Safe Planning through Incremental Decomposition of Signal Temporal Logic Specifications

Parv Kapoor[1], Eunsuk Kang[1], and Rômulo Meira-Góes[2]

[1] Carnegie Mellon University, Pittsburgh, PA, USA
[2] Pennsylvania State University, State College, PA, USA
parvk@cs.cmu.edu, eunsukk@andrew.cmu.edu, romulo@psu.edu

Abstract. *Trajectory planning* is a critical process that enables autonomous systems to safely navigate complex environments. *Signal temporal logic (STL)* specifications are an effective way to encode complex, temporally extended objectives for trajectory planning in cyber-physical systems (CPS). However, the complexity of planning with STL using existing techniques scales exponentially with the number of nested operators and the time horizon of a given specification. Additionally, poor performance is exacerbated at runtime due to limited computational budgets and compounding modeling errors. Decomposing a complex specification into smaller subtasks and incrementally planning for them can remedy these issues. In this work, we present a method for decomposing STL specifications to improve planning efficiency and performance. The key insight in our work is to encode all specifications as a set of basic constraints called *reachability* and *invariance constraints*, and schedule these constraints sequentially at runtime. Our experiment shows that the proposed technique outperforms the state-of-the-art trajectory planning techniques for both linear and non-linear dynamical systems.

Keywords: Signal Temporal Logic · Planning · Cyber Physical Systems

1 Introduction

Most autonomous robots interacting with the physical world need to achieve complex objectives while dealing with uncertainty and stochasticity in their environment. This problem is exacerbated by short response times expected while ensuring runtime efficiency. Hence, formulating these complex objectives accurately is a crucial step in realizing the desired behaviors for robotic operations.

Temporal logics such as *linear temporal logic (LTL)* [19] and *signal temporal logic (STL)* [16] provide a precise way to encode objectives that are expressed in a natural language. STL has received special attention in the community due to its rich quantitative semantics that can quantitatively measure satisfaction of a given property that encodes an objective. Additionally, it can be used to describe complex properties over real valued signals such as state trajectories arising from continuous dynamical systems. For robotic planning, STL can be used to

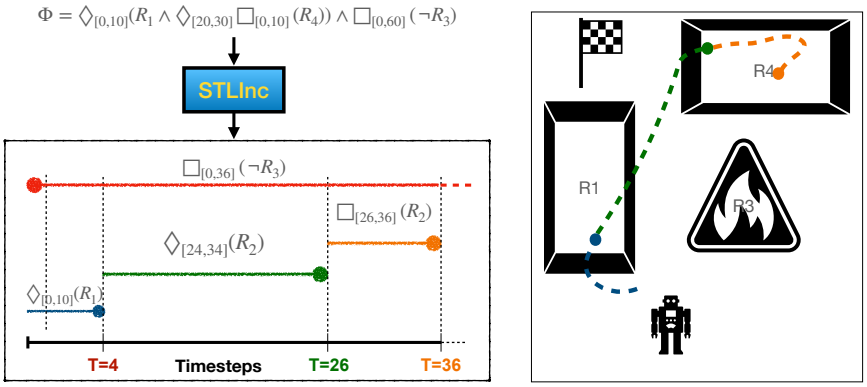


Fig. 1. Left: An STL specification ϕ with multiple nested temporal operators and a possible decomposition into subtasks. **Right**: A sample trajectory that satisfies ϕ in a planar environment.

describe complex behaviors with concrete time deadlines such as those found in trajectory planning and task planning. Planners can use these specifications to generate specification-conforming behavior.

A significant amount of common robotic objectives can be interpreted as a sequence of subtasks. It has been shown that incremental subtask planning can be done more efficiently compared to planning for a composite task [5, 7, 18]. However, when STL is used to represent these composite tasks, incremental planning becomes challenging. This issue is because STL semantics can encode the sequential nature of tasks but does not expose this structure to the planner. In such cases, the planners are forced to work with complex long-horizon specifications. When the horizon of the specification is longer than the planning horizon, planners can often generate suboptimal or violating plans. This problem is exacerbated when planning occurs at runtime with computational constraints and compounding modeling errors [2].

In this work, we propose a theory to decompose long-horizon, arbitrarily nested specifications into sub-specifications that can be satisfied incrementally. We define recursive rules for decomposition and propose a novel scheduling algorithm for incremental task planning. The key insight here is to "divide and conquer" STL requirements while ensuring, by construction, that the resulting plan satisfies the original composite specification. We illustrate the effectiveness of our proposed approach over an experiment involving robot exploration problems with linear and non-linear dynamics. Our preliminary experiment shows that our approach is able to more efficiently generate plans for complex, composite specifications in comparison to the existing state-of-the-art STL-based planning methods. In addition, our decomposition technique is agnostic to the

underlying system dynamics and the choice of planner, and can potentially be adapted by different planners.

The key contributions of this paper are:

- A method for decomposing an STL specification into a set of smaller STL specifications that represent subtasks (Section 4.2);
- A planning algorithm that incrementally schedules and executes these subtasks (Sections 4.3 and 4.4);
- An evaluation of the proposed approach over a benchmark of motion planning tasks (Section 5).

2 Motivation

We illustrate the problem of planning from complex specifications using an example from the motion planning domain. We use a planning problem similar to the one defined in [14].

As illustrated in Figure 1, the goal of the agent (robot) is to visit regions R_1 and R_4 sequentially while avoiding an unsafe region, R_3. Additionally, upon reaching R_4, the agent needs to stay in it for 10 time steps. We combine three common motion planning patterns such as sequenced visit, stabilization, and global avoidance to create a specification with timed deadlines as follows:

$$\begin{aligned}\Phi &= \phi_1 \wedge \phi_2 \\ \phi_1 &= \Diamond_{[0,10]}(R_1 \wedge \Diamond_{[20,30]}\Box_{[0,10]}(R_4)) \\ \phi_2 &= \Box_{[0,60]}(\neg R_3)\end{aligned}$$

The state-of-the-art (SOTA) technique for planning from Φ, originally proposed in [20], involves encoding the STL specification and the system dynamics as Mixed Integer Program (MIP) constraints and solving the constrained optimization problem in a receding horizon fashion. A new binary decision variable is introduced for each atomic proposition per time step in the STL specification. A known drawback of this technique is its exponential worst-case complexity with respect to the number of binary variables [12]. Various encoding modifications have been suggested to enhance the efficiency of the technique by reducing the number of variables and constraints [21, 12].

However, even with reduced variable encoding, current methods excel primarily with short-horizon specifications. When encoding nested temporal operators, a large number of additional variables and constraints are needed to capture the relationship between different temporal operators, in contrast to non-nested operators, where temporal constraints associated with each operator are considered independently. For example, let us take the subformula $\Diamond_{[20,30]}\Box_{[0,10]}(R_4)$ from ϕ_1. For encoding this subformula into an MIP, we would need 11 (outer eventually) + 121 (inner always) = 132 binary variables.[3] In general, as the nesting depth increases, the number of variables can increase exponentially.

[3] For the outer $\Diamond$ clause, 11 binary variables are introduced to encode that the inner $\Box$ clause is satisfied within interval [20,30]; for each time point in [0,10] interval, another set of 11 variables are introduced, thus resulting in 11*11 = 121 variables.

In this work, we propose a technique to improve the scalability of STL planning algorithms through decomposition of STL specifications. Our idea is inspired by human planning, where long-term goals are achieved by breaking tasks into incremental sub-goals [8]. Concretely, by decomposing the specification, we effectively remove the complexity of nested operators and also reduce the length of the lookahead horizon.

