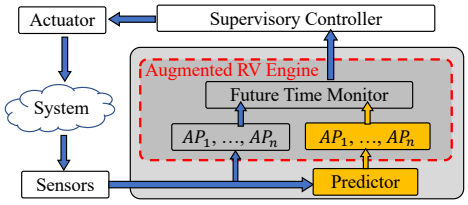

Fig. 4. High-level overview of MPRV. The blue and yellow arrows represent real and predicted data values, respectively.

Definition 9. *(Deadline)* Given an MLTL formula φ and trace π starting from time index $i \geq 0$, the deadline $d \in \mathbb{Z}$ is the number of time steps measured relative to i by which the satisfiability result of $\pi, i \models \varphi$ must be evaluated such that $0 \leq i + d \leq M$, where M denotes the end of the mission (i.e., $\pi, i \models \varphi$ cannot be evaluated before the mission begins or after the mission ends).

Definition 10. *(MLTL Semantics with Deadline)* MLTL Semantics with deadline d extends the MLTL Semantics in Definition 2. Trace $\hat{\pi}$ is defined as a trace of length $|\hat{\pi}| \geq |\pi|$ where $|\pi| \leq i+d$, the segment $\hat{\pi}[0,|\pi|-1] = \pi$, and the segment $\hat{\pi}[|\pi|,|\hat{\pi}|-1]$ may be populated using prediction in order to be able to make an evaluation decision by d. We recursively define $\pi,i,d \models \varphi$ (our decision based on $\hat{\pi}$ that trace π starting from time index $i \geq 0$ satisfies, or "models" MLTL formula φ by deadline d) as

- $\pi,i,d \models \mathsf{true}$,
- $\pi,i,d \models p$ for $p \in \mathcal{AP}$ iff $p \in \hat{\pi}(i)$,
- $\pi,i,d \models \neg\varphi$ iff $\hat{\pi},i \not\models \varphi$,
- $\pi,i,d \models \varphi_1 \wedge \varphi_2$ iff $\hat{\pi},i \models \varphi_1$ and $\hat{\pi},i \models \varphi_2$,
- $\pi,i,d \models \varphi_1 \, \mathcal{U}_{[lb,ub]} \, \varphi_2$ iff $|\hat{\pi}| \geq i+lb$ and $\exists j \in [i+lb,i+ub]$ such that $\hat{\pi},j \models \varphi_2$ and for every $k < j$ and $k \in [i+lb,i+ub]$ we have $\hat{\pi},k \models \varphi_1$

Illustrative Example. Consider the UAS example from Section 1. We want to deploy a parachute if the UAS's motors fail as a mitigation action to ensure a safe landing. For example, let's assume that if the MLTL formula $\varphi = \Box_{[0,6]} a \wedge \Diamond_{[0,8]} b$ evaluates to false, this indicates motor failure. Fig. 5 shows the runtime trace π. Here the worst-case propagation delay is eight (i.e., $\varphi.wpd = 8$), while the best-case propagation delay is zero (i.e., $\varphi.bpd = 0$). As a result, for $i = 3$, $\pi[3,11]$ is the maximum trace segment required to evaluate $\pi,i \models \varphi$. Let d be the deadline to trigger the deployment of the parachute, and let t_R be the current time stamp. MPRV monitors atoms a and b based on real and predicted data. If φ evaluates to false at $t_R \leq i+d$ based on real data, then the parachute is deployed at time t_R. However, there may not be enough real data at $t_R = i+d$ (i.e., because we do not have real data for the future), and the runtime monitor requires additional predicted data to make an on-deadline evaluation. In this case, MPRV incrementally queries the model predictor as needed to populate $\hat{\pi}(t_R+1)$, $\hat{\pi}(t_R+2)$, ... with predicted values of a and b until φ evaluates to true or false. Going back to our example in Fig. 5, if we let $i = 3$, $d = d_0 = -2$, and $t_R = i+d = 1$, the model predictor will populate $\hat{\pi}(2)$, $\hat{\pi}(3)$, ... , $\hat{\pi}(11)$ until the satisfiability of $\pi,i,d \models \varphi$ is determinable, and if we let $i = 3$, $d = d_1 = 5$, and $t_R = i+d = 8$, the model predictor will populate $\hat{\pi}(9)$, $\hat{\pi}(10)$, and $\hat{\pi}(11)$ until the satisfiability of $\pi,i,d \models \varphi$ is determinable. Parachute deployment is triggered if φ evaluates to false as $\hat{\pi}$ is incrementally populated by the predictor. Since $t_R \leq i+d$, MPRV ensures that the mitigation action is triggered before the deadline. The goal of MPRV is to produce a verdict evaluating whether $\pi,i,d \models \varphi$ holds. MPRV uses the minimum set of predicted variable evaluations needed to return a verdict; we determine this minimum set through *partial evaluation* of φ at time d.

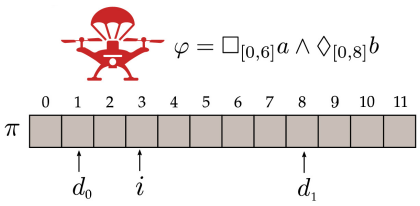

Fig. 5. Illustrative example.

Definition 11. *(Partial Evaluation)* The partial evaluation of an MLTL formula φ over trace π starting from time index $i \geq 0$, denoted as $\varphi|_{(\pi,i)}$, is the evaluation of φ based on the trace segment $\pi[i,|\pi|]$. There are two cases of partial evaluation to consider: 1) $\varphi|_{(\pi,i)} \in \{\mathsf{true},\mathsf{false}\}$, and 2) $\varphi|_{(\pi,i)} \notin \{\mathsf{true},\mathsf{false}\}$: in this case, $\varphi|_{(\pi,i)}$ returns a subformula produced by standard logic rewriting rules.

Going back to our illustrative example, we have $\varphi = (\Box_{[0,6]} a) \wedge (\Diamond_{[0,8]} b)$, $i = 0$, and $|\pi| = 6$. Let evaluations of a and b be $\pi_{\langle a,b \rangle} = [\langle 1,0 \rangle, \langle 1,1 \rangle, \langle 1,0 \rangle, \langle 1,1 \rangle, \langle 1,0 \rangle, \langle 1,0 \rangle]$. Since the

subformula $\Diamond_{[0,8]}b$ is satisfied at time stamp 1, we have $\varphi|_{(\pi,0)} = \Box_{[6,6]}a$. To evaluate the satisfiability of φ, the value for a must be predicted for time stamp 6.

Definition 12. *(Prediction Horizon)* Let $\hat{\pi}$ be a trace of length $|\hat{\pi}| \geq |\pi|$ where the segment $\hat{\pi}[0,|\pi|-1] = \pi$ and the segment $\hat{\pi}[|\pi|,|\hat{\pi}|-1]$ may be populated using prediction. The prediction horizon H_p is the length of the predicted segment of $\hat{\pi}$. The maximum prediction horizon is denoted by $max(H_p)$.

