Runtime Monitoring of Time Window Temporal Logic

Ernest Bonnah and Khaza Anuarul Hoque

***Abstract*—Temporal logic is becoming increasingly popular for its application in the analysis and control of dynamic systems. Time window temporal logic (TWTL) is a rich expressive language for specifying time-bounded serial tasks in a compact manner that is common in many control applications, such as robotics. Typically, TWTL specifications are verified using an automata-based model checking algorithm. However, verification of important properties of a given system using model checking in design time does not guarantee that the system will behave as expected during the runtime operation which falls under the scope of runtime verification. In this paper, we present a rewriting-based algorithm for runtime monitoring of safety requirements expressed in TWTL. The feasibility and efficiency of our proposed approach is demonstrated using two case studies related to unmanned aerial vehicle (UAV) surveillance and industrial robotics for manufacturing.**

***Index Terms*—Formal Methods in Robotics and Automation, Task and Motion Planning, Time Window Temporal Logic, Runtime verification**

I. Introduction

LINEAR temporal logic (LTL) [1] is a widely accepted language for specifying properties of reactive systems and their behavior over time. Specifically, LTL is popular in verifying safety-critical hardware, software, and communication systems [2]. In the past few years, LTL has been extensively used for analysis and control of dynamical systems, such as robotic mission specification and planning [3], [4], [5], [6]. Even though LTL offers a rich set of operators, it has a few limitations as well when it comes to the robotic mission specification. For instance, LTL cannot express tasks with time constraints such as "*visit P for 2 time units within 10 time units, and after this visit Q for 1 time unit within 12 time units, and visiting Z must be performed after visiting both P and Q twice*". Such specifications with time constraints can be expressed using bounded linear temporal logic (BLTL) [7], signal temporal logic (STL) [8], metric temporal logic (MTL) [9] and time window temporal logic (TWTL) [10]. The TWTL language has rich semantics for specifying different time-bounded specifications and has several advantages. To be specific, time-bounded specifications can be represented more compactly and comprehensively in TWTL when compared to BLTL, MTL, and STL. This is since the deadlines in TWTL represent the exact time bounds in contrast to an STL formula where the time bounds need to be shifted. For instance, let us consider a specification as "*stay at P for 3 time steps within the time window [0, 20]*". This can be expressed in TWTL as $[\mathbf{H}^3 P]^{0,20}$ where the outermost time window needs to be modified with respect to the inner time window. The same specification can be expressed in STL as $\mathbf{F}_{[0,20-3]}\mathbf{G}_{[0,3]}P$ where the outermost time window needs to be modified with respect to the inner time window. Moreover, another advantage of TWTL over the STL, BLTL and, MTL is the explicit concatenation operation which is very useful in specifying the serial tasks that are very common in robotics for real-world applications including robotic assembly, UAV surveillance/mission planning, manufacturing plant inspection and data collection, and autonomous package delivery. For instance, the IKEA furniture assembly environment for accelerating the automation of physical assembly processes [11] is a real-world example for which a strict (timed) sequence needs to be followed. Thus, TWTL is becoming popular in robotic mission planning applications [12], [13]. It is worth mentioning that another distinguishing feature of TWTL (when compared to the other logics) is the notion of *temporal relaxation* (the relaxation of task deadlines), which is a quantity computed over the time intervals of a given TWTL formula.

Traditionally, TWTL specifications have been verified using an automata-based model checking framework [10]. Model checking is indeed an unavoidable part of designing safety-critical systems. However, verification of important properties of a given system using model checking does not guarantee that the system will behave as expected during the runtime operation which falls under the scope of runtime verification (RV) [14]. Runtime verification (RV) is a lightweight verification technique that checks whether a run of a system satisfies or violates a given correctness property, which is also a requirement for robotic or autonomous systems that are regulated by international standards (ISO 26262, IEC 61508) [15]. Furthermore, unlike model checking, RV does not need the model of the system for monitoring the specification, it is well-suited for black-box systems. Due to the increasing popularity of runtime verification, researchers are actively working on developing and improvising the performance of runtime monitoring algorithms for different formal specification languages. For instance, the authors in [16] presented runtime monitoring algorithms for LTL and timed LTL (TLTL). Similarly, the runtime monitoring algorithms for STL and MTL are presented in [17], [18] and [19], respectively. A comprehensive survey of state-of-the-art runtime monitoring

Manuscript received: October, 19, 2021; Revised January, 21, 2022; Accepted March, 4, 2022.

This paper was recommended for publication by Editor Lucia Pallottino upon evaluation of the Associate Editor and Reviewers' comments. *(Corresponding author: Khaza Anuarul Hoque.)*

The authors are with the Department of Electrical & Computer Engineering, University of Missouri, Columbia, MO 65211 USA (e-mail: eeb7z3@umsystem.edu; hoquek@umsystem.edu)

Digital Object Identifier (DOI): see top of this page.

techniques and tools can be found in [15], [20].

In this paper, we propose novel and efficient algorithms for offline runtime monitoring[1] of TWTL specifications. Specifically, our approach leverages the idea of constructing basic rewriting rules (the subordinate terms of a given formula are replaced based on a given set of rules) [21] which can iteratively be used to evaluate the behaviour of a given system. Though rewriting-based techniques are often considered as an alternative technique when compared to the traditional runtime monitoring techniques, rewriting logic offers several advantages. For instance, rewriting-based algorithms can effectively and efficiently evaluate formal specifications on finite execution traces online (and offline) by processing each event as it arrives. Rewriting-based algorithms are indeed aggressive iterative algorithm that performs formula reduction based on formula rewriting to save any necessary trace state. The formulas are eventually reduced to truth values which show whether a given formula is satisfied or violated the given trace. To demonstrate the effectiveness of our approach, we formalize the requirements for unmanned aerial vehicles (UAVs) in a time-critical surveillance mission as TWTL specifications and monitor their satisfaction/violation in a simulated environment using our proposed algorithms. Additionally, we evaluate the performance of our proposed algorithms using an industrial robotics case study for manufacturing system [22]. The obtained results show that the algorithm has linear space and time complexity with respect to the number of the traces. To the best of our knowledge, this is the first work in the runtime monitoring of TWTL.

The rest of the paper is organized as follows: Section II presents the syntax and semantics of TWTL. The proposed rewriting algorithms for TWTL monitoring is introduced in Section III. In sections IV and V, we present two case studies to evaluate the effectiveness of the proposed algorithms. Finally, section VI concludes the paper.

II. Preliminaries

Let AP be a finite set of *atomic propositions* and $\Sigma = 2^{AP}$ be the powerset of AP. Let $\mathbb{Z}_{\geq 0} \times \Sigma$ be the *alphabet*, where $\mathbb{Z}_{\geq 0}$ is the set of non-negative integers. A *time-stamped event* is a member of our alphabet and is of the form (τ, e), where $e \in \Sigma$ and $\tau \in \mathbb{Z}_{\geq 0}$. A *timed trace* (or simply *trace*), t, is a sequence of time-stamped events of the form:

$$t = (\tau_0, e_0), (\tau_1, e_1), (\tau_2, e_2), \ldots$$

We abbreviate trace t by $(\tau_i, e_i)_{i \in \mathbb{Z}_{\geq 0}}$ and we call the sequence $\tau_0 \tau_1 \tau_2 \cdots$ of *time-stamps* and their indices i *time-points* and require that (1) $\tau_0 = 0$, and (2) $\tau_i \leq \tau_{i+1}$, for every $i \geq 0$; i.e., time-stamps increase monotonically. For a trace t, we represent the n^{th} time-stamped event by $t[n]$. let $t[i,j]$ denote the subtrace of t from position i up to and including position j. For a trace $t = (\tau_i, e_i)_{i \in \mathbb{Z}_{\geq 0}}$, by $t[i].e$, we mean e_i and by $t[i].\tau$, we mean τ_i.