A possible decomposition of the specification Φ into four subtasks is as follows:

$$
\begin{aligned}
sch_1 &= \Diamond_{[0,10]}(R1) \wedge \Box_{[0,10]}(\neg R_3) \\
sch_2 &= \Diamond_{[t_{R1}+20, t_{R1}+30]}(R_4) \wedge \Box_{[t_{R1}+20, t_{R1}+30]}(\neg R_3) \\
sch_3 &= \Box_{[t_{R4}+0, t_{R4}+10]}(\neg R_3 \wedge R_4) \\
sch_4 &= \Box_{[t_{R42}, 60]}(\neg R_3)
\end{aligned}
$$

Here, the symbolic time variables $(t_{R1}, t_{R4}, t_{R42})$ indicate when those subtasks get satisfied. More specifically, t_{R4} indicates when the agent reaches Region 1 and t_{R42} indicates when the agent has been inside Region 4 for 10 timesteps after reaching it. These variables then shift the time intervals of the other constraints that depend on them (e.g., time t_{R1} from sch_1 is used to concretize the time intervals for sch_2, whose time of satisfaction, in turn, influences sch_3). These subtasks have shorter time horizons and no nested temporal operators, resulting in MIP constraints that are less complex than those that would result from composite specifications. As shown later in Section 5, this decomposition-based approach has potential to significantly improve the efficiency of planning.

3 Preliminaries

STL is a logical formalism used to define properties of continuous time real valued signals [16]. A signal $\mathbf{s}$ is a function $\mathbf{s} : \mathbb{T} \to \mathbb{R}^n$ that maps a time domain $T \subseteq \mathbb{R}_{\geq 0}$ to a real valued vector. Then, an STL formula is defined as:

$$\phi := \mu \mid \neg\phi \mid \phi \wedge \psi \mid \phi \vee \psi \mid \phi \, \mathcal{U}_{[a,b]} \, \psi$$

where μ is a predicate on the signal $\mathbf{s}$ at time t in the form of $\mu \equiv \mu(\mathbf{s}(t)) > 0$ and $[a, b]$ is the time interval (or simply I). The *until* operator $\mathcal{U}$ defines that ϕ must be true until ψ becomes true within a time interval $[a, b]$. Two other operators can be derived from *until*: *eventually* ($F_{[a,b]} \ \phi := \top \ \mathcal{U}_{[a,b]} \ \phi$) and *always* ($G_{[a,b]} \ \phi := \neg F_{[a,b]} \ \neg\phi$).

Definition 1. *Given a signal s_t representing a signal starting at time t, the Boolean semantics of satisfaction of $s_t \models \phi$ are defined inductively as follows:*

$$
\begin{aligned}
s_t \models \mu &\iff \mu(s(t)) > 0 \\
s_t \models \neg\varphi &\iff \neg(s_t \models \varphi) \\
s_t \models \varphi_1 \wedge \varphi_2 &\iff (s_t \models \varphi_1) \wedge (s_t \models \varphi_2) \\
s_t \models \mathrm{F}_{[a,b]}(\varphi) &\iff \exists t' \in [t+a, t+b] \text{ s.t. } s_{t'} \models \varphi \\
s_t \models \mathrm{G}_{[a,b]}(\varphi) &\iff \forall t' \in [t+a, t+b] \text{ s.t. } s_{t'} \models \varphi
\end{aligned}
$$

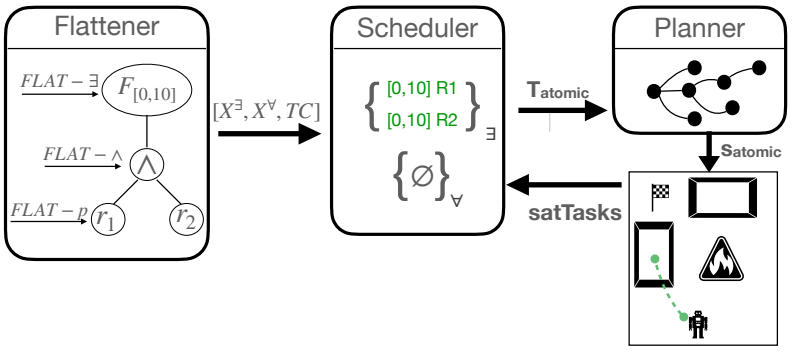


Fig. 2. Overview of the StlInc approach

Apart from the Boolean semantics, quantitative semantics are defined for a signal to compute a real-valued metric indicating *robustness*, i.e., the strength of satisfaction or violation.

Definition 2. *Given a signal s_t representing a signal starting at time t, the quantitative semantics of satisfaction of $s_t \models \phi$ are defined inductively as follows:*

$$\begin{aligned}
\rho(s_t, \mu_c) &= \mu(x_t) - c \\
\rho(s_t, \neg\varphi) &= -\rho(s_t, \varphi) \\
\rho(s_t, \varphi_1 \wedge \varphi_2) &= \min(\rho(s_t, \varphi_1), \rho(s_t, \varphi_2)) \\
\rho(s_t, \mathrm{F}_{[a,b]}(\varphi)) &= \max_{t' \in [t+a, t+b]} \rho(s'_t, \varphi) \\
\rho(s_t, \mathrm{G}_{[a,b]}(\varphi)) &= \min_{t' \in [t+a, t+b]} \rho(s'_t, \varphi)
\end{aligned}$$

For example, suppose that we are given (1) $\phi \equiv \mathrm{G}_{[0,3]}(\texttt{distToR3}(t) \geq 3.0)$, which states that the agent should maintain at least 3.0 meters away from region R_3 for the next 4 time steps and (2) signal s_t that contains sequence $\langle 3.0, 2.5, 3.0, 3.5 \rangle$ for `distToR3`. Evaluting the robustness of satisfaction of ϕ over s_t would result in a value of -0.5, implying that the agent violates the property by a degree of 0.5 (i.e., it fails to stay away from R_3 by 0.5 meters).

4 Approach

4.1 Basic Concepts and Definitions

The overview of our planning framework (StlInc) is shown in Figure 2. The key idea behind this approach is that a bounded STL formula in our fragment can be decomposed into a finite set of the following two types of *task constraints*, each of which is associated with time interval $I = [a, b]$ and state proposition p:

Reachability: The system ensures that p holds over at least one time step t within I.
Invariance: The system ensures that p holds over every step t within I.

Based on this idea, the framework carries out the incremental planning process over three steps. First, the *flattener* takes a user-specified STL specification (ϕ) and decomposes it into two sets of task constraints, $\mathcal{X}^{\exists}$ and $\mathcal{X}^{\forall}$, which contain the reachability and invariance constraints, respectively. The decomposition is performed such that satisfying all of the constraints in these two sets implies the satisfaction of the original formula ϕ.

Next, the *scheduler* takes the two sets, $\mathcal{X}^{\exists}$ and $\mathcal{X}^{\forall}$, and generates a sequence of *atomic tasks*, $\sigma = \langle at_0, at_1 ... at_k \rangle$, where (1) each atomic task at is a non-nested STL formula consisting of $G_{[a,b]}(p)$ or $F_{[a,b]}(p)$ (where p is a propositional formula) and (2) two different atomic tasks do not overlap in their time intervals. After this sequence is generated, the *planner* executes these atomic tasks one by one in the designated order. Once all tasks are executed, the system will have fulfilled $\mathcal{X}^{\exists}$ and $\mathcal{X}^{\forall}$, thus satisfying the original goal of ϕ. The rest of this section describes each of the three steps in detail.