Definition 13. *(Model Predictive Runtime Verification (MPRV))* Given an MLTL formula φ, a trace π, and a deadline d, MPRV produces an execution sequence $\langle T_\varphi \rangle$ (as defined in Definition 8) such that each tuple T_φ with $T_\varphi.\tau \geq i$ is produced no later than $i+d$. It populates a predicted trace $\hat{\pi}$ such that $|\hat{\pi}| \geq |\pi|$, $|\pi| \leq i+d$, the segment $\hat{\pi}[0,|\pi|-1] = \pi$, and the segment $\hat{\pi}[|\pi|,|\pi|+H_p]$ by incrementally increasing prediction horizon H_p until $T_\varphi.v \in \{\mathsf{true},\mathsf{false}\}$ as follows:

- $d \geq \varphi.wpd$: $T_\varphi \equiv ((\pi,i \models \varphi),i)$ (prediction is not required)
- $d < \varphi.bpd$: $T_\varphi \equiv ((\pi,i,d \models \varphi|_{(\hat{\pi},i)}),i)$ (prediction is required)
- otherwise: $T_\varphi \equiv \begin{cases} T_\varphi \equiv ((\pi,i \models \varphi),i) \text{ if } \varphi|_{(\pi,i)} \in \{\mathsf{true},\mathsf{false}\} \\ \qquad\qquad \text{(prediction is not required)} \\ T_\varphi \equiv ((\pi,i,d \models \varphi|_{(\hat{\pi},i)}),i) \text{ otherwise} \\ \qquad\qquad \text{(prediction is required)} \end{cases}$

Lemma 1, 2, and 3 and corollaries 1 and 2 guarantee the correctness of MPRV and establish bounds on the produced execution tuples.

Lemma 1 (Minimum Trace Length). *Given an MLTL formula φ, a trace π, and a time $t \leq |\pi|$, MPRV is guaranteed to produce all of the execution sequence tuples T_φ such that $0 \leq T_\varphi.\tau \leq t - \varphi.wpd$. In other words, the shortest trace segment starting from $\pi[0]$ that we can use to guarantee $\pi,t-\varphi.wpd \models \varphi$ is $\pi[0,t]$. Fig. 6 provides a visualization of this lemma.*

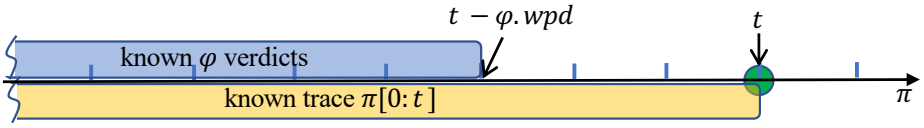

Fig. 6. Pictorial representation of Lemma 1 for *guaranteed* execution sequence elements.

Proof. The proof follows directly from induction on the structure of MLTL formula φ and Definitions 5, 6, and 7. □

Corollary 1 (Verdicts Guaranteed from Real Data). *Let t_R be the current time stamp. Given an MLTL formula φ and a trace segment $\pi[0,t_R]$, MPRV is guaranteed to produce from π (without prediction) all of the execution sequence tuples T_φ such that $0 \leq T_\varphi.\tau \leq t_R - \varphi.wpd$.*

Lemma 2 (Time Stamp Range of New Verdicts). *Let t_R be the current time stamp. Given an MLTL formula φ and a trace segment $\pi[0,t_R]$, if MPRV produces $T_\varphi.v$ from π at time stamp t_R, then $T_\varphi.\tau \in [t_R - \varphi.wpd, t_R - \varphi.bpd]$. That is, at time t_R, we know the time stamp range of any newly-produced execution sequence tuple. Fig. 7 provides a visualization of this lemma.*

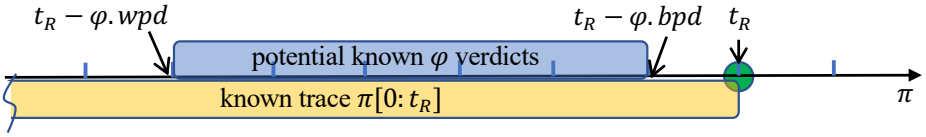

Fig. 7. Pictorial representation of Lemma 2 for range of T_φ produced at time t_R.

Proof. The proof follows directly from induction on the structure of MLTL formula φ and Definitions 5, 6, and 7. □

Corollary 2 (Verdicts from Real Data). *Let t_R be the current time stamp. Given an MLTL formula φ and a trace segment $\pi[0,t_R]$, MPRV can produce from π (without prediction) execution sequence tuples T_φ such that $0 \leq T_\varphi.\tau \leq t_R - \varphi.bpd$.*

Lemma 3 (Maximum Prediction Horizon). *Let t_R be the current time stamp. Given an MLTL formula φ, a trace segment $\pi[0,t_R]$, and a deadline d, to determine T_φ with $T_\varphi.\tau \geq i$ in at most d time steps from i (i.e., determine the satisfiability of $\pi,i,d \models \varphi$), the maximum prediction horizon $max(H_p)$ is bounded such that $max(H_p) = \varphi.wpd - d$. Fig. 8 provides a visualization of this lemma.*

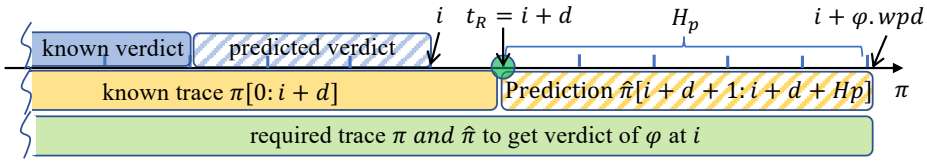

Fig. 8. Pictorial representation of Lemma 3. Note that in this case, $t_R = i + d$. Hashed regions represent predicted values, and solid regions represent known values.

Proof. If we let $t = i + \varphi.wpd$ in Lemma 1, then we are guaranteed to produce all T_φ with $T_\varphi.\tau \in [0,i]$. Given $\pi[0,t_R]$, by Definition 6, one needs to predict at most $\hat{\pi}[t_R+1, i+\varphi.wpd]$ to determine $\pi,i,d \models \varphi$. When prediction is required, $t_R = i + d$ (as depicted in Fig. 8). Thus $max(H_p) = i + \varphi.wpd - (i+d) = \varphi.wpd - d$. □

3.1 MPRV Algorithm

This section presents the generic MPRV algorithm (Algorithm 1). In Algorithm 1, the RV engine in lines 1, 6, and 10 can be *any* RV engine. We execute this algorithm when new sensor signals ($s_{real} \in \mathbb{R}^n$, where n is the number of sensors) are available at each time stamp (t_R). These sensor signals are converted into boolean values and represented in trace π. The sensor signals s_{real}, current time stamp t_R, and trace π are passed as inputs along with the specification details (i, d, and φ). The RV engine will update the current verdict based on the trace segment ($\pi[0,t_R]$) without prediction. If at deadline d we cannot assert true or false for $\varphi|_{\pi,i}$, the engine will continue partially evaluating the trace using prediction trace data $\hat{\pi}[0,i+d+1], \hat{\pi}[0,i+d+2], \ldots$ generated by $model\_predict$. Once true or false can be asserted for $\varphi|_{\hat{\pi},i}$, the verdict will be returned.