[1]This paper proposes the offline version of the algorithm that is suitable for logged traces. However, the proposed algorithms can be easily extended to support online monitoring with minor modifications where it consumes events from the application.

A. Time Window Temporal Logic

Time window temporal logic (TWTL) was first introduced in [10]. Besides robotics, TWTL can be used in various domains (e.g., manufacturing, control, software development) that involve specifications with explicit time bounds. In particular, TWTL formulas can express tasks, their durations, and their time windows. TWTL is a linear-time logic encoding sets of discrete-time sequences with values in a finite alphabet.

TWTL syntax: The set of TWTL formulas is inductively defined by the following grammar:

$$\phi := \top \mid \mathbf{H}^{\mathbb{D}} a \mid \mathbf{H}^{\mathbb{D}} \neg a \mid \phi_1 \wedge \phi_2 \mid \neg \phi \mid \phi_1 \odot \phi_2 \mid [\phi]^{\mathbb{T}}$$

where $\top$ stands for true, a is an atomic proposition in AP. The operators $\mathbf{H}^{\mathbb{D}}$, $\odot$ and $[\ ]^{\mathbb{T}}$ represent the hold operator with $\mathbb{D} \in \mathbb{Z}_{\geq 0}$, concatenation operator and within operator respectively with $\mathbb{T}$ as a discrete-time constant interval $[\tau, \tau']$, where $\tau, \tau' \in \mathbb{Z}_{\geq 0}$ and $\tau' \geq \tau$, respectively and $\wedge$ and $\neg$ are the conjunction and negation Boolean operators respectively. The disjunction operator ($\vee$) can be derived from the negation and conjunction operators. Likewise, the implication operator ($\rightarrow$) can also be derived from the negation and disjunction operators.

TWTL semantics: A list of events can be considered as sequence of states of our proposed algorithm with each probable state representing the propositions that satisfy the state. The satisfaction relation defined by $\models$ defines when subtrace $t[i,j]$ of a (possibly infinite) timed-trace t, satisfies the TWTL formula. This is denoted by $t[i,j] \models \phi$. Given a timed-trace, $t[i,j]$ where the trace starts at $\tau_i \geq 0$ and terminates at $\tau_j \geq \tau_i$, the semantics of the operators is defined as:

$$\begin{array}{lll}
t[i,j] \models \top & & \\
t[i,j] \models \mathbf{H}^{\mathbb{D}} a & \text{iff} & a \in t[n].e,\ \forall n \in \{i, ..., i+\mathbb{D}\} \wedge \\
& & (t[j].\tau - t[i].\tau) \geq \mathbb{D} \\
t[i,j] \models \mathbf{H}^{\mathbb{D}} \neg a & \text{iff} & a \notin t[n].e,\ \forall n \in \{i, ..., i+\mathbb{D}\} \wedge \\
& & (t[j].\tau - t[i].\tau) \geq \mathbb{D} \\
t[i,j] \models \phi_1 \wedge \phi_2 & \text{iff} & (t[i,j] \models \phi_1) \wedge (t[i,j] \models \phi_2) \\
t[i,j] \models \neg \phi & \text{iff} & \neg (t[i,j] \models \phi) \\
t[i,j] \models \phi_1 \odot \phi_2 & \text{iff} & \exists k = \arg\min_{i \leq k < j} \{t[i,k] \models \phi_1\} \wedge \\
& & (t[k+1,j] \models \phi_2) \\
t[i,j] \models [\phi]^{[\tau,\tau']} & \text{iff} & \exists k \geq i + \tau.\, t[k, i+\tau'] \models \phi \ \wedge \\
& & (t[j].\tau - t[i].\tau) \geq \tau'
\end{array}$$

With $\phi = \mathbf{H}^{\mathbb{D}} a$ and $\mathbf{H}^{\mathbb{D}} \neg a$, a is expected to be repeated or not repeated for $\mathbb{D}$ time units with the condition that $a \in AP$ and $a \notin AP$ respectively. With $\phi = \phi_1 \wedge \phi_2$, the trace $t[i,j]$ satisfies both formula. The trace $t[i,j]$ is expected to satisfy at least one of the formulas in $\phi = \phi_1 \vee \phi_2$ while in $\neg \phi$, the trace, $t[i,j]$ does not satisfy the given formula. A given formula in the form $\phi_1 \odot \phi_2$ specifies that a given trace should satisfy the first formula first and the second afterwards with one time unit difference between the end of execution of ϕ_1 and start of execution of ϕ_2. The trace, $t[i,j]$ must satisfy ϕ between the time window $[\tau, \tau']$ given $[\phi]^{[\tau,\tau']}$.

Algorithm 1: `Monitor`

```
Inputs  : TWTL formula $\varphi$, Trace $t$
Outputs: Verdict = $\bot, \top$
1: $\beta \leftarrow$ Reduce($\varphi$)
2: if $\beta \in (\top, \bot)$ then
3:     break
4: else
5:     while get_event $(e_i \in t)$ do
6:         $\psi \leftarrow$ Progress($\varphi, e_i$)
7:         $\beta \leftarrow$ Reduce($\psi$)
8:         if $\beta \in (\top, \bot)$ then
9:             break
10:        else
11:            $\varphi \leftarrow \beta$
12:        end if
13:    end while
14: end if
15: return $(\beta)$
```

III. Rewriting-based Monitoring Algorithms for TWTL

A. Algorithm Overview

Given a TWTL formula φ, our proposed rewriting algorithm consists of the following three parts:

- `Monitor`: The main monitor iterates over the new events by recursively rewriting the given formula using the `Progress` and `Reduce` functions. The `Monitor` terminates if there are no new events or the re-written formula evaluates to *true* or *false* indicating the satisfaction or violation of the given formula, respectively.
- `Progress`: For each event e_i, the progress function rewrites the given formula φ to a new formula ψ based on the TWTL semantics (Section II).
- `Reduce`: The rewritten formula ψ is reduced to an equivalent formula β using the propositions of various TWTL formulas.

B. Algorithm Details

1) Monitoring Algorithm: The monitoring algorithm (Algorithm 1) has a while loop which continues to iterate over the trace until a verdict is returned. The algorithm first checks if the formula can be reduced or not (Line 1). The algorithm reads a timed-stamped event from the trace (Line 5, starting from the first event e_0 in the trace), then calls the `Progress` function which rewrites φ based on the event e_i (where i is the time point as explained in Section II) and TWTL semantics (Line 6). We then simplify the rewritten formula ψ using the `Reduce` function (Line 7) and check for the satisfaction or violation (Line 8). Finally, we terminate the monitoring and return the verdict in case of satisfaction/violation ($\top/\bot$) or we continue monitoring for the incoming events.