STL fragment Our approach is designed to handle specifications written in the following fragment of STL:

$$\phi ::= F_{[a,b]}(\phi) \mid G_{[a,b]}(\phi) \mid \phi_1 \wedge \phi_2 \mid p$$
$$p ::= p_1 \wedge p_2 \mid p_1 \vee p_2 \mid \neg p \mid \mu$$

where p is a propositional formula that does not contain any temporal operator, and μ is an atomic proposition. We target this STL fragment as (1) it signficantly simplifies the decomposition process and (2) it is still expressive enough to capture many common behavioral patterns in the robotics planning domain [17]. Note that the fragment allows a nesting of temporal operators of an arbitrary depth, which is important for specifying sequential tasks. For example, the objective of *visiting* locations in a particular order can be defined as:

$$\phi \equiv F_{[1,5]}(l_1 \wedge F_{[1,10]}(l_2 \wedge F_{[1,7]}(l_3)))$$

where l_1, l_2, l_3 represent the locations to be visited.

Task constraints As mentioned at the beginning of the section, an STL formula imposes two types of constraints over system behavior: *reachability* and *invariance* constraints, which are formally defined as follows:

Definition 3 (Reachability set). *A reachability set, $\mathcal{X}^{\exists} \in \mathbb{P}(\mathbb{T} \times \mathbb{T} \times \mathbb{U})$ is a set of tuples of form (l, h, p), each stating that there exists some time t in interval $[l, h]$, inclusively, such that proposition p holds over the system state at time t.*

Definition 4 (Invariance set). *An invariance set, $\mathcal{X}^{\forall} \in \mathbb{P}(\mathbb{T} \times \mathbb{T} \times \mathbb{U})$, is a set of tuples of form (l, h, p), each stating that for every time t in interval $[l, h]$, inclusively, proposition p holds over the system state at time t.*

$$\frac{\phi = p}{\begin{array}{l}\mathcal{X}^{\exists} = \{(0,0,p)\} \quad \mathcal{X}^{\forall} = \emptyset \\ TC = \emptyset\end{array}}\ \textsc{Flat-}p \qquad \frac{\begin{array}{c}\phi = \phi_1 \wedge \phi_2 \\ (\mathcal{X}_1^{\exists}, \mathcal{X}_1^{\forall}, TC_1) = \textsc{Flat}(\phi_1) \\ (\mathcal{X}_2^{\exists}, \mathcal{X}_2^{\forall}, TC_2) = \textsc{Flat}(\phi_2)\end{array}}{\begin{array}{c}\mathcal{X}^{\exists} = \mathcal{X}_1^{\exists} \cup \mathcal{X}_2^{\exists} \\ \mathcal{X}^{\forall} = \mathcal{X}_1^{\forall} \cup \mathcal{X}_2^{\forall} \\ TC = TC_1 \cup TC_2\end{array}}\ \textsc{Flat-}\wedge$$

$$\frac{\phi = F_{[a,b]}\phi_1 \qquad (\mathcal{X}_1^{\exists}, \mathcal{X}_1^{\forall}, TC_1) = \textsc{Flat}(\phi_1)}{\begin{array}{l}\mathcal{X}^{\exists} = \{(t+l, t+h, p) \mid \exists (l,h,p) \in \mathcal{X}_1^{\exists}\} \\ \mathcal{X}^{\forall} = \{(t+l, t+h, p) \mid \exists (l,h,p) \in \mathcal{X}_1^{\forall}\} \\ TC = TC_1 \cup \{(a,b,t)\}\end{array}}\ \textsc{Flat-}\exists$$

$$\frac{\phi = G_{[a,b]}\phi_1, \quad (\mathcal{X}_1^{\exists}, \mathcal{X}_1^{\forall}, TC_1) = \textsc{Flat}(\phi_1)}{\begin{array}{l}\mathcal{X}^{\exists} = \{(k+l, k+l, p) \mid \exists (l,h,p) \in \mathcal{X}_1^{\exists} \wedge k \in [a,b]\} \\ \mathcal{X}^{\forall} = \{(a+l, b+h, p) \mid \exists (l,h,p) \in \mathcal{X}_1^{\forall}\} \\ TC = TC_1\end{array}}\ \textsc{Flat-}\forall$$

Fig. 3. Flattening rules

We introduce an additional set of constraints that specify intervals over symbolic time variables that are introduced during the flattening of F formulas:

Definition 5 (Time variable intervals). *A time variable interval set, $TC \in \mathbb{P}(\mathbb{T} \times \mathbb{T} \times \mathbb{T})$, is a set of tuples of the form (l, h, v), each stating that symbolic time variable v takes a value in the interval $[l, h]$, inclusively.*

4.2 Flattening

Given an input STL specification, ϕ, the goal of *flattening* is to construct two sets of constraints—reachability and invariance sets—whose satisfaction also implies the satisfaction of ϕ. Flattening is applied recursively based on the structure of ϕ, as shown through the rules in Figure 3. Along with $\mathcal{X}^{\forall}$ and $\mathcal{X}^{\exists}$, each recursive step produces an additional auxiliary output TC, which is later used by the scheduler to resolve symbolic time variables.

Flat-p In the basic case where $\phi = p$, flattening generates one reachability constraint that requires p to be satisfied at the current time (i.e., $l = h = 0$). Hence, the reachability set for ϕ is $(0, 0, p)$.

Flat-$\wedge$ Given cojunctive formula $\phi = \phi_1 \wedge \phi_2$, the invariance set for ϕ is the union of $\mathcal{X}_1^{\forall}$ and $\mathcal{X}_2^{\forall}$; i.e., every invariance constraint in ϕ_1 and ϕ_2 must be satisfied. Similarly, the reachability set for ϕ is the union of $\mathcal{X}_1^{\exists}$ and $\mathcal{X}_2^{\exists}$; i.e., every reachability constraint in ϕ_1 and ϕ_2 must be satisfied.

Flat-$\exists$ Given $\phi = F_{[a,b]}(\phi_1)$, for each constraint $(l_1, h_1, p_1) \in \mathcal{X}_1^{\exists}$, flattening involves shifting interval $[l_1, h_1]$ by a symbolic time variable $t \in [a, b]$. This

bound on t is encoded by adding a tuple (a, b, t) to TC. Intuitively, this can be understood from the following transformation similar to the one in [13]:

$$F_{[a,b]}(F_{[c,d]}(\phi_1)) = F_{[a+c,b+d]}(\phi_1)$$

Here, instead of adding a and b directly to the upper and lower bound, respectively of time interval $[c, d]$, we add a symbolic variable that can take a value between a and b. For example, given specification $\phi = F_{[2,5]}(p)$, $\mathcal{X}^{\exists}$ would contain constraint $(t + 0, t + 0, p)$ and TC would contain $(2, 5, t)$.

As another example, consider specification $\phi = F_{[2,5]}(p_2 \wedge F_{[1,3]}(p_1))$. When the FLAT-$\exists$ rule is applied, the resulting $\mathcal{X}^{\exists}$ contains $(t_2 + t_1 + 0, t_2 + t_1 + 0, p1)$ and $(t_2, t_2, p2)$, and TC contains $\{(1, 3, t_1), (2, 5, t_2)\}$. Here, t_1 is introduced when FLAT-$\exists$ rule is applied to the innermost F operator ($F_{[1,3]}$); t_2 is then introduced when FLAT-$\exists$ is applied for outermost F operator ($F_{[2,5]}$).