Theorem 1 (Correctness of MPRV Algorithm). *Given an MLTL formula φ, sensor signals s_{real}, trace π, deadline d, and a predictor function ($model\_predict$) that predicts the sensor signals at a future time step, the MPRV algorithm (Algorithm 1) computes a predicted trace $\hat{\pi}$ and the execution sequence tuple T_φ by using maximum real values and minimum predicted values required to evaluate $\pi,i,d \models \varphi$ such that $T_\varphi.v = \mathsf{true}$ iff $\pi,i,d \models \varphi$.*

Proof. We split the proof into two parts. (1) *Maximum real values and minimum predicted values:* MPRV uses all real data values from π up to deadline d (line 1–6). If before the deadline d, MPRV guarantees RV results without prediction (line 1). If at deadline d, it partially evaluates formula φ on trace π (line 2–6). If $\varphi|_{\pi,i}$ asserts true or false, it returns the result. Otherwise, $\varphi|_{\pi,i}$ returns a subformula with atomic propositions that require at least one time step of prediction to resolve the formula (follows from Definition 11). This subformula is checked iteratively at each subsequent time step of prediction until resolved, resulting in minimum predicted values (lines 7–10). (2) $T_\varphi.v = \mathsf{true} \leftrightarrow \pi,i,d \models \varphi$: (*only-if direction*) $\pi,i,d \models \varphi \rightarrow T_\varphi.v = \mathsf{true}$: If evaluating before deadline d, MPRV guarantees to return the current result of $\pi,i,d \models \varphi$. If evaluating at deadline d, MPRV guarantees that trace $\hat{\pi}$ contains enough data (follows from Lemma 3) to evaluate $\pi,i,d \models \varphi$, and then uses partial evaluation of the formula φ to return a final result based on available data (line 11). (*if direction*) $T_\varphi.v = \mathsf{true} \rightarrow \pi,i,d \models \varphi$: If evaluating before deadline d, MPRV guarantees to terminate and return the current result of $\pi,i,d \models \varphi$. If evaluating at deadline d, MPRV terminates and returns the final result (true or false) from the partial evaluation when the satisfiability of $\pi,i,d \models \varphi$ is determinable. Note that when at deadline d, MPRV returns verdicts only when the test condition of the while loop (line 7) evaluates to false. □

Algorithm 1: MPRV Algorithm (Def. 13)

Input: Signals: $s_{real} \in \mathbb{R}^n$; Current time stamp: t_R; Time index: i; Deadline: d; MLTL formula: φ; Trace: $\pi[0,t_R]$ derived from s_{real}
Output: Current verdict result for $\pi,i,d \models \varphi$

1 **if** $t_R < i+d$ **then return** $RV(\pi[0,t_R],i,\varphi)$; // Evaluating before deadline
2 **else** // Evaluating at deadline (i.e., $t_R = i+d$)
3 $\hat{\pi} \leftarrow \pi$; // initialize $\hat{\pi}$ with the real data
4 $t \leftarrow t_R$; // initialize t with current time stamp
5 $s \leftarrow s_{real}$; // initialize s with signals data
6 $\varphi|_{\hat{\pi},i} \leftarrow RV(\hat{\pi}[0,t],i,\varphi)$; // RV result of $\pi[0,t_R],i \models \varphi$
7 **while** $\varphi|_{\hat{\pi},i} \notin \{\mathsf{true},\mathsf{false}\}$ **do** // if prediction is needed, loop
8 $t \leftarrow t+1$; // look into next prediction step
9 $(s,\hat{\pi}[t]) \leftarrow model\_predict(s,t)$; // update s and $\hat{\pi}[t]$
10 $\varphi|_{\hat{\pi},i} \leftarrow RV(\hat{\pi}[0,t],i,\varphi)$; // RV result of $\hat{\pi}[0,t],i \models \varphi$
11 **return** $\varphi|_{\hat{\pi},i}$; // return true or false at deadline d

3.2 MPRV Implementation using R2U2

We implement the MPRV algorithm using the R2U2 RV engine framework. We first convert an MLTL formula φ into an AST offline (Section 2.2) and then topologically sort the nodes of the AST, denoted φ_{AST}, by arranging all child nodes before their parent nodes. This offline step creates a sequence of custom instructions that respects dependencies between instructions, as depicted in Fig. 2.

The R2U2-specific MPRV algorithm is defined in Algorithm 2. We optimize RV operations using the MLTL asynchronous observer algorithm in Algorithm 3, which does not require checking the entire trace when the trace is updated from $\pi[0,t]$ to $\pi[0,t+1]$ (refer to [48] for algorithm details). The verification results in the form of execution sequences for each node are stored in a data structure called a Shared Connection Queue (SCQ) [34], which is a circular buffer for storing the execution sequence generated by each node of φ_{AST}. Since the circular buffer overwrites data in a circular way, only a segment of the execution sequence

is kept in each SCQ. To store the necessary real and predicted data, we size the SCQs per the procedure given in Section 3.3. Before prediction begins, we must cache the local variables of the SCQ (i.e., read and write pointers). The predictive RV phase of the algorithm executes until $\varphi|_{\hat{\pi},i}$ evaluates to true or false, which is evaluated by reading the SCQ of the root node (denoted as $\varphi.Queue$). Finally, we restore the previously cached local variables of the SCQs since the predicted values will now be outdated for the next time step. All of these tasks should be completed in one sensor sampling period to allow RV to keep pace with real-time. The algorithm terminates once $\varphi|_{\hat{\pi},i}$ evaluates to true or false, and the verdict is returned.

Algorithm 2: R2U2-specific algorithm for MPRV (Def. 13)

Input: Signals: $s_{real} \in \mathbb{R}^n$; Current time stamp: t_R; Time index: i; Deadline: d; MLTL formula: φ; Trace: $\pi[0,t_R]$ derived from s_{real}
Output: Current verdict result for $\pi,i,d \models \varphi$

/* Update φ_{AST} for current time stamp t_R */
for *Node g from topologically sorted node list of* φ_{AST} **do**
  $RV\_node\_one\_step(\pi,t_R,g)$; // Algorithm 3
if $t_R < i+d$ **then return** $read(\varphi.Queue)$; // Evaluating before deadline
else // Evaluating at deadline (i.e., $t_R = i+d$)
  /* store original RV engine state */
  for *Node g from topologically sorted node list of* φ_{AST} **do**
    Store Node g's local variables; // read/write pointers
  $\hat{\pi} \leftarrow \pi$; // initialize $\hat{\pi}$ with the real data
  $t \leftarrow t_R$; // initialize t with current time stamp
  $s \leftarrow s_{real}$; // initialize s with signals data
  $\varphi|_{\hat{\pi},i} \leftarrow read(\varphi.Queue)$; // RV result of $\pi[0,t_R],i \models \varphi$
  while $\varphi|_{\hat{\pi},i} \notin \{$true,false$\}$ **do** // if prediction is needed, loop
    $t \leftarrow t+1$; // look into next prediction step
    $(s,\hat{\pi}[t]) \leftarrow model\_predict(s,t)$; // update s and $\hat{\pi}[t]$
    for *Node g from topologically sorted node list of* φ_{AST} **do**
      $RV\_node\_one\_step(\hat{\pi},t,g)$; // Algorithm 3
    $\varphi|_{\hat{\pi},i} \leftarrow read(\varphi.Queue)$; // RV result of $\hat{\pi}[0,t],i \models \varphi$
  /* restore original RV engine state */
  for *Node g from topologically sorted node list of* φ_{AST} **do**
    Restore Node g's local variables; // read/write pointers
  return $\varphi|_{\hat{\pi},i}$; // return true or false at deadline d

3.3 Memory and Time Analysis of R2U2 Implementation

Memory Utilization without Prediction. A system utilizing the R2U2 framework needs memory to store the instructions and SCQs. For each node g in φ_{AST}, one queue is needed (denoted as $g.Queue$). Because the SCQ overwrites data in a circular way, the queues can be viewed as sliding windows. Each sliding window takes a segment of the execution sequence generated from the corresponding child node(s). Each sliding window must store the necessary data to evaluate the satisfiability of $\pi,i \models \varphi$. The time stamp difference between the inputs of child nodes can cause an input tuple from a child node g with a higher time index τ to wait for input(s) from g's sibling(s). That is, any newly generated output from g will be stalled in g's output queue until g's siblings have the same τ. We show that the required new trace data for generating the matched time indices of the two input queues is bounded, and we use bpd and wpd to prove these bounds in Lemma 4. In [25], a similar approach is used for

Algorithm 3: Run RV for one time stamp on an AST node g; update g's Queue with the execution sequence $\langle T_g \rangle$, which will be propagated up as the input of g's parent node(s).