2) Progress Algorithm: The `Progress` is a recursive function[2] (Algorithm 2) that takes as input a TWTL formula along with an event and returns the corresponding rewritten formula based on the TWTL semantics (Section II). The first two cases (Lines 3 and 4) are trivial (i.e., false and true) and `Progress` function returns false and true, respectively. The rewriting for the hold operator $\mathbf{H}^{\mathbb{D}}p$ (Line 5) is based on the fact that $\mathbf{H}^0 p = \mathbf{H}^1 p = p$ (see [10] for detail). Indeed, for the cases in Lines 6 and 7, we check the satisfaction of the proposition p in the event e_i and return true and false accordingly. For the cases where $\mathbb{D} > 1$, we ensure that p is satisfied in the current event e_i and return $\mathbf{H}^{\mathbb{D}-1}p$ to reflect the remaining requirement for p to be satisfied in the upcoming events. The rewriting for the hold operator with the negation of a given proposition (i.e., $\mathbf{H}^{\mathbb{D}}\neg p$) is formalized in a similar way as the previous case (Lines 9–12). For the conjunction and disjunction operators (Line 13), we recursively apply the `Progress` function on the operands (i.e., φ_1 and φ_2). For the negation operator (Line 14), we push the `Progress` function inside the negation operator. We consider two cases (Lines 15–17) for rewriting the concatenation operator ($\varphi_1 \odot \varphi_2$): 1) if φ_1 is true then we recursively apply the `Progress` function on the formula φ_2 and return `Progress` (φ_2); 2) for all other cases, we recursively apply the `Progress` function on the formula φ_1 and return `Progress` $(\varphi_1, e_i) \odot \varphi_2$. We consider two cases for the within operator (Line 18): 1) if the formula φ corresponds to the hold operator (Line 19) then we ensure that the holding condition $\mathbb{D}$ can be satisfied within the interval $[\tau, \tau']$. We return a disjunctive formula which essentially ensures that either the formula φ satisfies at the current event e_i or it will satisfy within the remaining time interval $[\tau+1, \tau']$. 2) for all other cases (Line 21), we return the same disjunctive formula as the previous case except we don't impose any holding condition. The last case (Line 22) represents the default condition for which we return the given

[2]Note that Algorithm 2 and Algorithm 3 are described in the functional programming/Ocaml style.

Algorithm 2: `Progress`

```
1: Progress $(\varphi, e_i)$ :=
2: match $\varphi$ with
3: | $\top \to \top$
4: | $\bot \to \bot$
5: | $\mathbf{H}^{\mathbb{D}}p \to$ match $\mathbb{D}$ with
6:                | $0 \to$ if $p \in e_i$ then $\top$ else $\bot$
7:                | $1 \to$ if $p \in e_i$ then $\top$ else $\bot$
8:                | _ $\to$ if $p \notin e_i$ then $\bot$ else $\mathbf{H}^{\mathbb{D}-1}p$
9: | $\mathbf{H}^{\mathbb{D}}\neg p \to$ match $\mathbb{D}$ with
10:               | $0 \to$ if $p \notin e_i$ then $\top$ else $\bot$
11:               | $1 \to$ if $p \notin e_i$ then $\top$ else $\bot$
12:               | _ $\to$ if $p \in e_i$ then $\bot$ else $\mathbf{H}^{\mathbb{D}-1}\neg p$
13: | $\varphi_1 \wedge \varphi_2 \to$Progress $(\phi_1, e_i) \wedge$ Progress $(\varphi_2, e_i)$
14: | $\neg\varphi \to \neg$ Progress $(\varphi, e_i)$
15: | $\varphi_1 \odot \varphi_2 \to$ match Progress $(\varphi_1, e_i)$ with
16:               | $\top \to$ Progress $(\varphi_2, e_i)$
17:               | _ $\to$ (Progress $(\varphi_1, e_i) \odot \varphi_2$)
18: | $[\varphi]^{[\tau,\tau']} \to$ match $\varphi$ with
19:               | $\mathbf{H}^{\mathbb{D}}\,p \to$ if $\mathbb{D} \le (\tau' - \tau)$ then
20:                  (Progress $(\varphi, e_i) \vee [\varphi]^{[\tau+1,\tau']}$) else $\bot$
21:               | _ $\to$ (Progress $(\varphi, e_i) \vee [\varphi]^{[\tau+1,\tau']}$)
22: | _ $\to \varphi$
```

Algorithm 3: `Reduce`

```
1: Reduce $(\psi)$ :=
2: match $\psi$ with
3: | $\neg f \rightarrow$ match (Reduce $f$) with
4:               | $\bot \rightarrow \top$
5:               | $\top \rightarrow \bot$
6:               | $\neg g \rightarrow g$
7: | $\varphi_1 \wedge \varphi_2 \rightarrow$ match (Reduce $\varphi_1$), (Reduce $\varphi_2$) with
8:                 | $(\bot, f) \rightarrow \bot$
9:                 | $(f, \bot) \rightarrow \bot$
10:                | $(\top, f) \rightarrow f$
11:                | $(f, \top) \rightarrow f$
12:                | $(\top, \top) \rightarrow \top$
13: | $\varphi_1 \odot \varphi_2 \rightarrow$ match (Reduce $\varphi_1$), (Reduce $\varphi_2$) with
14:                | $\bot, f \rightarrow \bot$
15:                | $f, \bot \rightarrow \bot$
16:                | $\top, f \rightarrow f$
17:                | $f, g \rightarrow f \odot g$
18: | $[\phi]^{[\tau,\tau']} \rightarrow$ if $\tau = \tau'$ then $\phi$ else $[\phi]^{[\tau,\tau']}$
19: | _ $\rightarrow \psi$
```

formula φ.

3) Reduce Algorithm: The `Reduce` is a recursive function (Algorithm 3) that accepts as input a TWTL formula and returns a reduced TWTL formula. The first two cases (i.e., $\neg$, $\wedge$) reduce the formula based on the Boolean simplification rules (Lines 3–12) where f and g denote TWTL formulas. The reduction of the concatenation operator ($\varphi_1 \odot \varphi_2$) requires following sub-cases: a) if one of the formula is false ($\bot$) then the result becomes false (Lines 13–17); b) if φ_1 is true ($\top$) then we return the reduced form of the φ_2, i.e., `Reduce` (φ_2) (Line 16); and c) in all other case, we rerun `Reduce` (φ_1) $\odot$ `Reduce` (φ_2) (Line 17). We reduce the within operator $[\phi]^{[\tau,\tau']}$ to ϕ in case of $\tau = \tau'$, i.e., unity interval. For all other case we return the formula without any reduction (Line 18). If any of the patterns between Lines 3–18 does not match, then Line 19 returns the formula (which means the formula cannot be further reduced, or the formula is not reducible at all).

Correctness. We prove the correctness of the proposed monitoring algorithm using the following theorem. We prove this theorem based on the structural induction on the formula.

Theorem 1: Let ϕ be a TWTL formula. For a given trace t, Algorithm 1 returns $\bot/\top$ *iff* $[t[i,j] \models \phi] = \bot/\top$.