The flattening of an F formula applied to an invariance set is handled similarly. Consider a specification of the form

$$\phi \equiv F_{[a,b]}(\phi_1) = F_{[a,b]}(G_{[c,d]}(p))$$

with $\mathcal{X}_1^{\forall} = \{(c, d, p)\}$. Here, according to the Boolean STL semantics, there exists a $t \in [a, b]$ such that $\forall t' \in [t + c, t + d]$, p must be satisfied. Hence, in the resulting invariance set for ϕ, the time interval for the existing invariance constraint is shifted by t. This bound on t is encoded by adding a tuple (a, b, t) to TC. For example, given specifcation $\phi = F_{[2,5]}(G_{[3,10]}(p))$, $\mathcal{X}^{\forall}$ would contain constraint $(t + 3, t + 10, p)$ and TC would contain $(2, 5, t)$.

Flat-$\forall$ The flattening of an $\forall$ formula over $\mathcal{X}_1^{\exists}$ is handled in the following way. Consider a specification of the form:

$$\phi \equiv G_{[a,b]}(\phi_1) = G_{[a,b]}(F_{[c,d]}(p))$$

with $\mathcal{X}_1^{\exists} = \{tc\} = \{(t_1, t_1, p)\}$ and $TC = \{(c, d, t_1)\}$. Since ϕ states that constraint tc must hold at every time step between $[a, b]$, the idea is to create multiple reachability constraints of form $(t_1 + k, t_1 + k, p)$, one for each time value k in interval $[a, b]$. For example, let $\phi = G_{[1,100]}(F_{[1,5]}(p))$. After flattening, $\mathcal{X}^{\exists}$ for ϕ would contain 100 reachability constraints, each in form of $(t_1 + k, t_1 + k, p)$ for $1 \leq k \leq 100$.

The flattening of an $\forall$ over $\mathcal{X}_1^{\forall}$ is handled in the following way. Consider a specification of the form

$$\phi \equiv G_{[a,b]}(\phi_1) = G_{[a,b]}(G_{[c,d]}(p))$$

with $\mathcal{X}_1^{\forall} = \{tc\} = \{(c, d, p)\}$. Since ϕ states that constraint tc must hold at every time step between $[a, b]$, by shifting the interval of each constraint by $[a, b]$, the resulting formula $\mathcal{X}^{\forall}$ is constructed as $\{(a + c, b + d, p)\}$. Intuitively, this can be understood from the following transformation similar to the one in [13]:

$$G_{[a,b]}(G_{[c,d]}(\phi_1)) = G_{[a+c,b+d]}(\phi_1)$$

We now introduce a theorem that establishes the soundness of our flattening operation with respect to a given STL specification.

Lemma 1. *Let ϕ be an input STL specification, and let ($\mathcal{X}^{\exists}$, $\mathcal{X}^{\forall}$, TC) be the output of $flatten(\phi)$. Then, for every signal s_t:*

$$\forall v \in \mathcal{V}(vars(TC)) \bullet \forall x \in \mathcal{X}^{\exists} \cup \mathcal{X}^{\forall} \bullet s_t \models Inst(x, v) \implies s_t \models \phi$$

where vars is a function that returns the set of all symbolic variables in TC, $\mathcal{V}(\mathcal{X})$ is the set of all possible assignments of values to variables in $\mathcal{X}$ (restricted to values in their respective time intervals), and $Inst(x, v)$ instantiates symbolic variables in constraint x with the values from v.

In other words, if every possible reachability or invariance constraint is satisfied over a particular signal, then the original specification ϕ must also hold over the same signal.

4.3 Symbolic Time Resolution

The $\mathcal{X}^{\exists}$ and $\mathcal{X}^{\forall}$ constraints generated from flattening may contain multiple symbolic variables. To enable scheduling, StlInc first attempts to resolve as many of these variables as possible by applying substitution rules to the constraints.

Algorithm 1 shows the sketch of the *symbolic time resolution (STR)* process. As inputs, it takes the output of the flattening procedure—the $\mathcal{X}^{\exists}$ and $\mathcal{X}^{\forall}$ constraints, and the time intervals over symbolic time variables. As outputs, it produces (1) a new pair of $\mathcal{X}^{\exists}$ and $\mathcal{X}^{\forall}$, with some of the time variables replaced by concrete time values, and (2) time variable intervals (TC') that specifies, for each concrete constraint produced in (1), an interval during which the constraint must be satisfied. This latter set of time intervals are used to enforce conjunctive constraints (i.e., $\phi_1 \wedge \phi_2$) to be satisfied simultaneously.

The algorithm iterates through the time variables in the bottom-up order (i.e., starting with the variables that appear in the lower part of an AST for a given STL expression). For each constraint that contains variable t (line 4), STR applies two types of substitutions, depending on whether the constraint is a reachability or an invariance constraint.

Invariance constraints An invariance constraint, $x = (l, h, p)$, with time variable $t \in [a, b]$ represents an STL expression $F_{[a,b]}(G_{[l,h]}\ p)$ (where t appears in l and h, and p is an atomic proposition). Intuitively, this expression can be regarded a kind of "reach and stay" task, where the system must first reach a state where p holds within time interval $[a + l, b + l]$, and then continue to satisfy p for the following $(h - l)$ time steps. The rule ApplyFG takes constraint x and interval tc, and produces a new pair of reachability and invariance constraints, as follows:

$$\textsc{ApplyFG}(x = (l, h, p), tc = (a, b, t)) \equiv ((a + l, b + l, p), (t_{sat}, t_{sat} + h - l, p))$$

Algorithm 1 Symbolic Time Resolution (STR)

1: **Input**: Flattened constraints $\mathcal{X}^{\exists}, \mathcal{X}^{\forall}$, time variable intervals TC
2: **Output**: Modified constraints $\mathcal{X}^{\exists}, \mathcal{X}^{\forall}$, new time variable intervals TC'
3: $TC' = \emptyset$
4: **for** $tc = (a, b, t)$ in TC **do**
5: $\quad \mathcal{X}_t^{\forall} :=$ FILTER$(\mathcal{X}^{\forall}, t)$, $\mathcal{X}_t^{\exists} :=$ FILTER$(\mathcal{X}^{\exists}, t)$
6: $\quad$ **for** $x = (l, h, p)$ in $\mathcal{X}_t^{\forall}$ **do**
7: $\quad\quad (y^{\exists}, y^{\forall}) :=$ APPLYFG(x, tc)
8: $\quad\quad \mathcal{X}^{\exists} := \mathcal{X}^{\exists} \cup \{y^{\exists}\}$
9: $\quad\quad \mathcal{X}^{\forall} := (\mathcal{X}^{\forall} - \{x\}) \cup \{y^{\forall}\}$
10: $\quad\quad TC' := TC' \cup \{(t + l, t + l, \text{T-SAT}(y^{\exists})), (t + l + 1, t + h, \text{T-SAT}(y^{\forall}))\}$
11: $\quad$ **end for**
12: $\quad$ **for** $x = (l, h, p)$ in $\mathcal{X}_t^{\exists}$ **do**
13: $\quad\quad y^{\exists} :=$ APPLYFF(x, tc)
14: $\quad\quad \mathcal{X}^{\exists} := (\mathcal{X}^{\exists} - \{x\}) \cup \{y^{\exists}\}$
15: $\quad\quad TC' := TC' \cup \{(t + l, t + h, \text{T-SAT}(y^{\exists}))\}$
16: $\quad$ **end for**
17: **end for**
18: **return** $(\mathcal{X}^{\exists}, \mathcal{X}^{\forall}, TC')$

where t_{sat} is a new symbolic variable that represents the time at which p is satisfied between $(a + l, b + l)$.