1 **function** $RV\_node\_one\_step(\pi,i,g)$ **is**
  Input: Trace: π; Time index: i; Node: g;
2 **if** *g is an $\mathcal{AP}$ operator* **then**
    /* record the value of the atomic proposition */
3 **if** $g \in \pi[i]$ **then**
4 $g.Queue.write((\mathsf{true},i))$; // write $\langle T_g \rangle$
5 **else**
6 $g.Queue.write((\mathsf{false},i))$; // write $\langle T_g \rangle$
7 **else**
8 $\langle T_g \rangle \leftarrow$ evaluate MLTL operator g; // Algorithms 3--6 from [34]
9 $g.Queue.write(\langle T_g \rangle)$; // write $\langle T_g \rangle$

estimating resource usage, where they call the delay a *horizon*; however, they did not consider the best-case propagation delay or prediction.

Lemma 4 (Memory Usage). *Let φ_{AST} be the abstract syntax tree generated from the MLTL formula φ as explained in Section 2.2. For any node g from φ_{AST}, let $\mathbb{S}_g$ be the set of all sibling nodes of g. Let $g.Queue$ be the queue for the execution sequence generated by the runtime observer of g. Then the maximum memory usage of $g.Queue$ is given by* $g.Queue.size \leq max(max\{s.wpd \mid s \in \mathbb{S}_g\} - g.bpd, 0) + 1$.

Proof. Refer to the appendix.

MPRV Memory Utilization. We use $g_{MPRV}.Queue.size$ to represent the size of $g.Queue$ when using MPRV. To prevent overwriting the original SCQ content with predicted data, we need $max(H_p)$ (as defined in Lemma 3) extra entries in $g.Queue$; we must prevent overwriting the original content so we can restore operations after prediction. Therefore, $g_{MPRV}.Queue.size = g.Queue.size + max(H_p)$. Algorithm 4 of the appendix further details how to determine the memory usage for each SCQ.

Each queue entry stores a tuple of a verdict and a time index. We use one bit to represent the verdict, and if we let $max(\tau)$ be the length of the mission in terms of the number of time stamps, then we need $\lceil \log_2 max(\tau+1) \rceil)$ bits to represent the time stamp. The following equation gives the total memory size (in bits):

$$total\ memory\ size = (1 + \lceil \log_2 max(\tau+1) \rceil) \times \sum_{g \in \varphi_{AST}} g_{MPRV}.Queue.size$$

Worst-case Execution Time (WCET). The MPRV worst-case execution time splits into two parts: the model prediction ($WCET_{MODEL}$) and RV on the trace $\hat{\pi}$ ($WCET_{RV}$). This section analyzes $WCET_{RV}$ from line 1 to line 19 except for line 13 in Algorithm 2. The WCET of line 13 are equivalent to $WCET_{MODEL}$, which depends on the chosen model predictor (refer to Section 5.1 for examples).

Lemma 5 (Worst-case Execution Time of RV ($WCET_{RV}$)). *Given an MLTL formula φ and the output queue size of each node in φ_{AST}, let $g.input\_Queue$ be the sum of queue sizes of direct child nodes of g, t_{exe} be the execution time for an MLTL operator node to consume one element of the input execution sequence (Algorithm 3), t_{store} be the*

store operation time (Algorithm 2 lines 5–6), and $t_{restore}$ be the restore operation time (Algorithm 2 lines 17–18). Then the worst-case execution time is given by $WCET_{RV}(\varphi) = \sum_{g \in \varphi_{AST}} (g.input\_Queue.size \times t_{exe} + t_{store} + t_{restore})$.

Proof. Refer to the appendix.

4 Quadcopter Case Study

We demonstrate the utility of MPRV for a UAS quadcopter by simulating a non-linear quadcopter model ([9, 13])[4] whose trajectory is controlled via a linear Model Predictive Control (MPC) controller; refer to [40] for MPC algorithm details. The upper portion of Fig. 9 shows the actual position of the quadcopter compared to the planned trajectory chosen by MPC. Note that we use the same model predictor for control and for MPRV in order to save hardware resources.

Let π be a trace over atomics $a0$ and $a1$ at each time index. Table 2 specifies the atomic propositions that map system requirements to atomics $a0$ and $a1$. The specification $\varphi = (\Box_{[5]} a0) \mathcal{U}_{[10]} a1$ in Fig. 9 limits the quadcopter's vertical velocity for a future five time steps if the quadcopter keeps diverging from the planned trajectory for the next ten time steps. This specification is designed to prevent the quadcopter from crashing into the ground at high speed when diverged far off course. If MPRV returns a false verdict for φ, an appropriate mitigation strategy can be triggered (e.g., deploying a parachute).

Table 2. Quadcopter Atomic Propositions

Atomic	Atomic Proposition
$a0$	Magnitude of trajectory error $(z_{ref} - z) \leq 0.12$m
$a1$	Vertical speed $(\dot{z_k}) \geq -0.9$ m/s

In the lower portion of Fig. 9, if the verdict is true, then $\pi, t \models \varphi$, where t is the corresponding time-axis value. Due to the nature of future time asynchronous observers, the evaluation of $\pi, t \models \varphi$ is not known until time t or later. We assume the required controller actuation update rate and sensor sampling rate are 50 Hz (.02 seconds); consequently, MPRV is run every 0.02 seconds. Fig. 9 compares the responsiveness of $\pi, i, d \models (\Box_{[5]} a0) \mathcal{U}_{[10]} a1$ using MPRV with a deadline $d = -5$ steps (0.1s before violation) and $d = 10$ steps (0.2s after violation). Given the formula has $\varphi.wpd = 15$, we compute the maximum prediction horizon using Lemma 3. This results in $max(H_p) = 20$ when $d = -5$, and $max(H_p) = 5$ when $d = 10$. The green and orange bars are associated with a 20 and 5 step prediction horizon, respectively. The left boundary of the bar is the time when MPRV detects the formula becoming false, and right boundary is the time when a false verdict is detected without prediction. Note that for $d = -5$, the green bars maintain at least 5 time steps ahead of the corresponding false verdict (marked by red cross). Similarly, when $d = 10$, the orange bars maintain at least 10 steps after the corresponding false verdict.

In summary, the earlier the deadline, the longer the maximum prediction horizon and the earlier conclusive results are obtained. These results are in alignment with what one would intuitively expect. Note that real-world sensors and imperfect knowledge of the model limit how far one can effectively estimate the future, and the larger the prediction horizon, the more the prediction is prone to inaccuracies. While we did not account for inaccuracies in our

[4] Specific model parameters for the quadcopter model were obtained from [26] and [42]

implementation, methods have been developed to robustify MPC against model uncertainties and disturbances [12, 16].