Proof: The base cases are associated with $\phi \in (\top, \bot, \mathbf{H}^{\mathbb{D}}p, \mathbf{H}^{\mathbb{D}}\neg p)$. The proof for base cases $(\top, \bot)$ is trivial since Algorithm 2 mimics the semantics of TWTL for $\top$ and $\bot$. The TWTL semantics of the hold operators follows the case analysis as presented in Algorithm 2.

We proceed with the induction on the structure of the formula $\phi \in (\varphi_1 \wedge \varphi_2,\ \varphi_1 \odot \varphi_2,\ \neg\varphi,\ [\varphi]^{[\tau,\tau']})$. For the case of conjunction operator $\phi = \varphi_1 \wedge \varphi_2$, the induction hypothesis requires that $\forall t. t[i,j] \models \varphi_1 \wedge \varphi_2 \equiv$ `Monitor` $(\varphi_1 \wedge \varphi_2, t[i,j])$. We distinguish two sub-cases: $\varphi_1 = \bot$ and $\varphi_2 = \bot$. For the first sub-case where $\varphi_1 = \bot$, we have $\phi = \bot \wedge \varphi_2$. From the TWTL semantics on the $\wedge$ operator, $t[i,j] \models (\bot \wedge \varphi_2) = \bot$. Similarly, according to Algorithm 1, `Monitor` $(\bot \wedge \varphi_2, t[i,j]) = \bot$ after applying Algorithm 3 (`Reduce`) on ϕ. For the second sub-case $\varphi_2 = \bot$, we have $\phi = \varphi_1 \wedge \bot$. Based on TWTL semantics, $t[i,j] \models (\varphi_1 \wedge \bot) = \bot$. Consequently, based on Algorithm 1, `Monitor` $(\varphi_1 \wedge \bot, t[i,j]) = \bot$ after applying Algorithm 3 (`Reduce`) on ϕ. Similarly, the other sub-cases for the conjunction operator ($\wedge$) where $\varphi_1 = \top$ and $\varphi_2 = \top$, $\varphi_1 = \bot$ and $\varphi_2 = \top$, and $\varphi_1 = \top$ and $\varphi_2 = \bot$ can be proved by induction. The induction cases for the other operators can also be proved in a similar fashion.

Complexity. The worst case time and space complexity for Algorithm 1 is $O(|t|)$, where $|t|$ is length of a trace t. This is due to the fact that the execution of "while loop" in the Algorithm 1 is dependent on number of events in each trace, i.e. the length of a trace $|t|$. The worst-case time complexity of Algorithm 1 is therefore bounded by $O(|t|)$. For N number of traces, each with a fixed length of $|t|$, the time complexity of Algorithm 1 is $O(N * |t|)$.

The space required for executing Algorithm 1 also depends on the length of a trace $|t|$. Thus, similar to the time complexity, we can deduce that the space complexity is also $O(|t|)$ for a trace, and for N number of traces, each with a fixed length of $|t|$, the space complexity of Algorithm 1 is $O(N * |t|)$.

C. Example

Consider an autonomous robot which is capable of performing n Tasks (T_1, T_2, $\cdots$, T_n). One important requirement for this robot is to perform task T_1 for two time units within the interval $[0,4]$. We can transform this requirement into a TWTL formula as $\varphi = [\mathbf{H}^2\ T_1]^{[0,4]}$. In the following, we demonstrate the application of Algorithms 1, 2 and 3, given the following logged trace of the robot:

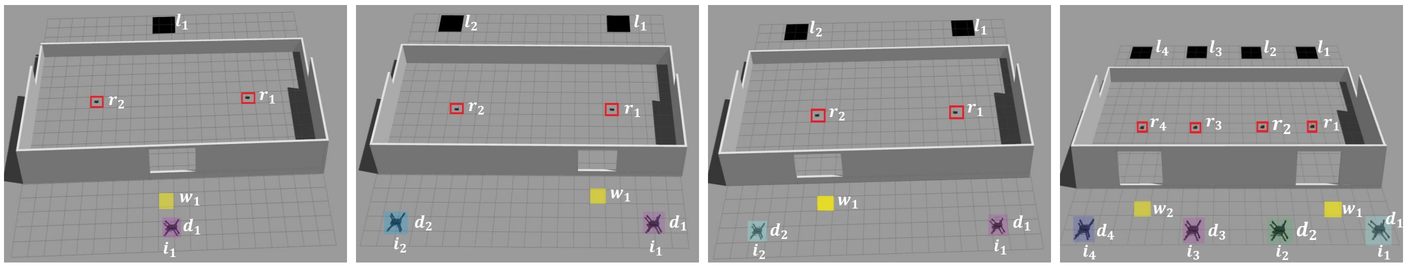
$$trace = (0, \{T_2\}), (1, \{T_1\}), (2, \{T_1\}), (3, \{T_2\}), (4, \{T_2\})$$

Step 1: $e_i = (0, \{T_2\})$

$\psi \leftarrow$Progress$([\mathbf{H}^2\ T_1]^{[0,4]}, (0, \{T_2\}))$
$\psi \leftarrow$Progress$([\mathbf{H}^2\ T_1]^{[0,4]}, (0, \{T_2\}) \vee ([\mathbf{H}^2\ T_1]^{[1,4]}))$
$\psi \leftarrow \bot \vee ([\mathbf{H}^2\ T_1]^{[1,4]})$
$\beta \leftarrow$Reduce(ψ)
$\beta \leftarrow$Reduce$(\bot \vee ([\mathbf{H}^2\ T_1]^{[1,4]}))$
$\beta \leftarrow [\mathbf{H}^2\ T_1]^{[1,4]}$

Step 2: $e_i = (1, \{T_1\})$

$\psi \leftarrow$ `Progress` $([\mathbf{H}^2\ T_1]^{[1,4]}, (1, \{T_1\}))$
$\psi \leftarrow$`Progress` $([\mathbf{H}^2\ T_1]^{[1,4]}, (1, \{T_1\}) \vee ([\mathbf{H}^2\ T_1]^{[2,4]}))$
$\psi \leftarrow \mathbf{H}^1\ T_1 \vee ([\mathbf{H}^2\ T_1]^{[1,4]})$
$\beta \leftarrow$`Reduce` (ψ)
$\beta \leftarrow$`Reduce` $(\mathbf{H}^1\ T_1 \vee ([\mathbf{H}^2\ T_1]^{[1,4]}))$
$\beta \leftarrow \mathbf{H}^1\ T_1 \vee ([\mathbf{H}^2\ T_1]^{[2,4]})$



(a) Scenario 1 (b) Scenario 2 (c) Scenario 3 (d) Scenario 4

Fig. 1: Gazebo environment for a surveillance mission with designated waypoints for scenarios 1-4 in case study I