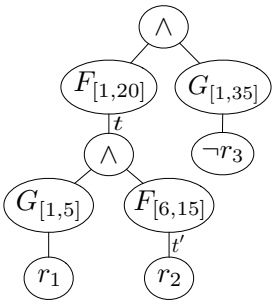
In addition, STR adds two time variable intervals (line 10) to ensure that: (1) the new reachability constraint, $y^{\exists}$, is satisfied exactly at $(t + l)$ and (2) the invariance constraint, $y^{\forall}$, is satisfied subsequently for the following $(l - h)$ steps. Here, T-SAT(y) returns a symbolic time variable representing the time of satisfaction of constraint y.

Consider the example in Figure 4. One of the invariance constraints that flattening generates is $(t+1, t+5, r_1)$, which depends on time variable $t \in [1, 20]$. The application of APPLYFG results in constraints stating that (1) the system must satisfy p within [2,21], and (2) from the point of the satisfaction of this constraint (t_2), it must hold p true for the following $(5 - 1) = 4$ steps. Note that the other invariance constraint, $(1, 35, \neg r_3)$, remains untouched, as it does not depend on any time variable.

Reachability constraints A reachability constraint, $x = (l, h, p)$, with time variable $t \in [a, b]$ represents an STL expression $F_{[a,b]}(F_{[l,h]}\ p)$. When a pair of F operators are nested in this manner, they can be simplified by using the following substitution rule:

$$\text{APPLYFF}(x = (l, h, p), tc = (a, b, t)) \equiv (l + a, h + b, p)$$

The resulting reachability constraint, $y^{\exists}$, replaces the existing constraint in $\mathcal{X}^{\exists}$ (line 14). In addition, a new time variable interval is added to ensure that $y^{\exists}$ is satisfied between $(t + l)$ and $(t + h)$.



Flattening output:

$\mathcal{X}^{\exists}$ $(t+t', t+t', r_2)$

$\mathcal{X}^{\forall}$ $(t+1, t+5, r_1)$
$(1, 35, \neg r_3)$

TC $t \in [1, 20]$
$t' \in [6, 15]$

Resolution output:

$\mathcal{X}^{\exists}$ $(6, 35, r_2)_{id=1}$
$(2, 21, r_1)_{id=2}$

$\mathcal{X}^{\forall}$ $(t_2, t_2+4, r_1)_{id=3}$
$(1, 35, \neg r_3)_{id=4}$

TC $t_1 \in [t+6, t+15]$
$t_2 \in [t+1, t+1]$
$t_3 \in [t+2, t+5]$

Fig. 4. An AST for an example STL specification, $F_{[1,20]}(G_{[1,5]}(r_1) \wedge F_{[6,15]}(r_2)) \wedge G_{[1,35]}(\neg r_3)$, with outputs from the flattening and symbolic time resolution steps. Each constraint resulting from the resolution step is assigned an identifier (id); a symbolic time variable that represents the time of satisfying the constraint is annotated with id as the subscript (i.e., t_2 represents the satisfaction of reachability constraint $(2, 21, r_1)$).

For the example in Figure 4, flattening produces one reachability constraint, $(t+t', t+t', r_2)$. During the first iteration of the outermost loop (line 4), STR selects $tc = (5, 15, t')$ and applies ApplyFF to the constraint, producing a new reachability constraint $(t+5, t+15, r_2)$. In next iteration, STR selects $tc = (1, 20, t)$ and applies ApplyFF to $(t+5, t+15, r_2)$, producing an additional reachability constraint $(6, 35, r_2)$.

Note that at the end of resolution for the example (Figure 4), variable t appears in multiple intervals in TC'. In the following section, we describe how StlInc schedules tasks to generate concrete values for the time variables incrementally one-by-one, ultimately synthesizing a plan that satisfies the original STL formula.

Lemma 2. *Let ($\mathcal{X}^{\exists}$, $\mathcal{X}^{\forall}$, TC) be the output of $flatten(\phi)$ and ($\mathcal{X}_2^{\exists}$, $\mathcal{X}_2^{\forall}$, TC') be the output of $STR(\mathcal{X}^{\exists}, \mathcal{X}^{\forall}, TC)$. Then, for every signal s_t:*

$$\forall v \in \mathcal{V}(vars(TC')) \bullet (\forall x_2 \in \mathcal{X}_2^{\exists} \cup \mathcal{X}_2^{\forall} \bullet s_t \models Inst(x_2, v)) \implies (\forall x \in \mathcal{X}^{\exists} \cup \mathcal{X}^{\forall} \bullet s_t \models Inst(x, v))$$

In other words, if every reachability and invariance constraint generated from STR is satisfied under some instantiation (v) of symbolic time variables in TC', then the original set of flattened constraints must also be satisfied under the same condition.

4.4 Scheduling

Given the output from the resolution step (the two constraint sets, $\mathcal{X}^{\exists}, \mathcal{X}^{\forall}$ and the time interval variables, TC'), the goal of the scheduler is synthesize a plan that satisfies the original STL specification ϕ. To achieve this, the scheduler iteratively interacts with a *planner* that is capable of synthesizing a plan to

Algorithm 2 Schedule

1: **Input**: $\mathcal{X}^{\exists}, \mathcal{X}^{\forall}, TC'$
2: **Output**: Signal $\mathbf{s}_{plan}$ for synthesized plan
3: $\mathbf{s}_{plan} = \langle\rangle$
4: $\prec$:= ComputeOrder($\mathcal{X}^{\exists}, \mathcal{X}^{\forall}, TC'$)
5: $currTasks$:= NextTasks($\mathcal{X}^{\exists}, \mathcal{X}^{\forall}, \prec, \emptyset$)
6: **while** $currTasks \neq \emptyset$ **do**
7: $currTasks$:= Slice($currTasks$)
8: $atomicTask$:= NextAtomic($currTasks$)
9: $currTasks := currTasks - \{atomicTask\}$
10: $\mathbf{s}_{atomic}$:= Plan ($atomicTask$)
11: **if** $\mathbf{s}_{atomic} = \langle\rangle$ **then break end if**
12: $\mathbf{s}_{plan} := \mathbf{s}^{\frown}\mathbf{s}_{atomic}$
13: satTasks := ExtractTime($\mathbf{s}_{atomic}$)
14: $currTasks := currTasks \cup$ NextTasks($\mathcal{X}^{\exists}, \mathcal{X}^{\forall}, \prec, satTasks$)
15: **end while**
16: **return** $\mathbf{s}_{plan}$

satisfy an *atomic task* formula of form $F_{[a,b]}(\phi_1) \wedge G_{[a,b]}(\phi_2)$, where ϕ_1 and ϕ_2 are quantifier-free STL expressions. The scheduling process is incremental: The scheduler generates a sequence of atomic tasks formulas and invokes the planner to solve them one-by-one, using information (i.e., the time of satisfaction of a reachability or invariance constraint) generated by the planner to resolve any dependencies on symbolic time variables that were introduced during the flattening and resolution steps. The scheduling algorithm (Alg. 2) comprises of three major parts: ordering, slicing of constraints, and planning of atomic tasks.

Ordering In the first step (line 4), the scheduler computes a partial order ($\prec$) among the given reachability and invariance constraints, to determine which of these constraints must be satisfied before others. In particular, given a pair of constraints, x_1 and x_2, $x_1 \prec x_2$ if and only if the time of satisfaction of x_1 necessarily precede that of x_2, based on the time intervals that are assigned to those constraints in $\mathcal{X}^{\exists}$ or $\mathcal{X}^{\forall}$.