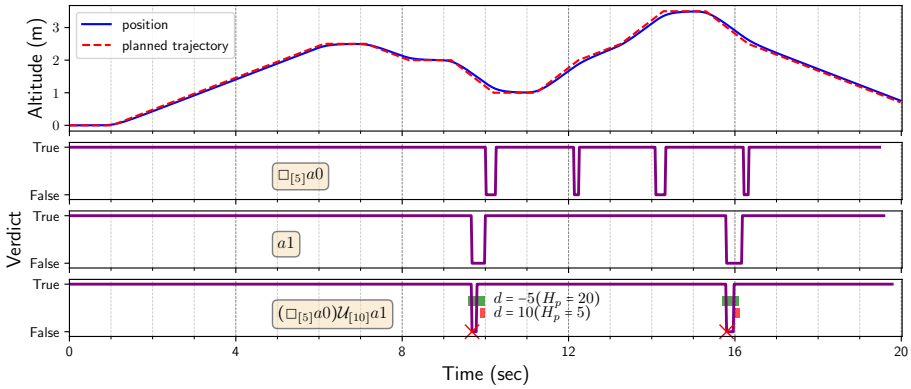

Fig. 9. The position of a quadcopter resulting from MPC and verification results of three formulas. The red $\times$ indicates motor failure. The left end of each green (orange) bar is the formula-violation detection time using MPRV with a prediction horizon of H_p. The right end of each green (orange) bar is the formula-violation detection time using RV without prediction.

5 Analysis of Hardware Implementation

Our methodology is applicable across a wide range of systems and applications. When targeting a given system, users would want to deploy their own mission-specific MLTL formulas and model predictor. Users can select *any* model predictor, weighing the trade-offs between accuracy and performance. Currently, many model predictors exist in the literature. The most common method for developing system models for prediction is to derive a differential-algebraic model by analyzing the physical system's dynamics [15, 18]. Another common modeling method is system identification, which derives the model of the system by observing the system's inputs and corresponding outputs [28, 44]. Data-driven modeling via machine learning [59] or simulation-based modeling [10, 33] are a few more popular modeling techniques. Note that MPRV allows for simpler low-fidelity system models for the prediction of individual variables, and there exists a trade-off in terms of resources and computing time between low-fidelity and high-fidelity models. We provide details on determining if MPRV is feasible for an embedded platform's resource constraints and mission's performance requirements in this section.

5.1 FPGA Implementation of Model Predictive Runtime Verification (MPRV)

For experimental evaluation of our implementation, we target a modest-sized Xilinx FPGA (ZYNQ 7020) [1] and use the Vivado 2019.2 tool-chain [2] to synthesize our MPRV design. The resource usage of our quadcopter case study is shown in Fig. 10. Note that our implementation of MPRV and MPC [62] are both modular by design; several software-configurable registers allow the user to modify MLTL formulas and the MPC's control algorithm on-the-fly without having to re-synthesize the hardware. Additionally, the MPC hardware design has a trade-off between performance and resource utilization; we have maximized the MPC design to fill up the remaining resources the augmented RV engine (R2U2 with MPRV) did not use. One example of this performance and resource trade-off is in the MPC's matrix-vector multiplier (MVM). The larger the MVM, the faster the MPC can handle larger matrices, i.e.,

the longer the MPC's prediction horizon or the larger the number of the state variables N. For our quadcopter example ($N=12$), Fig. 10 shows that our augmented RV engine requires additional BRAMs (2.5% of the 280 available 18Kb BRAMs) for instructions, local variables (defined in [34]), and data transmission FIFOs. When maximizing the MPC design to fill the rest of the BRAM resources, we can reach a prediction horizon $H_p=17$.

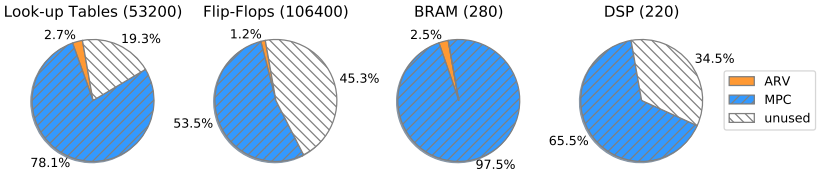

Fig. 10. Percentage of resource usage by augmented RV engine (ARV), MPC, and unused resource. The total amount of each resource available on chip is included inside the parentheses. For the BRAM, the blue corresponds to a system of 12 state variables (N) with $H_p=17$.

Recall that our implementation uses an SCQ buffer for each node in φ_{AST} as described in Section 3.2. Given an MLTL formula, sizing SCQs per the procedure given in Section 3.3 guarantees SCQs never overflow. We chose a time stamp width of 31-bits and a total SCQ size of 512*32 bits, which consumes one 18Kb block RAM (BRAM) [1]. Such memory is sufficient for the example formulas with a corresponding $max(H_p)$ in Table 3. We also show the $WCET_{RV}$ of Algorithm 2 for each formula when predicting with $max(H_p)$. Example execution time of the MPC controller ($WCET_{MODEL}$) is shown in Fig. 11.

Table 3. Examples of MLTL formulas that can be supported with one 18Kb BRAM in Fig. 10

MLTL Formula	$max(H_p)$	$WCET_{RV}$
$a0$	509	5.28μs
$\Box_{[0,10]}\ a0$	248	47.87μs
$\Box_{[0,10]}\ a0\,\mathcal{U}_{[0,10]}\ a1$	115	67.87μs
$\Box_{[0,10]}\ a0 \vee \Box_{[0,10]}\ a1$	90	73.99μs
$(a0\,\mathcal{U}_{[0,5]}a1)\vee(a2\,\mathcal{U}_{[0,10]}a3)\vee(\Box_{[0,10]}\ a4)$	41	82.16μs

Hardware architectures exist that support online configuration of MPC for different systems, e.g., [62]. Assuming the embedded hardware-based MPC controller of [62] is acting as the model predictor of our MPRV architecture, we show in Fig. 11 our approach scales feasibly across a wide range of systems. To give a sense of the range of system complexity our approach can support, we highlight a point-mass system ($N=2$), a quadcopter [35] ($N=12$), and even a reduced order model of a fusion machine [14] ($N=$ 2700 for full model, but $N=30$ for reduced model at an accuracy loss of .1%). The left-hand side of Fig. 11 depicts how memory usage scales with N and H_p, while the right-hand side depicts how execution time scales. Additionally, the right-hand side of Fig. 11 illustrates how one would take into account a system's required sensor update rate to determine how far into the future we can predict (H_p). This is accomplished using the two horizontal dashed lines located at .01 seconds (10ms) and at .02 seconds (20ms). For example, if a system requires a sensor update rate of .01 seconds, then for a quadcopter system model with $N=12$ one could have $H_p=10$ or $H_p=20$, but not $H_p=50$ as computation requires more than .01 seconds. If this same sized system required a sensor update rate of .02 seconds, $H_p=50$ would be computed fast enough to meet system dynamic constraints.