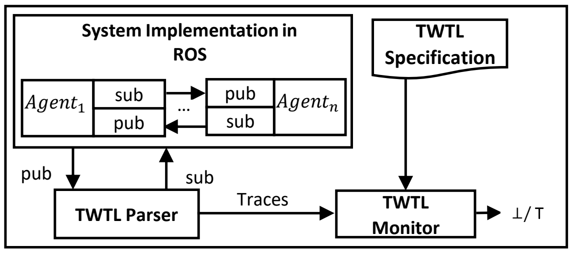


Fig. 2: ROS and TWTL monitor integration architecture

Step 3: $e_i = (2, \{T_1\})$

```
$\psi \leftarrow$Progress $((\mathbf{H}^1\ T_1 \vee ([\mathbf{H}^2\ T_1]^{[1,4]})), (2, \{T_1\}))$
$\psi \leftarrow$Progress $((\mathbf{H}^1\ T_1), (2, \{T_1\})) \vee$
    Progress $([\mathbf{H}^2\ T_1]^{[1,4]}), (2, \{T_1\}))$
$\psi \leftarrow \top \vee$Progress $(([\mathbf{H}^2\ T_1]^{[1,4]}), (2, \{T_1\}))$
$\beta \leftarrow$Reduce $(\psi)$
$\beta \leftarrow$Reduce $(\top \vee$Progress $([\mathbf{H}^2\ T_1]^{[1,4]}), (2, \{T_1\}))$
$\beta \leftarrow \top$ (requirement satisfied)
```

IV. Case Study I

To demonstrate the feasibility of the proposed runtime monitoring algorithms, the first case study we perform resembles a time-bounded surveillance mission in a combat environment. We consider 4 different scenarios by varying the number of UAVs with one or multiple windows as shown in Figure 1. In each scenario, the mission is to navigate through the designated trajectories to the point of interest (an object in this case) and gather information about the object through videos or images relayed to the control room before proceeding to the landing position after the completion of the surveillance task. Such surveillance missions are common in combat situations where UAVs are deployed to observe or collect information about the objects of interest, such as enemy positions, explosives, etc.

The overall experimental setup is shown in Figure 2 and has three main components: system implementation in ROS, a parser for parsing traces that are obtained from the *system implementation in ROS* block, and TWTL monitor based on the proposed runtime monitoring algorithms, described as follows.

System Implementation in ROS: The case study is simulated using the Gazebo open-source simulator and the Robot Operating System (ROS) [23]. The UAVs deployed in the ROS environment are equipped with different ROS packages such as navigation, obstacle avoidance, and localization. The navigation package helps an autonomous robot (in this case UAV) to navigate from its initial position to the goal position while avoiding obstacles utilizing the sensors attached to the UAVs. Localization is one of the fundamental requirements of an autonomous robot since the knowledge of the location of a robot at any point in time with respect to its environment is crucial for determining the next action which can be modeled using the localization package in a ROS environment.

We implement four different scenarios with a different number of UAVs and trajectories as shown in Figure 1. Scenario 1 consists of one UAV in action, with one window in the middle of the wall, two regions of interest (ROIs) with objects to be surveilled, and one landing site. Scenario 2 and scenario 3 both have two UAVs with different window positions with two regions of interest (ROIs) with objects to be surveilled and two landing sites. Finally, in scenario 4, we consider two windows, with four UAVs, four regions of interest (ROIs) with objects to be surveilled, and four landing sites.

In each scenario, UAVs are denoted as d_j, their initial position and landing positions are denoted as i_j and l_j, respectively, where $j \in \{1, ..., J\}$ and $J_{>0}$ represents the total number of UAVs. The regions of interests (ROIs) are denoted as r_m, where $m \in \{1, ..M\}$ and $M_{>0}$ represents the total number of ROIs. Finally, w_n represents the number of waiting areas in a scenario, where $n \in \{1, ..., N\}$ and $N_{>0}$.

Trace Parsing: Agents (UAVs in this case) in ROS perform their assigned tasks based on topics that they either publish or subscribe to. Topics are named channels or buses through which agents exchange messages of a particular type [24]. The goal of the parsing component is to translate the messages from the ROS environment into a textual form suitable as traces for our TWTL monitor. The trace parser does this by subscribing (sub) to the messages such as `nav_msgs/Odometry` published (pub) by the surveillance environment implemented in ROS as shown in Figure 1.

Monitoring: The proposed monitoring algorithms are implemented using OCaml [25] that accepts the finite traces returned by the trace parser and checks them against given TWTL specifications. The monitoring block returns $\top$ when a given trace t is satisfied the TWTL specification ϕ, and $\perp$ when the specification is violated.

TABLE I: Mission requirements for UAVs in case study I

No.	Mission requirements	TWTL specification
1	An UAV is armed (or started) from its initial position, and then takes off to a required altitude between time 0 to T_1.	$\theta_j^1 = [\mathbf{H}^1\ Arm] \odot [\mathbf{H}^1\ Take\_off]^{[0,T_1]}$
2	Proceed to the waiting area between time T_2 and T_3 while avoiding any obstacle or any other UAVs (if any).	$\theta_j^2 = [\mathbf{H}^1\ Waiting\_area]^{[T_2,T_3]} \wedge [\mathbf{H}^1\ \neg Obstacle] \wedge [\mathbf{H}^1\ \neg UAV_{nearby}]$
3	If d_j does not sense any other drone upon reaching the designated waiting area, it can then proceed through the window to perform the perform the surveillance task. In other words, proceed to ROI r_1 between time T_6 and T_7 while avoiding any obstacle or other UAVs.	$\theta_j^3 = [\mathbf{H}^1\ RoI_1]^{[T_4,T_5]} \wedge [\mathbf{H}^1\ \neg Obstacle] \wedge [\mathbf{H}^1\ \neg UAV_{nearby}]$
4	When in ROI r_1, hover it 3 times to gather information (via image or video) between time T_6 and T_7.	$\theta_j^4 = [\mathbf{H}^1\ RoI_1] \wedge [\mathbf{H}^3\ Hover]^{[T_6,T_7]}$
5	Proceed to ROI r_2 between time T_8 and T_9 while avoiding any obstacle or other UAVs.	$\theta_j^5 = [\mathbf{H}^1\ RoI_2]^{[T_8,T_9]} \wedge [\mathbf{H}^1\ \neg Obstacle] \wedge [\mathbf{H}^1\ \neg UAV_{nearby}]$
6	When in ROI r_2, hover it 3 times to gather information (via image or video) between time T_{10} and T_{11}.	$\theta_j^6 = [\mathbf{H}^1\ RoI_2] \wedge [\mathbf{H}^3\ Hover]^{[T_{10},T_{11}]}$
7	Proceed to the designated landing area l_j between time T_{12} and T_{13} and then perform landing between time T_{14} and T_{15}	$\theta_j^7 = [\mathbf{H}^1\ Landing\_point]^{[T_{12},T_{13}]} \odot [\mathbf{H}^1\ Land]^{[T_{14},T_{15}]}$
8	If the mission is failed then go back to the initial position between time T_{16} and T_{17}.	$\theta_j^8 = \neg\theta_j \rightarrow [\mathbf{H}^1\ Initial\_position]^{[T_{16},T_{17}]}$

TWTL specification. The TWTL specification for a UAV d_j in a given scenario is decomposed into k sub-tasks denoted as θ_j^k, where $k \in \mathbb{Z}_{\geq 0}$, and expressed in Table I.