If one or more of x_1 and x_2 depends on another symbolic variable, t, for satisfaction, then the information in TC' is used to determine the presence of a precedence relationship. Consider t_1 and t_3 from Figure 4; after resolution, the satisfaction of these two constraints depend on the symbolic variable t (as specified in TC'). Although the value of t is unknown, it can be determined that for any possible value of t, t_3 will necessarily be satisfied before t_1 (i.e., $t_3 \prec t_1$). Overall, for this example, ComputeOrder determines that $t_2 \prec t_3 \prec t_1$. Note that t_4 does not appear in this ordering as it needs to be satisfied in parallel with these other constraints.

Slicing The scheduler then determines the first set of constraints (or tasks) to be carried out based on the order $\prec$ (line 5). In general, the time intervals over these constraint may overlap with each other in an arbitrary way. Recall,

however, that each atomic task to be solved by the planner must be in form $F_{[a,b]}(\phi_1) \wedge G_{[a,b]}(\phi_2)$. Thus, the scheduler must first convert the constraints in $currTasks$ into a set of atomic tasks constraint; this step involves *slicing* one or more constraints in $currTasks$ (line 7).

Due to limited space, we provide the full details of the slicing algorithm in the appendix. We briefly illustrate it here using an example, shown in Figure 5. For our running example, the first set of constraints to be fulfilled is $\{t_2 = (2, 21, r_1), t_4 = (1, 35, \neg r_3)\}$. To generate atomic tasks out of these, the SLICE operation slices t_4 into three constraints, t_{4a}, t_{4b}, t_{4c}; this, in turn, results in the following three atomic tasks:

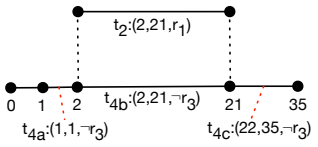


Fig. 5. A slicing example.

$$G_{[1,1]}(\neg r_3) \qquad F_{[2,21]}(r_2) \wedge G_{[2,21]}(\neg r_3) \qquad G_{[22,35]}(\neg r_3)$$

However, the slicing step may introduce dependencies among the atomic tasks, especially for the constraints in $\mathcal{X}^{\exists}$. For example, suppose that task $F_{[2,21]}(r_2)$ is further split into two slices, $F_{[2,12]}(r_2)$ and $F_{[12,21]}(r_2)$. Since r_2 needs to be satisfied only once in interval $[2, 21]$, we would need to ensure that we are not over-constraining the space of possible behaviors by requiring r_2 to be satisfied twice, as the slices would imply. To achieve this, we keep track of dependencies among constraints, which are used by the scheduler to remove unnecessary atomic tasks (e.g., remove $F_{[12,21]}(r_2)$ once r_2 is satisfied between $[2, 12]$). This dependency management is handled inside NEXTTASKS (Algorithm 2, Line 14).

Planning atomic tasks Once atomic tasks have been generated through slicing, the scheduler selects the next atomic task and invokes the planner (lines 8-10). If the planner is able to synthesize a plan that satisfies the atomic task, it returns a signal that represents the satisfying trajectory, which is then appended to the cumulative signal $\mathbf{s}_{plan}$ (line 12); if not, the scheduler terminates by returning the signal that contains a partially satisfactory plan (line 11).

From the synthesized signal, the scheduler extracts the constraints from $\mathcal{X}^{\exists}$ and $\mathcal{X}^{\forall}$ that were satisfied, along with the concrete time values for their satisfaction (line 13). This information ($satTasks$) is then used to determine the next increment of constraints to be solved.

Finally, when all of the constraints in $\mathcal{X}^{\exists}$ and $\mathcal{X}^{\forall}$ have been satisfied, the scheduler terminates by returning $\mathbf{s}_{plan}$ as the final plan.

Lemma 3. *Let s_{plan} be the output of Schedule($\mathcal{X}^{\exists}$, $\mathcal{X}^{\forall}$, TC') and v be an assignment of values to symbolic time variables in TC', which are determined during the scheduling step. Then, the following statement holds:*

$$\forall x \in \mathcal{X}^{\exists} \cup \mathcal{X}^{\forall} \bullet s_{plan} \models Inst(x, v)$$

In other words, the synthesized plan, s_{plan}, satisfies all of the reachability and invariance constraints that were generated from the preceding flattening and STR steps.

	STL Specification	Pattern
ϕ_1	$F_{[0,15]}(R_1) \wedge F_{[5,25]}(R_2) \wedge F_{[20,30]}(R_3) \wedge G_{[0,40]}(\neg O_1)$	R+A
ϕ_2	$F_{[0,15]}(R_1 \wedge F_{[0,15]}(R_2)) \wedge G_{[0,40]}(\neg O_1)$	SV+A
ϕ_3	$F_{[0,15]}(R_1 \wedge F_{[0,15]}(R_2 \wedge F_{[0,20]}(R_3 \wedge F_{[0,15]}(R_1))))$	SV
ϕ_4	$F_{[0,15]}G_{[0,10]}(R_1) \wedge F_{[0,35]}(R_2) \wedge G_{[0,40]}(\neg O_1)$	R+A+SB
ϕ_5	$F_{[0,15]}(R_1 \wedge F_{[0,20]}G_{[0,10]}(R_2))$	SV+SB

Table 1. Benchmark STL specifications created from motion planning patterns. Here, R: Reach, A: Avoid, SV: Sequenced Visit, SB: Stabilization.

Building on Lemmas 1, 2, and 3, we finally introduce a theorem to state that our proposed approach generates a plan that satisfies the given specification ϕ:

Theorem 1. *Given specification ϕ and s_{plan} as the output of the scheduling algorithm, $s_{plan} \models \phi$.*

5 Evaluation

This section begins with a detailed description of our experimental setup, including specifications and implementation details, alongside the research questions we aim to address. Following this, we present our findings and conclude with a discussion of our approach over the benchmarks.

5.1 Experimental Setup

Specifications. We investigated multiple motion planning STL specifications from [6, 17]. Based on the most common planning patterns, such as Reach (R), Avoid (A), Stabilisation (SB), Sequenced Visits (SV) etc., we created representative STL benchmark specifications as outlined in Table 1. These specifications are defined over STL subformulas of the form R_i or O_i where R_i / O_i is satisfied if the agent is inside Region i or Obstacle i. These subformulas are defined in a similar fashion using conjunction of linear and nonlinear predicates as done in [12]. Please refer to [12] for more information on how these are defined for rectangular/circular regions.

Implementation Details. We investigate planning from benchmark specifications in two robot exploration environments (similar to Figure 1), namely LinEnv and NonLinEnv created using STLPY [12]. STLPY has the functionality to encode any arbitrary STL formula, dynamics and actuation limits into constraints and use existing state-of-the-art solvers (Gurobi [10], SNOPT [9], etc.,) to generate satisfying plans. Our two environments are both planar but differ in underlying dynamics governing the robot. LinEnv has linear dynamics (Double Integrator) whereas NonLinEnv has nonlinear dynamics (Unicycle). We benchmark our technique against existing MICP methods (for linear dynamics) and

other gradient-based techniques like SNOPT (for non-linear dynamics). We use Python to implement our tool[4] while using stlpy and Drake [23] to encode the STL constraints. Additionally, we use Gurobi or SNOPT to solve the final constrained optimization problem. All experiments were run on a workstation with an Intel Xeon W-1350 processor and 32 GB RAM.