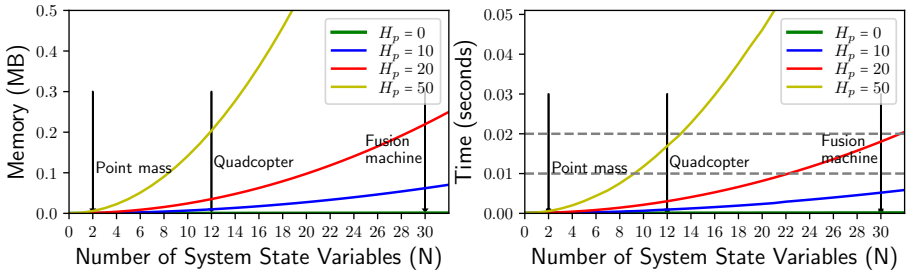

Fig. 11. Memory and execution time of MPC from [62] with FPGA clock frequency of 130MHz.

In summary, Fig. 11 can help users determine early in a design process whether an embedded platform's memory and computing capabilities are sufficient to support MPRV and reason about system performance and resource trade-offs. For example, in our Quadcopter case study we needed a prediction horizon of 5 and 20. Regarding computing speed, our FPGA implementation can handle both cases; however, in terms of memory, the largest H_p we can support is 17 for the quadcopter model. Thus, either $H_p = 17$ would need to be sufficient for the mission, or a slightly larger FPGA would be required.

6 Conclusion and Future Work

The definition and general algorithm for MPRV are *extensible*; while we have chosen to exemplify MPRV using MLTL properties verified in the R2U2 engine with an MPC controller for a quadcopter, all of these choices can be changed or extended. Our results are promising regarding MPRV's utility for improving RV responsiveness in real-life scenarios. In particular, our implementation paves the way for better mitigation of faults by enabling evaluation of future-time requirements with enough time to trigger mitigation actions during system runtime. There is now a basis to extend MPRV to other logics (e.g., MTL [4], STL [41], MLTLM [29]), create implementations that build on other RV engines, and plug in different model predictors for when prediction steps are necessary. Investigating the trade-offs between resource demands, performance, and accuracy of different model predictors maps a valuable landscape for future MPRV design decisions.

We design MPRV to use maximum real values and minimum predicted values presuming the former is more accurate. However, this is not always the case as both real data and predicted data have their own associated error distribution (i.e., sensor noise, model parameter variation, etc.). To robustify MPRV against inaccurate distribution-valued signals, future work will include investigating techniques for robust satisfaction of MLTL specifications similar to [22, 45]. While we focused on a priori known deadlines, MPRV is capable of evaluating specifications with dynamically-defined deadlines; we leave this for future work. An application for requiring a dynamically-defined deadline would be an autonomous vehicle's braking system; the time required to come to a complete stop dynamically changes based on the current velocity of the vehicle. Additionally, [23] provides an interesting direction for MPRV. This work focuses on using multiple model predictors for different components in a complex system of systems (e.g., a robotic system), but they assume the system has a shared global clock. This is not always the case, and current work has extended R2U2 to monitor different timescales through MLTLM specifications [29]. In future work, we also plan to revisit case studies of running R2U2 on real UAS [17, 27, 48, 49, 52–54] to further explore MPRV.

Appendix

Proof of Lemma 4

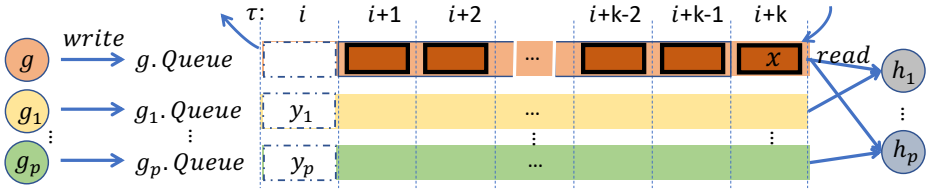

Fig. 12. Analysis of maximum execution sequence time stamp mismatch between sibling nodes. The squares are data content of $g.Queue$ and all other queues from g's sibling nodes (g_1,...,g_p) that share the same parent nodes (any of h_1, h_2, ..., h_p) with g. In $g.Queue$, each execution tuple is labeled by the execution sequence's τ value ranging from i to $i+k$. x, y_1,...,y_p are the latest execution sequence tuples with the biggest time stamp in the corresponding queue.

Proof. Fig. 12 shows the pictorial representation of the execution sequence tuples inside $g.Queue$ and g's sibling nodes at the current time stamp t_R. Let the nodes g and g_1 be the child nodes of node h_1. The execution sequence tuples with the same time index τ are consumed/popped from $g.Queue$ and $g_1.Queue$ to produce the input tuples of h_1. Once these tuples are consumed by their parent node, they are dropped from the queue (marked in dashed boxes). Let x be the latest execution sequence tuple in $g.Queue$ with time index $i+k$. Let y_1 be the latest execution sequence tuple in $g_1.Queue$ with time index i, and let y_1 to have been consumed by $h_1.Queue$. Since y_1 was consumed at t_R and was not stalled by $g.Queue$, y_1 must have been produced at t_R-1. According to Lemma 2, when $\pi[0,t_R-1]$ is known, if a new execution tuple T_{g_1} is inserted into $g_1.Queue$, then $T_{g_1}.\tau \in [t_R-1-g_1.wpd, t_R-1-g_1.bpd]$. Therefore, $y_1.\tau \geq t_R-1-g_1.wpd$. Similarly, when $\pi[0,t_R]$ is known, if there are new execution sequence tuples T_g inserted into $g.Queue$, then $T_g.\tau \in [t_R-g.wpd, t_R-g.bpd]$. Therefore, $x.\tau \leq t_R-g.bpd$. The maximum value of k is $max(x.\tau)-min(y_1.\tau) = t_R-g.bpd-(t_R-1-g_1.wpd) = g_1.wpd-g.bpd+1$. The maximum number of elements to be stored in $g.Queue$ is bounded by $g_1.wpd-g.bpd+1$, or 1 if the difference is non-positive. The same argument follows for g_2,...,g_p. Therefore, we have $g.Queue$ bounded by $s.wpd-g.bpd+1$ for $s \in \mathbb{S}_g = \{g_1,...,g_p\}$, or $g.Queue.size \leq max(max\{s.wpd \mid s \in \mathbb{S}_g\}-g.bpd,0)+1$. □

Algorithm for sizing SCQs

We first derive the topologically sorted collection of all nodes from φ_{AST}. Second, we compute and record the *bpd* and *wpd* of each node in this collection sequentially (lines 1–2). Finally, we compute the output queue size for each subformula (lines 3–5).