For scenario 4, we consider four UAVs, four ROIs, and two windows with two waiting areas. As mentioned earlier, unlike scenarios 1,2, and 3, scenario 4 has two waiting areas namely w_1 and w_2. In scenario 4, drones d_1 and d_2 is assigned to w_1, and d_3 and d_4 is assigned to w_2. Since all the UAVs fly simultaneously, the TWTL specification d_3 and d_4 is also similar to θ_j^3 - θ_j^6 with same time bounds, and can be expressed as:

$\theta_j^9 = [\mathbf{H}^1\ RoI_3]^{[T_6,T_7]} \wedge [\mathbf{H}^1\ \neg Obstacle] \wedge [\mathbf{H}^1\ \neg UAV_{nearby}]$

$\theta_j^{10} = [\mathbf{H}^1\ RoI_3] \wedge [\mathbf{H}^3\ Hover]^{[T_8,T_9]}$

$\theta_j^{11} = [\mathbf{H}^1\ RoI_4]^{[T_{10},T_{11}]} \wedge [\mathbf{H}^1\ \neg Obstacle] \wedge [\mathbf{H}^1\ \neg UAV_{nearby}]$

$\theta_j^{12} = [\mathbf{H}^1\ RoI_4] \wedge [\mathbf{H}^3\ Hover]^{[T_{12},T_{13}]}$

In the specification θ_j^8, $\theta_j = \theta_j^1 \odot \theta_j^2 \odot \theta_j^3 \odot \theta_j^4 \odot \theta_j^5 \odot \theta_j^6 \odot \theta_j^7$.

We monitor the TWTL properties for all four scenarios considering the following time bounds: $T_1 = 2s, T_2 = 3s, T_3 = 6s, T_4 = 7s, T_5 = 11s, T_6 = 12s, T_7 = 16s, T_8 = 17s, T_9 = 20s, T_{10} = 21s, T_{11} = 25s, T_{12} = 26s, T_{13} = 30s, T_{14} = 31s, T_{15} = 35s, T_{16} = 36s$, and $T_{17} = 45s$. The experiments are performed on an Ubuntu 20.04 system with 65 GB RAM and Intel Core(TM) i9-10900 CPU (3.70 GHz) with ROS Noetic and Gazebo 11. The UAV altitude limit for all UAVs in all four scenarios for taking-off and steer modes is set to 2 meter. The obtained results are shown in Table II. We observe that in Scenario 1, d_1 was able to perform its assigned mission without violating any of the requirements. In contrast, d_2 in Scenario 2 violates the specification θ_j^2 to θ_j^7. Similar violations of some specifications are observed for d_1 in scenario 3, as well as d_1, d_2 and d_4 in scenario 4. d_1 in scenarios 1 and 2, d_2 in scenario 3 and d_3 in scenario 4 are the only cases that meets all the TWTL specifications. The execution time for monitoring specifications $\theta_j^1 - \theta_j^4$ in all four scenarios was approximately 0.001 second, whereas the execution time for monitoring specifications $\theta_j^5 - \theta_j^7$ was approximately 0.002 second. The execution times for monitoring θ_j^8 and θ_j were approximately 0.008 second and 0.014 second, respectively in all four scenarios. The memory consumption in all four scenarios for monitoring $\theta_j^1, \theta_j^2, \theta_j^3, \theta_j^4, \theta_j^5, \theta_j^6, \theta_j^7, \theta_j^8, and\ \theta_j$ never exceeded 0.003 MB, 0.003 MB, 0.004 MB, 0.005 MB, 0.006 MB, 0.007 MB, 0.008 MB, 0.057 MB, and 0.065 MB, respectively.

V. Case Study II

To demonstrate the efficiency of the proposed algorithms, we perform another case study that resembles the human-robot collaboration in the production line of an automotive manufacturing factory [22]. The human-robot collaboration case study involves doing assembly of different parts in a specific order. The production of the desired product (Product A) is carried out in a production line through eleven sub-processes. Each of the sub-processes (for instance, assembling of subset B) can be divided into 3 steps: importing of objects, positioning of objects, and the sequencing of the processes. The assembling of subset B also involves sub-processing right from the workstation setup to the delivery of the finalized subset to the next process. Once the operator completes the mounting and setup of the workstation, the collaborative robot is inserted in proximity to the device to hold the components as they are assembled. All the assembling of the subset is done by the robot with the operator assisting in the positioning of tools such as screws, marker pens, etc. into the robot. The sub-processes of subset B are divided into five steps and presented in Tables III - VII. For each step, the number of times for a given action to be repeated is specified by the frequency (Freq.), with their respective start times (ST) and end times (ET). The entire duration for performing a task is represented by the total time (TT). The letter in brackets in the description column of each row represents the propositions used for the TWTL formalization. We formalize the sub-processes of subset B as presented in Tables III-VII into five TWTL formulas as follows:

$\varphi_1 = [\mathbf{H}^1 A]^{[0,2]} \odot [\mathbf{H}^1 B]^{[3,4]} \odot [\mathbf{H}^1 C]^{[5,7]} \odot [\mathbf{H}^1 D]^{[8,9]}$

$\varphi_2 = [\mathbf{H}^1 E]^{[10,20]} \odot [\mathbf{H}^1 F]^{[21,40]} \odot [\mathbf{H}^1 G]^{[41,42]} \odot [\mathbf{H}^9 H]^{[43,50]} \odot [\mathbf{H}^5 I]^{[51,55]} \odot [\mathbf{H}^1 J]^{[56,57]}$

$\varphi_3 = [\mathbf{H}^1 K]^{[58,59]} \odot [\mathbf{H}^1 L]^{[60,61]} \odot [\mathbf{H}^1 M]^{[62,63]} \odot [\mathbf{H}^1 N]^{[64,65]} \odot [\mathbf{H}^1 O]^{66,67]}$

$\varphi_4 = [\mathbf{H}^1 P]^{[68,76]} \odot [\mathbf{H}^1 Q]^{[77,96]} \odot [\mathbf{H}^1 R]^{[97,98]} \odot$

TABLE II: Evaluation of sub-tasks of UAVs in case study I

TWTL	Scenario 1	Scenario 2		Scenario 3		Scenario 4			
spec.	d_1	d_1	d_2	d_1	d_2	d_1	d_2	d_3	d_4
θ_j^1	$\top$	$\top$	$\top$	$\top$	$\top$	$\top$	$\top$	$\top$	$\top$
θ_j^2	$\top$	$\top$	$\bot$	$\bot$	$\top$	$\bot$	$\top$	$\top$	$\bot$
θ_j^3	$\top$	$\top$	$\bot$	$\bot$	$\top$	$\bot$	$\top$	$\top$	$\bot$
θ_j^4	$\top$	$\top$	$\bot$	$\bot$	$\top$	$\bot$	$\top$	$\top$	$\bot$
θ_j^5	$\top$	$\top$	$\bot$	$\bot$	$\top$	$\bot$	$\top$	$\top$	$\bot$
θ_j^6	$\top$	$\top$	$\bot$	$\bot$	$\top$	$\bot$	$\top$	$\top$	$\bot$
θ_j^7	$\top$	$\top$	$\bot$	$\bot$	$\top$	$\bot$	$\top$	$\top$	$\bot$
θ_j^8	$\bot$	$\bot$	$\top$	$\top$	$\bot$	$\top$	$\bot$	$\bot$	$\top$
θ_j	$\top$	$\top$	$\bot$	$\bot$	$\top$	$\bot$	$\top$	$\top$	$\bot$