Benchmarks and Research Questions. We compare against the state-of-the-art techniques proposed in [4] (which we call *standard MICP*) and [12] (*reduced MICP*). Since the standard MICP encoding is only defined for environments with linear dynamics, we compare our technique against reduced MICP encoding for NonLinEnv. Reduced MICP claims better performance over standard MICP for long horizon and complex specifications due to their efficient encoding of disjunction and conjunction with fewer binary variables. However, standard MICP is faster for short-horizon specifications due to solver-specific presolve routines that leverage the additional binary variables for simplification.

Since our focus is on both short- and long-horizon specification with deep levels of temporal operator nesting, we benchmark against both techniques. The two main metrics we are concerned with are the time taken for solving and the final robustness values. To make the comparison fair, the total time taken by our technique includes the time taken by the flattener, scheduler, and solvers.

The two main research questions we investigate in this paper are:

1. **RQ1**: Does our decomposition technique result in shorter solve times?
2. **RQ2**: Does our decomposition technique result in higher robustness scores?

5.2 Results

Table 2 summarizes the results for StlInc performance compared to the baselines. In the tables, N represents the horizon of the specification and D represents the maximum depth of temporal nesting; TO represents a timeout, which means the solver did not terminate despite running it for 30 minutes. In those cases, the solver's output plan robustness is represented as -inf (which means no solution was found in the given time).

For LinEnv for all the specifications, our robustness values are comparable to the two techniques but our solve times are either lower or comparable to the baselines. Additionally, for specification ϕ_3, which has the deepest temporal nesting, our method significantly outperforms both baseline methods that experience timeouts.

For NonLinEnv for ϕ_1 and ϕ_4, the baseline encoding performs better in terms of solving time but StlInc only does slightly worse. However, for specification ϕ_2, ϕ_3 and ϕ_5, StlInc significantly outperforms the baselines.

[4] https://github.com/parvkpr/MCTSTL

Spec	N	D	Solve time (s)					Robustness				
			LinEnv			NonLinEnv		LinEnv			NonLinEnv	
			[4]	[12]	StlInc	[12]	StlInc	[4]	[12]	StlInc	[12]	StlInc
ϕ_1	40	0	**0.845**	2.698	0.891	**0.890**	1.464	0.500	0.500	**0.500**	**0.430**	0.572
ϕ_2	30	1	2.459	TO	**0.402**	12.674	**0.892**	0.491	-inf	**0.491**	-inf	**0.594**
ϕ_3	60	3	TO	TO	**0.874**	15.829	**1.554**	-inf	-inf	**0.228**	-inf	**0.065**
ϕ_4	40	2	**0.318**	0.330	0.629	**1.049**	1.131	**0.494**	0.500	0.500	0.470	**0.364**
ϕ_5	40	2	2.829	28.490	**0.694**	83.193	**1.776**	0.500	0.500	**0.500**	-inf	**0.596**

Table 2. StlInc Performance Benchmarking for LinEnv and NonLinEnv against standard MICP ([4]) and reduced MICP ([12]).

Summary. Our technique excels significantly for nesting depths > 1 in both LinEnv and NonLinEnv. However, for nesting depths < 1, the baseline techniques outperform us due to marginal overhead from flattening, scheduling, and solver invocations. The encoding for these specifications involves fewer binary variables, and the preprocessing overhead of using StlInc outweighs the performance benefits. Nevertheless, the experiment suggests that our technique is more efficient for multi-step tasks with deep temporal nesting, outperforming baselines by an order of magnitude.

6 Related Work

Trajectory synthesis from STL specifications is an active area of research for which multiple approaches have been proposed in the past few years [20, 2, 21, 1, 15]. One of the first papers in this direction involved translating STL specifications into constraints within a Mixed Integer Linear Program (MILP) [20]. This approach is sound and complete but faces scalability challenges for long-horizon specifications. To remedy this drawback, the original encoding has been modified by focusing on abstraction-based techniques [22] and reducing binary variables via logarithmic encoding [12]. Most of these techniques focus on reducing the MILP's complexity to observe performance benefits. Recently, the focus has shifted to developing techniques that leverage robustness feedback as a heuristic for trajectory synthesis instead of using MILP. These techniques involve using reinforcement learning [1, 11], search-based techniques [3] and control barrier functions [15] to generate STL satisfying trajectories. While these methods offer greater scalability, they are not complete and frequently struggle to accommodate complex specifications because of the intrinsically non-convex optimization problem posed by robustness semantics.

In this work, we focus on MILP-based techniques due to their completeness guarantees and improve their scalability by modifying input STL specifications themselves. However, since we perform structural manipulation of the specifications themselves, our decomposition technique is planner-agnostic and can also use learning- or search-based planners. Decomposition of STL specifications has

been studied before in [13, 24]. However, our work differs from existing work in multiple ways. In [24], the authors restrict themselves to an STL fragment that does not allow nesting of temporal operators, while a key contribution of our work is handling deep nested specifications. In [13], the authors perform structural manipulation using a tree structure. However, their focus is on multi-agent setups and they handle nested operators conservatively, especially for the eventually operator. This conservative notion generates specification satisfying behavior but it can be overly restrictive. Our interpretation is more flexible and in line with the Boolean semantics defined for the same operators.

7 Limitations and Future Work

In this work, we propose a structural manipulation-based technique for the temporal decomposition of STL specifications, enabling the incremental fulfillment of these specifications. We show our method generates correct-by -construction trajectories that satisfy deeply nested specifications with long time horizons for which existing baseline STL planning techniques struggle.

While the proposed approach is promising, our current decomposition technique does not handle disjunction. Additionally, the technique is sound but not complete, and designed to prioritize satisfaction over optimality. This limitation stems from incremental planning of objectives, which can be locally optimal compared to global planning, which considers the entire problem space. In future work, we aim to enrich our scheduling algorithm with backtracking capability, which can generate multiple satisfying plans, to overcome this limitation. Furthermore, for probabilistic systems, we plan to employ conformal prediction techniques to overcome compounding modeling error issues. Another avenue for future work is to adapt our theory to accommodate learning and heuristic-based approaches, such as reinforcement learning. Finally, our decomposition theory can potentially be employed for other STL applications, such as falsification, testing and runtime assurance and we plan to investigate that in the future.

Acknowledgments

We'd like to thank our reviewers for their insightful feedback. This work was supported in part by the National Science Foundation Award CCF-2144860.