Algorithm 4: Compute g's queue size for all node g in the R2U2 AST to optimize for storing a combination of real and predicted data for MPRV

Input: MLTL: φ; Prediction Horizon: H_p

1 **for** *Node g from topologically sorted node list of* φ_{AST} **do**
2 | Compute $g.bpd$ and $g.wpd$; `// Definition 7`
3 **for** *Node g in* φ_{AST} **do**
4 | $g.Queue.size \leftarrow max(max\{s.wpd \mid s \in \mathbb{S}_g\}-g.bpd,0)+1$; `// Lemma 4`
5 | $g_{MPRV}.Queue.size \leftarrow g.Queue.size+H_p$; `// final queue size`

Proof of Lemma 5

Proof. The $WCET_{RV}$ is the sum of the worst-case execution time of all the nodes in φ_{AST}. The time stamp of the execution sequence tuple for a unary operator node will increase when we write to $g.Queue$ in Algorithm 3. For binary operator node h, at least one of the inputs' $(g1, g2)$ time stamp will increase. The increase is bounded by the total size of the $g1.Queue$ and $g2.Queue$ (from Lemma 4). Let $h.input\_Queue.size = g1.Queue.size + g2.Queue.size$, then the $WCET_{RV}$ for node h is $h.input\_Queue.\ size \times t_{exe}$. The total execution time for all nodes in the φ_{AST} is $\sum_{g \in \varphi_{AST}} g.input\_Queue.size \times t_{exe}$. When we combine this with the store time (t_{store}) and restore time ($t_{restore}$) for each node, $WCET_{RV}(\varphi) = \sum_{g \in \varphi_{AST}} (g.input\_Queue.size \times t_{exe} + t_{store} + t_{restore})$. □

References

1. Zynq-7000 soc data sheet: Overview. https://www.xilinx.com/support/documentation/data_sheets/ds190-Zynq-7000-Overview.pdf (2018)
2. Vivado design suite user guide. https://www.xilinx.com/support/documentation/sw_manuals/xilinx2019_2/ug973-vivado-release-notes-install-license.pdf (2019)
3. Adolf, F.M., Faymonville, P., Finkbeiner, B., Schirmer, S., Torens, C.: Stream runtime monitoring on uas. In: RV. pp. 33–49. Springer (2017)
4. Alur, R., Henzinger, T.A.: Real-time Logics: Complexity and Expressiveness. In: LICS. pp. 390–401. IEEE (1990)
5. Aurandt, A., Jones, P.H., Rozier, K.Y.: Runtime verification triggers real-time, autonomous fault recovery on the cysat-i. In: NASA Formal Methods Symposium. pp. 816–825. Springer (2022)
6. Babaee, R., Ganesh, V., Sedwards, S.: Accelerated learning of predictive runtime monitors for rare failure. In: International Conference on Runtime Verification. pp. 111–128. Springer (2019)
7. Babaee, R., Gurfinkel, A., Fischmeister, S.: $\mathcal{P}$revent: A predictive run-time verification framework using statistical learning. In: SEFM. pp. 205–220. Springer (2018)
8. Babaee, R., Gurfinkel, A., Fischmeister, S.: Predictive run-time verification of discrete-time reachability properties in black-box systems using trace-level abstraction and statistical learning. In: RV. pp. 187–204. Springer (2018)
9. Balas, C., Whidborne, J., of Engineering, C.U.S.: Modelling and Linear Control of a Quadrotor. Theses 2007, Cranfield University, School of Engineering (2007), https://books.google.com/books?id=7PIYyAEACAAJ
10. Banaei, M.R., Alizadeh, R.: Simulation-based modeling and power management of all-electric ships based on renewable energy generation using model predictive control strategy. ITSM **8**(2) (2016)
11. Bartocci, E., Deshmukh, J., Donzé, A., Fainekos, G., Maler, O., Ničković, D., Sankaranarayanan, S.: Specification-based monitoring of cyber-physical systems: a survey on theory, tools and applications. Lectures on Runtime Verification: Introductory and Advanced Topics pp. 135–175 (2018)
12. Bemporad, A., Morari, M.: Robust model predictive control: A survey. In: Robustness in identification and control, pp. 207–226. Springer (2007)
13. Bolandi, H., Rezaei, M., Mohsenipour, R., Nemati, H., Smailzadeh, S.M.: Attitude control of a quadrotor with optimized pid controller. Intelligent Control and Automation **4**, 335–342 (2013)
14. Bonotto, M., Bettini, P., Cenedese, A.: Model-order reduction of large-scale state-space models in fusion machines via krylov methods. IEEE Transactions on Magnetics **53**(6), 1–4 (2017)
15. Brown, R.G., Hwang, P.Y.C.: Introduction to Random Signals and Applied Kalman Filtering with MATLAB Exercises. John Wiley & Sons, Inc., 4th edn. (2012), iSBN-13 978-0-470-60969-9

16. Bujarbaruah, M., Rosolia, U., Stürz, Y.R., Borrelli, F.: A simple robust mpc for linear systems with parametric and additive uncertainty. In: 2021 American Control Conference (ACC). pp. 2108–2113. IEEE (2021)
17. Cauwels, M., Hammer, A., Hertz, B., Jones, P., Rozier, K.Y.: Integrating Runtime Verification into an Automated UAS Traffic Management System. In: DETECT. Springer, L'Aquila, Italy (September 2020)
18. Chen, C.: Linear System Theory and Design. Oxford University Press, Inc., 3rd edn. (1999), iSBN-13 978-0-19-511777-6
19. Cimatti, A., Tian, C., Tonetta, S.: Assumption-based runtime verification with partial observability and resets. In: International Conference on Runtime Verification. pp. 165–184. Springer (2019)
20. Dabney, J.B., Badger, J.M., Rajagopal, P.: Adding a verification view for an autonomous real-time system architecture. In: Proceedings of SciTech Forum. p. Online. 2021-0566, AIAA (January 2021). https://doi.org/https://doi.org/10.2514/6.2021-0566
21. D'Angelo, B., Sankaranarayanan, S., Sanchez, C., Robinson, W., Finkbeiner, B., Sipma, H.B., Mehrotra, S., Manna, Z.: Lola: runtime monitoring of synchronous systems. In: TIME. pp. 166–174 (2005)
22. Fainekos, G.E., Pappas, G.J.: Robustness of temporal logic specifications for continuous-time signals. Theoretical Computer Science **410**(42), 4262–4291 (2009)
23. Ferrando, A., Cardoso, R.C., Farrell, M., Luckcuck, M., Papacchini, F., Fisher, M., Mascardi, V.: Bridging the gap between single-and multi-model predictive runtime verification. Formal Methods in System Design pp. 1–33 (2022)
24. Ferrando, A., Delzanno, G.: Incrementally predictive runtime verification. In: CILC. pp. 92–106 (2021)
25. Finkbeiner, B., Kuhtz, L.: Monitor circuits for ltl with bounded and unbounded future. In: RV. pp. 60–75. Springer (2009)
26. Förster, J.: System identification of the crazyflie 2.0 nano quadrocopter (2015)
27. Geist, J., Rozier, K.Y., Schumann, J.: Runtime Observer Pairs and Bayesian Network Reasoners On-board FPGAs: Flight-Certifiable System Health Management for Embedded Systems. In: RV. vol. 8734, pp. 215–230. Springer-Verlag (September 2014)
28. Greblicki, W.: Continuous-time hammerstein system identification from sampled data. TAC **51**(7), 1195–1200 (2006)
29. Hariharan, G., Kempa, B., Wongpiromsarn, T., Jones, P.H., Rozier, K.Y.: Mltl multi-type (mltlm): A logic for reasoning about signals of different types. In: International Workshop on Numerical Software Verification, Workshop on Formal Methods for ML-Enabled Autonomous Systems. pp. 187–204. Springer (2022)
30. Heffernan, D., Macnamee, C., Fogarty, P.: Runtime verification monitoring for automotive embedded systems using the iso 26262 functional safety standard as a guide for the definition of the monitored properties. IET Software **8**(5), 193–203 (October 2014). https://doi.org/10.1049/iet-sen.2013.0236
31. Hertz, B., Luppen, Z., Rozier, K.Y.: Integrating runtime verification into a sounding rocket control system. In: NASA Formal Methods Symposium. pp. 151–159. Springer (2021)
32. Jaksic, S., Bartocci, E., Grosu, R., Kloibhofer, R., Nguyen, T., Nickovic, D.: From signal temporal logic to FPGA monitors. In: MEMOCODE. pp. 218–227 (Sept 2015)
33. Kapinski, J., Deshmukh, J.V., Jin, X., Ito, H., Butts, K.: Simulation-based approaches for verification of embedded control systems: An overview of traditional and advanced modeling, testing, and verification techniques. Control Systems Magazine **36**(6), 45–64 (2016)
34. Kempa, B., Zhang, P., Jones, P.H., Zambreno, J., Rozier, K.Y.: Embedding Online Runtime Verification for Fault Disambiguation on Robonaut2. In: Proceedings of the 18th International Conference on Formal Modeling and Analysis of Timed Systems (FORMATS). pp. 196–214. Lecture Notes in Computer Science (LNCS), Springer, Vienna, Austria (September 2020), `http://research.temporallogic.org/papers/KZJZR20.pdf`