TABLE III: Assembly of subset B (Step 1)

Description	Freq.	ST(s)	ET(s)	TT(s)
(A) Walk to the slide	1	0	2	2
(B) Take subset	1	3	4	1
(C) Take pen	1	5	7	2
(D) Insert piece in the device	1	8	9	1

$[\mathbf{H}^1 S]^{[99,118]} \odot [\mathbf{H}^1 T]^{[119,120]}$
$\varphi_5 = [\mathbf{H}^1 U]^{[121,123]} \odot [\mathbf{H}^1 V]^{[124,125]} \odot [\mathbf{H}^1 W]^{[126,127]} \odot [\mathbf{H}^1 X]^{[128,129]} \cdot [\mathbf{H}^1 Y]^{[130,131]}$

We perform two sets of experiments for exploring the relationship between the number of traces and performance, and also the relationship between the number of events (length) and performance. For the first set of experiments, each formula from ϕ_1 to ϕ_5 was evaluated against a different number of traces ranging from $1,000$ to $10,000$ while the number of events in each trace was kept constant at 200. Their respective execution time and memory consumption was recorded. For the second set of experiments, we evaluate formulas ϕ_6 to ϕ_8 consisting of the sequential ($\odot$), and ($\wedge$) and implication ($\rightarrow$) operators against varying trace length T_n (i.e., number of events) ranging from 1,000 to 10,000. For these two sets of experiments we generated random traces using a Python program. It is important to randomly generate traces for case study II as we use these traces for the performance evaluation of our proposed algorithms. Generating such a large number of traces using the ROS environment (as we did in the case study I) for case study II is not feasible. The 3 formulas used in these experiments are:
$\varphi_6 = [\mathbf{H}^1 A]^{[0,100]} \odot [\mathbf{H}^1 B]^{[101,400]} \odot [\mathbf{H}^1 C]^{[401,700]} \odot [\mathbf{H}^1 D]^{[701,T_n]}$

TABLE IV: Assembly of subset B (Step 2)

Description	Freq.	ST(s)	ET(s)	TT(s)
(E) Take screw and position the piece	1	10	20	10
(F) Machine time of the cobot	1	11	40	19
(G) Take pen	1	41	42	1
(H) Seal all clips (9 clips)	9	43	50	7
(I) Seal all clips (5 clips)	5	51	55	4
(J) Release pen	1	56	57	1

TABLE V: Assembly of subset B (Step 3)

Description	Freq.	ST(s)	ET(s)	TT(s)
(K) Take steel	1	58	59	1
(L) Position on subset	1	60	61	1
(M) Walk to the slide	1	62	63	1
(N) Take top base	1	64	65	1
(O) Attach top plastic part to assembly	1	66	67	1

TABLE VI: Assembly of subset B (Step 4)

Description	Freq.	ST(s)	ET(s)	TT(s)
(P) Take screw and position the piece	1	68	76	8
(Q) Machine time of the Cobot (8 screws)	1	77	96	19
(R) Take pen	1	97	98	1
(S) Seal all screws (18 screws)	1	99	118	19
(T) Seal all clips (2 clips)	1	119	120	1

TABLE VII: Assembly of subset B (Step 5)

Description	Freq.	ST(s)	ET(s)	TT(s)
(U) Remove part from device (8 screws)	1	121	123	2
(V) Take label	1	124	125	1
(W) Take part	1	126	127	1
(X) Walk to the rack	1	128	129	1
(Y) Glue part on the rack	1	130	131	1

$\varphi_7 = [\mathbf{H}^1 A]^{[0,200]} \wedge [\mathbf{H}^1 B]^{[150,400]} \wedge [\mathbf{H}^1 C]^{[300,600]} \wedge [\mathbf{H}^1 D]^{[601,T_n]}$
$\varphi_8 = [\mathbf{H}^1 A] \rightarrow [\mathbf{H}^1 B]^{[0,T_n]}$

All the experiments are performed in the same computing environment as for case study I. Using stacked bar graph, the obtained results for the execution time and memory consumption for the first set of experiments is shown in Figure 3 (a) and (b). The obtained results for the second set of experiments is shown in Figure 3 (c) and (d).

Number of traces vs. performance: We observe in Figures 3 (a) and (b) that for each TWTL formula, the execution time monotonically increases with increase in the number of traces. For monitoring of φ_1 for $1,000$ traces, our algorithm takes 0.109 seconds. The execution increases to 0.359 seconds and 0.703 during the monitoring of φ_1 for $3,000$ and $6,000$ traces respectively. For $10,000$ traces while monitoring the same formula, the execution time increases to 1.234 seconds. The memory consumption for TWTL specifications also increases with the number of traces. For instance, while monitoring φ_5 for $1,000$, $4,000$ and $7,000$ traces, our algorithm consumes 219.5514 MB, 878.2058 MB, 1536.923 MB respectively. However, while monitoring for $10,000$ traces for the same formula, it consumes 2195.398 MB. Once again, we observe a linear trend for both execution time and memory consumption.

Number of events vs. performance: As shown in Figures 3 (c) and (d), we observe that the execution time increase with increase in the number of events in the trace. The proposed algorithm takes 0.000513 seconds to monitor θ_1 against a trace with $1,000$ events. The time however increased to 0.002301 and 0.003866 seconds for the same formula and and a trace with $5,000$ and $10,000$ events respectively. It is observed that memory consumption also increases with increase in number of events on a trace. For monitoring θ_3 for a trace with $1,000$ and $5,000$ events, our algorithm consumes 0.176 MB and 0.9996 MB respectively. The memory consumed however increases to 1.982058 MB when the number of events in the trace is increased to $10,000$ while monitoring the same formula. It can be observed that both the execution time and memory consumption follow a linear trend.

The obtained results using the two case studies in sections 4 and 5 show the effectiveness and efficiency of our proposed

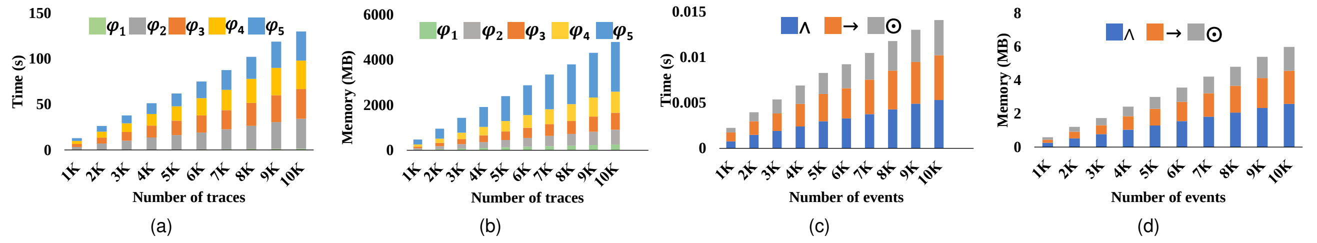


Fig. 3: (a) Number of traces vs. execution time for $\varphi_1 - \varphi_5$; (b) Number of traces vs. memory consumption for $\varphi_1 - \varphi_5$; (c) Number of events vs. execution time for $\odot$, $\wedge$, $\rightarrow$; (d) Number of events vs. memory consumption for $\odot$, $\wedge$, $\rightarrow$

TWTL runtime monitoring algorithm. Even though the illustration used logged traces, the same monitor can be also used for online monitoring with minor modifications in the algorithm. We believe that such monitoring can be used for UAVs or industrial robots by inspecting the logged traces and by adding monitors in the early controller prototypes for quickly evaluating the correctness of the underlying algorithms. Thus, engineers working on the design and development of the safety-critical robots can use the proposed approach with limited, or even no special knowledge of runtime verification and gain formally verified insights about the given robotic system.