References

1. Aksaray, D., Jones, A., Kong, Z., Schwager, M., Belta, C.: Q-learning for robust satisfaction of signal temporal logic specifications (2016)
2. Aloor, J.J., Patrikar, J., Kapoor, P., Oh, J., Scherer, S.: Follow the rules: Online signal temporal logic tree search for guided imitation learning in stochastic domains. In: 2023 IEEE International Conference on Robotics and Automation (ICRA). pp. 1320–1326 (2023). https://doi.org/10.1109/ICRA48891.2023.10160953

3. Aloor, J.J., Patrikar, J., Kapoor, P., Oh, J., Scherer, S.: Follow the rules: Online signal temporal logic tree search for guided imitation learning in stochastic domains. In: 2023 IEEE International Conference on Robotics and Automation (ICRA). pp. 1320–1326. IEEE (2023)
4. Belta, C., Sadraddini, S.: Formal methods for control synthesis: An optimization perspective. Annual Review of Control, Robotics, and Autonomous Systems **2**(1), 115–140 (2019). https://doi.org/10.1146/annurev-control-053018-023717, https://doi.org/10.1146/annurev-control-053018-023717
5. Botea, A., Ciré, A.A.: Incremental heuristic search for planning with temporally extended goals and uncontrollable events. In: Proceedings of the 21st International Joint Conference on Artificial Intelligence. p. 1647–1652. IJCAI'09, Morgan Kaufmann Publishers Inc., San Francisco, CA, USA (2009)
6. Chen, Y., Gandhi, R., Zhang, Y., Fan, C.: NL2TL: transforming natural languages to temporal logics using large language models. CoRR **abs/2305.07766** (2023). https://doi.org/10.48550/arXiv.2305.07766, https://doi.org/10.48550/arXiv.2305.07766
7. Czechowski, K., Odrzygóźdź, T., Zbysiński, M., Zawalski, M., Olejnik, K., Wu, Y., Kuciński, Ł., Miłoś, P.: Subgoal search for complex reasoning tasks. Advances in Neural Information Processing Systems **34**, 624–638 (2021)
8. Donnarumma, F., Maisto, D., Pezzulo, G.: Problem solving as probabilistic inference with subgoaling: Explaining human successes and pitfalls in the tower of hanoi. PLoS Computational Biology **12**(4) (2016)
9. Gill, P.E., Murray, W., Saunders, M.A.: Snopt: An sqp algorithm for large-scale constrained optimization. SIAM Rev. **47**(1), 99–131 (jan 2005). https://doi.org/10.1137/S0036144504446096, https://doi.org/10.1137/S0036144504446096
10. Gurobi Optimization, Inc.: Gurobi optimizer reference manual (2012), http://www.gurobi.com
11. Kapoor, P., Balakrishnan, A., Deshmukh, J.V.: Model-based reinforcement learning from signal temporal logic specifications. arXiv preprint arXiv:2011.04950 (2020)
12. Kurtz, V., Lin, H.: Mixed-integer programming for signal temporal logic with fewer binary variables. IEEE Control Systems Letters (2022)
13. Leahy, K., Mann, M., Vasile, C.I.: Rewrite-based decomposition of signal temporal logic specifications. In: Rozier, K.Y., Chaudhuri, S. (eds.) NASA Formal Methods. pp. 224–240. Springer Nature Switzerland, Cham (2023)
14. Leung, K., Aréchiga, N., Pavone, M.: Backpropagation through signal temporal logic specifications: Infusing logical structure into gradient-based methods. The International Journal of Robotics Research p. 02783649221082115 (2023)
15. Lindemann, L., Dimarogonas, D.V.: Control barrier functions for signal temporal logic tasks. IEEE Control Systems Letters **3**, 96–101 (2019), https://api.semanticscholar.org/CorpusID:50767137
16. Maler, O., Nickovic, D.: Monitoring temporal properties of continuous signals. In: FORMATS/FTRTFT (2004), https://api.semanticscholar.org/CorpusID:15642684
17. Menghi, C., Tsigkanos, C., Pelliccione, P., Ghezzi, C., Berger, T.: Specification patterns for robotic missions. IEEE Transactions on Software Engineering **47**(10), 2208–2224 (2021). https://doi.org/10.1109/TSE.2019.2945329
18. Nair, S., Finn, C.: Hierarchical foresight: Self-supervised learning of long-horizon tasks via visual subgoal generation. ArXiv **abs/1909.05829** (2019), https://api.semanticscholar.org/CorpusID:202565422

19. Pnueli, A.: The temporal logic of programs. In: 18th Annual Symposium on Foundations of Computer Science (sfcs 1977). pp. 46–57 (1977). https://doi.org/10.1109/SFCS.1977.32
20. Raman, V., Donzé, A., Maasoumy, M., Murray, R.M., Sangiovanni-Vincentelli, A., Seshia, S.A.: Model predictive control with signal temporal logic specifications. In: 53rd IEEE Conference on Decision and Control. pp. 81–87 (2014). https://doi.org/10.1109/CDC.2014.7039363
21. Sadraddini, S., Belta, C.: Formal synthesis of control strategies for positive monotone systems. IEEE Transactions on Automatic Control **64**(2), 480–495 (2019). https://doi.org/10.1109/TAC.2018.2814631
22. Sun, D., Chen, J., Mitra, S., Fan, C.: Multi-agent motion planning from signal temporal logic specifications. IEEE Robotics and Automation Letters **PP**, 1–1 (2022), https://api.semanticscholar.org/CorpusID:245986629
23. Tedrake, R., the Drake Development Team: Drake: Model-based design and verification for robotics (2019), https://drake.mit.edu
24. Yu, X., Wang, C., Yuan, D., Li, S., Yin, X.: Model predictive control for signal temporal logic specifications with time interval decomposition (2022)

A Appendix

A.1 Slicing Algorithm and Illustration

Our slicing algorithm is illustrated in Algorithm 3 and its application is demonstrated using the running example. In this example, the first set of constraints to be fulfilled is $\{t_2 = (2, 21, r_1), t_4 = (1, 35, \neg r_3)\}$. In the first step (line 3), we identify the lowest and the highest time bounds out of all the constraints in $currTasks$ (which, for the example, would be 1 and 35). Then, in lines 5 to 11,

for the horizon, we identify which constraints are active at each given time step. After this step, in lines 14 to 29, we first create "slices", which involves generating multiple time intervals out of time steps where similar constraints are active. Then, we create constraints of type $\mathcal{X}^s$ and $\mathcal{X}^\forall$ out of them by combining propositions of constraints of the same type. For the running example, these time slices would be [1,1], [2,21] and [22,35]. Finally, the constraints are converted into the following three atomic tasks:

$$G_{[1,1]}(\neg r_3) \qquad F_{[2,21]}(r_2) \wedge G_{[2,21]}(\neg r_3) \qquad G_{[22,35]}(\neg r_3)$$

Algorithm 3 SLICE

Input: Set of currTasks $\mathcal{X}^c$
Output: Set of atomic tasks constraints $\mathcal{X}^{c'}$
t_{min}, t_{max} := HORIZON($currTasks$)
$active := \langle\rangle$
for t in $[t_{min}, t_{max}]$ **do**
    $active_t := \langle\rangle$
    for $x = (l, h, p)$ in $\mathcal{X}^c$ **do**
        if t in $[l, h]$ **then** $active_t := active_t^\frown \langle x \rangle$ **end if**
    end for
    $active := active^\frown active_t$
end for
$t_{low} := t_{min}$
$\mathcal{X}^{c'} = \{\}$
for t in $[t_{min} + 1, t_{max}]$ **do**
    if $active[t-1]! = active[t]$ **then**
        $\mathcal{X}^{temp} := active[t-1]$
        $\mathcal{X}^{temp}_{\exists} := \{(t_{low}, t-1, \top)\}$
        $\mathcal{X}^{temp}_{\forall} := \{(t_{low}, t-1, \top)\}$
        for $x = (l, h, p)$ in $\mathcal{X}^{temp}$ **do**
            if x in $\mathcal{X}^{\exists}$ **then**
                $\mathcal{X}^{temp}_{\exists}.p := \mathcal{X}^{temp}_{\exists}.p \wedge p$
            else
                $\mathcal{X}^{temp}_{\forall}.p := \mathcal{X}^{temp}_{\forall}.p \wedge p$
            end if
        end for
        $\mathcal{X}^{c'} := \mathcal{X}^{c'} \cup \mathcal{X}^{temp}_{\exists} \cup \mathcal{X}^{temp}_{\forall}$
        $t_{low} := t$
    end if
end for
return $\mathcal{X}^{c'}$