35. Kurak, S., Hodzic, M.: Control and estimation of a quadcopter dynamical model. Periodicals of Engineering and Natural Sciences **6**(1), 63–75 (2018)
36. Leucker, M.: Sliding between model checking and runtime verification. In: RV. pp. 82–87. Springer (2012)
37. Li, J., Vardi, M.Y., Rozier, K.Y.: Satisfiability checking for mission-time LTL. In: International Conference on Computer Aided Verification. pp. 3–22. Springer (2019)
38. Lindemann, L., Qin, X., Deshmukh, J.V., Pappas, G.J.: Conformal prediction for stl runtime verification. arXiv preprint arXiv:2211.01539 (2022)
39. Lu, H., Forin, A.: The Design and Implementation of P2V, An Architecture for Zero-Overhead Online Verification of Software Programs. Tech. Rep. MSR-TR-2007-99, Microsoft Research (August 2007)
40. Maciejowski, J.M.: Predictive control: with constraints. Pearson education (2002)
41. Maler, O., Nickovic, D.: Monitoring temporal properties of continuous signals. In: Formal Techniques, Modelling and Analysis of Timed and Fault-Tolerant Systems, pp. 152–166. Springer (2004)
42. McInerney, I.: Development of a multi-agent quadrotor research platform with distributed computational capabilities. Ph.D. thesis, Iowa State University (2017)
43. Meredith, P.O., Jin, D., Griffith, D., Chen, F., Roşu, G.: An overview of the MOP runtime verification framework. STTT **14**(3), 249–289 (2012)
44. Naung, Y., Schagin, A., Oo, H.L., Ye, K.Z., Khaing, Z.M.: Implementation of data driven control system of dc motor by using system identification process. In: EIConRus. pp. 1801–1804 (2018)
45. Pant, Y.V., Abbas, H., Quaye, R.A., Mangharam, R.: Fly-by-logic: Control of multi-drone fleets with temporal logic objectives. In: 2018 ACM/IEEE 9th International Conference on Cyber-Physical Systems (ICCPS). pp. 186–197. IEEE (2018)
46. Pellizzoni, R., Meredith, P., Caccamo, M., Rosu, G.: Hardware runtime monitoring for dependable COTS-based real-time embedded systems. RTSS pp. 481–491 (2008)
47. Pinisetty, S., Jéron, T., Tripakis, S., Falcone, Y., Marchand, H., Preoteasa, V.: Predictive runtime verification of timed properties. Journal of Systems and Software **132**, 353–365 (2017)
48. Reinbacher, T., Rozier, K.Y., Schumann, J.: Temporal-logic based runtime observer pairs for system health management of real-time systems. In: Proceedings of the 20th International Conference on Tools and Algorithms for the Construction and Analysis of Systems (TACAS). Lecture Notes in Computer Science (LNCS), vol. 8413, pp. 357–372. Springer-Verlag (April 2014)
49. Rozier, K.Y., Schumann, J., Ippolito, C.: Intelligent Hardware-Enabled Sensor and Software Safety and Health Management for Autonomous UAS. Technical Memorandum NASA/TM-2015-218817, NASA (May 2015)
50. Rozier, K.Y., Schumann, J.: R2U2: Tool Overview. In: Proceedings of International Workshop on Competitions, Usability, Benchmarks, Evaluation, and Standardisation for Runtime Verification Tools (RV-CUBES). vol. 3, pp. 138–156. Kalpa Publications, Seattle, WA, USA (September 2017)
51. Schumann, J., Moosbrugger, P., Rozier, K.Y.: R2U2: Monitoring and Diagnosis of Security Threats for Unmanned Aerial Systems. In: Proceedings of the 15th International Conference on Runtime Verification (RV15). Springer-Verlag, Vienna, Austria (September 2015)
52. Schumann, J., Moosbrugger, P., Rozier, K.Y.: Runtime Analysis with R2U2: A Tool Exhibition Report. In: RV. Springer-Verlag, Madrid, Spain (September 2016)
53. Schumann, J., Rozier, K.Y., Reinbacher, T., Mengshoel, O.J., Mbaya, T., Ippolito, C.: Towards real-time, on-board, hardware-supported sensor and software health management for unmanned aerial systems. In: PHM. pp. 381–401 (October 2013)
54. Schumann, J., Rozier, K.Y., Reinbacher, T., Mengshoel, O.J., Mbaya, T., Ippolito, C.: Towards real-time, on-board, hardware-supported sensor and software health management for unmanned aerial systems. IJPHM **6**(1), 1–27 (June 2015)
55. Selyunin, K., Nguyen, T., Bartocci, E., Nickovic, D., Grosu, R.: Monitoring of MTL specifications with IBM's spiking-neuron model. In: DATE. pp. 924–929 (March 2016)

56. Selyunin, K., Nguyen, T., Bartocci, E., Grosu, R.: Applying Runtime Monitoring for Automotive Electronic Development, pp. 462–469. Springer International Publishing, Cham (2016)
57. Tiger, M., Heintz, F.: Stream reasoning using temporal logic and predictive probabilistic state models. In: TIME. pp. 196–205. IEEE (2016)
58. Todman, T., Stilkerich, S., Luk, W.: In-circuit temporal monitors for runtime verification of reconfigurable designs. In: DAC. pp. 50:1–50:6. ACM, New York, NY, USA (2015)
59. Torabi, A.J., Er, M.J., Li, X., Lim, B.S., Zhai, L., Oentaryo, R.J., Peen, G.O., Zurada, J.M.: A survey on artificial intelligence-based modeling techniques for high speed milling processes. IEEE Systems Journal **9**(3), 1069–1080 (2015)
60. Yoon, H., Chou, Y., Chen, X., Frew, E., Sankaranarayanan, S.: Predictive runtime monitoring for linear stochastic systems and applications to geofence enforcement for uavs. In: RV. pp. 349–367. Springer (2019)
61. Yu, X., Dong, W., Yin, X., Li, S.: Model predictive monitoring of dynamic systems for signal temporal logic specifications. arXiv preprint arXiv:2209.12493 (2022)
62. Zhang, P., Zambreno, J., Jones, P.H.: An embedded scalable linear model predictive hardware-based controller using admm. In: ASAP. pp. 176–183. IEEE (2017)
63. Zhang, X., Leucker, M., Dong, W.: Runtime verification with predictive semantics. In: NFM. pp. 418–432. Springer (2012)