VI. Conclusion and Future Work

In this paper, we proposed algorithms for runtime monitoring of time window temporal logic (TWTL) specifications. Based on rewriting rules, the algorithm iterates through events in a given trace and returns a verdict determining whether a TWTL specification is satisfied or violated. Using an UAV surveillance mission and an industrial robotics case study, we demonstrated that the proposed approach is effective and also efficient in terms of memory and execution time with an increasing number of traces. In the future, we plan to extend the TWTL specification language with new temporal operators and provide online runtime monitoring algorithms for them. Also, we plan to implement the online version of the proposed algorithms on robots and demonstrate their on-field effectiveness.

References

[1] A. Pnueli, "The temporal logic of programs," in *18th Annual Symposium on Foundations of Computer Science (sfcs 1977)*. IEEE, 1977, pp. 46–57.

[2] K. Y. Rozier, "Linear temporal logic symbolic model checking," *Computer Science Review*, vol. 5, no. 2, pp. 163–203, 2011.

[3] M. Cai, M. Hasanbeig, S. Xiao, A. Abate, and Z. Kan, "Modular deep reinforcement learning for continuous motion planning with temporal logic," *arXiv preprint arXiv:2102.12855*, 2021.

[4] Y. E. Sahin, P. Nilsson, and N. Ozay, "Multirobot coordination with counting temporal logics," *IEEE Transactions on Robotics*, vol. 36, no. 4, pp. 1189–1206, 2019.

[5] M. Kloetzer and C. Mahulea, "Path planning for robotic teams based on ltl specifications and petri net models," *Discrete Event Dynamic Systems*, vol. 30, no. 1, pp. 55–79, 2020.

[6] G. Xie, Z. Yin, and J. Li, "Temporal logic based motion planning with infeasible ltl specification," in *2020 Chinese Control And Decision Conference (CCDC)*. IEEE, 2020, pp. 4899–4904.

[7] I. Tkachev and A. Abate, "Formula-free finite abstractions for linear temporal verification of stochastic hybrid systems," in *Proceedings of the 16th international conference on Hybrid systems: computation and control*, 2013, pp. 283–292.

[8] O. Maler and D. Nickovic, "Monitoring temporal properties of continuous signals," in *Formal Techniques, Modelling and Analysis of Timed and Fault-Tolerant Systems*. Springer, 2004, pp. 152–166.

[9] S. Saha and A. A. Julius, "Task and motion planning for manipulator arms with metric temporal logic specifications," *IEEE robotics and automation letters*, vol. 3, no. 1, pp. 379–386, 2017.

[10] C. I. Vasile, D. Aksaray, and C. Belta, "Time window temporal logic," *Theoretical Computer ence*, 2016.

[11] Y. Lee, E. S. Hu, and J. J. Lim, "IKEA furniture assembly environment for long-horizon complex manipulation tasks," in *IEEE International Conference on Robotics and Automation (ICRA)*, 2021. [Online]. Available: https://clvrai.com/furniture

[12] A. S. Asarkaya, D. Aksaray, and Y. Yazıcıoğlu, "Temporal-logic-constrained hybrid reinforcement learning to perform optimal aerial monitoring with delivery drones," in *2021 International Conference on Unmanned Aircraft Systems (ICUAS)*. IEEE, 2021, pp. 285–294.

[13] R. Peterson, A. T. Buyukkocak, D. Aksaray, and Y. Yazıcıoglu, "Decentralized safe reactive planning under twtl specifications," in *2020 IEEE/RSJ International Conference on Intelligent Robots and Systems (IROS)*. IEEE, 2020, pp. 6599–6604.

[14] M. Leucker and C. Schallhart, "A brief account of runtime verification," *The Journal of Logic and Algebraic Programming*, vol. 78, no. 5, pp. 293–303, 2009.

[15] C. Sánchez, G. Schneider, W. Ahrendt, E. Bartocci, D. Bianculli, C. Colombo, Y. Falcone, A. Francalanza, S. Krstić, J. M. Lourenço *et al.*, "A survey of challenges for runtime verification from advanced application domains (beyond software)," *Formal Methods in System Design*, vol. 54, no. 3, pp. 279–335, 2019.

[16] A. Bauer, M. Leucker, and C. Schallhart, "Runtime verification for ltl and tltl," *ACM Transactions on Software Engineering and Methodology (TOSEM)*, vol. 20, no. 4, pp. 1–64, 2011.

[17] J. V. Deshmukh, A. Donzé, S. Ghosh, X. Jin, G. Juniwal, and S. A. Seshia, "Robust online monitoring of signal temporal logic," *Formal Methods in System Design*, vol. 51, no. 1, pp. 5–30, 2017.

[18] S. Zudaire, F. Gorostiaga, C. Sánchez, G. Schneider, and S. Uchitel, "Assumption monitoring using runtime verification for uav temporal task plan executions," in *2021 IEEE International Conference on Robotics and Automation (ICRA)*. IEEE, 2021, pp. 6824–6830.

[19] H.-M. Ho, J. Ouaknine, and J. Worrell, "Online monitoring of metric temporal logic," in *International Conference on Runtime Verification*. Springer, 2014, pp. 178–192.

[20] E. Bartocci, J. Deshmukh, A. Donzé, G. Fainekos, O. Maler, D. Ničković, and S. Sankaranarayanan, "Specification-based monitoring of cyber-physical systems: a survey on theory, tools and applications," in *Lectures on Runtime Verification*. Springer, 2018, pp. 135–175.

[21] G. Roşu and K. Havelund, "Rewriting-based techniques for runtime verification," *Automated Software Engineering*, vol. 12, no. 2, pp. 151–197, 2005.

[22] F. Lima, C. N. D. Carvalho, M. B. S. Acardi, E. G. D. Santos, and A. A. Massote, "Digital manufacturing tools in the simulation of collaborative robots: Towards industry 4.0," 2019.

[23] N. Koenig and A. Howard, "Design and use paradigms for gazebo, an open-source multi-robot simulator," in *2004 IEEE/RSJ International Conference on Intelligent Robots and Systems (IROS)(IEEE Cat. No. 04CH37566)*, vol. 3. IEEE, 2004, pp. 2149–2154.

[24] J. M. O'Kane, "A gentle introduction to ROS."

[25] Y. Minsky, A. Madhavapeddy, and J. Hickey, *Real World OCaml: Functional programming for the masses.* " O'Reilly Media, Inc.", 2013.