# Plan Generation via Behavior Trees Obtained from Goal-Oriented LTLf Formulas

Aadesh Neupane[1 [0000−0003−0039−8832]], Michael A. Goodrich[1 [0000−0002−2489−5705]], and Eric G Mercer[1 [0000−0002−2264−2958]]

Brigham Young University, Provo UT 84602, USA
adeshnpn@byu.edu, {mike,egm}@cs.byu.edu
https://cs.byu.edu

**Abstract.** Temporal logic can be used to formally specify autonomous agent goals, but synthesizing planners that guarantee goal satisfaction can be computationally prohibitive. This paper shows how to turn goals specified using a subset of *finite trace Linear Temporal Logic* ($LTL_f$) into a *behavior tree* (BT) that guarantees that successful traces satisfy the $LTL_f$ goal. Useful $LTL_f$ formulas for *achievement goals* can be derived using achievement-oriented task mission grammars, leading to missions made up of tasks combined using LTL operators. Constructing BTs from $LTL_f$ formulas leads to a relaxed behavior synthesis problem in which a wide range of planners can implement the *action* nodes in the BT. Importantly, any successful trace induced by the planners satisfies the corresponding $LTL_f$ formula. The usefulness of the approach is demonstrated in two ways: a) exploring the alignment between two planners and $LTL_f$ goals, and b) solving a sequential *key-door* problem for a *Fetch* robot.

**Keywords:** Behavior Tree · Finite Linear Temporal Logic

## 1 Introduction

Specifying robot goals, creating plans, and verifying plans have received considerable attention in the research literature. Linear Temporal Logic (LTL) has been used in this context to specify system properties such as safety, liveness (something will keep happening), and goal-satisfaction [29]. However, the automata-based planning algorithms that accompany these systems often increase exponentially with an increase in the specification's length [24]. Moreover, the tight coupling between LTL verification and planning problems with automata-based controllers leads to expensive re-computation even for small specification changes.

One way to address these shortcomings is to decompose a complex goal specification into smaller modular specifications. Decomposing a complex goal specification using a Behavior Trees (BTs) is often useful because BTs are modular, maintainable, and reusable [11]. Prior work demonstrated polynomial time correct-by-construction BTs from an LTL formula [9], but a particularly restrictive specification format must be followed. The planning is also tightly coupled

with the BT decomposition algorithm, making the integration of off-the-shelf planners impossible.

This paper presents a mission-oriented grammar that generates $LTL_f$ formulas appropriate for sequential *achievement goals* [34]. An achievement goal is defined as one that succeeds if a certain postcondition is satisfied without violating constraints. A sound mapping between the $LTL_f$ formula and a corresponding BT is then presented, where the BT is structured to use Postcondition-Precondition-Action (PPA) structures. The PPA-style structure allows action nodes to be implemented with off-the-shelf planners. The paper demonstrates the usefulness of the resulting BTs using two case studies. First, the compatibility of a planner's objective with an $LTL_f$ goal is explored (a) by designing plans using Markov Decision Process (MDP)-based planners, (b) by constructing a simple sampling algorithm, and (c) then comparing how well the outcome of the planners match the $LTL_f$ formula. Second, the $LTL_f$ -to-BT algorithm is used to construct *Fetch* robot behaviors that successfully perform *key-door* tasks when perturbations occur.

## 2 Related Work

LTL [23] is expressive enough to describe complex requirements (safety, liveness, goal-satisfaction) for discrete-time state transition systems. Bacchus et al. [3] were the first to show that linear temporal logic not only can be used to specify system properties but also can be used to specify goals for formal logic systems.

Verifying and synthesizing plans to satisfy complex goal specifications can be computationally prohibitive [24, 17], but plans for restricted goal specifications can be found in polynomial time [22]. Thus, there appears to be a trade-off between the expressivity of the specification and the efficiency of the planners. One way to address this expressivity-computability trade-off is by decomposing expressive specifications into small modular pieces. Decomposing complex plans into smaller pieces is not new. Most earlier methods applied decomposition not to the actual specifications but to supporting algorithms ranging from parsing to planning, including reinforcement learning-based approaches [4, 25, 19, 31, 21]. Colledanchise et al. [9] were the first to demonstrate direct decomposition of LTL specifications. Vazquez et al. [36] went in a reverse direction by demonstrating a method to construct temporal LTL specifications from a grammar containing atomic specifications.

Frequently, decomposing a LTL specification and creating planners are done concurrently. Generally, the planning process involves: a) converting LTL goal specification to an automaton, b) creating an automaton modeling the environment, c) constructing a product automaton, d) playing the Rabin game using game theory concepts, and e) discretizing the plans [2, 14, 5, 16, 13, 33, 28, 32]. Probabilistic Computation Tree Logic is an alternative sometimes used when the planning process is computationally expensive and when uncertainties are present [18, 12]. When the guarantees provided by automaton-based planning are not required, sampling-based motion planning algorithms can be used [35,

| **Node** | Figure | Symbol | **Success** | **Failure** |
|---|---|---|---|---|
| **Sequence** | $\rightarrow$ | $\sigma$ | all children succeed | one child fails |
| **Parallel** | $\rightrightarrows$ | $\pi$ | all children succeed | one child fails |
| **Selector** | ? | $\lambda$ | one child succeeds | all children fail |
| **Decorator** | $\Diamond$ | $\delta$ | User defined | |
| **Action** | $\square$ | $\alpha$ | task complete | task failed |
| **Condition** | $\bigcirc$ | $\kappa$ | true | false |

**Table 1.** BT node type notation.

1]. Interestingly, the acceptance of the trace by the automaton can be used as a reward function in some MDP problems [27].

BT representations are (i) equivalent to Control Hybrid Dynamical Systems and Hierarchical Finite State Machines [20] and (ii) generalizations to the Teleo-Reactive paradigm and And-Or-Trees [10]. BT modularity can be combined with the verification properties of LTL. Biggar et al. [6, 7] developed a mathematical framework in LTL is used to verify BT correctness without compromising valuable BT traits: modularity, flexibility, and reusability.

## 3 Behavior Tree and $LTL_f$ Semantics

Assume the world is represented as a finite set of atomic propositions denoted by AP. Assume further that the set of atomic propositions is partitioned into those internal to the behavior tree, $\mathrm{AP}_b$, and those external to the behavior tree, $\mathrm{AP}_w$ where the subscripts $b$ and $w$ represents "behavior tree" and "world", respectively. Thus, $\mathrm{AP} = \mathrm{AP}_b \cup \mathrm{AP}_w$.

Let $\{null, a_1, a_2, \ldots, a_m\} \in A$ denote the set of valid actions, where *null* indicates inaction by the agent. The world is "open", which means that atomic propositions can change even when the agent does not act. Inaction is represented by *null*. Assume a discrete time world where changes occur at fixed intervals. Further, assume uncertainty in the world's starting state, and assume that the world is deterministic given its starting state. Differences in a state trajectory induced by a fixed action sequence are modeled as random selection of starting state.

### 3.1 Behavior Tree Semantics

A *Behavior Tree* (BT) is a directed rooted tree where the internal nodes are called *control nodes*, and the leaf nodes are called *execution nodes*. A node can be in only one of three states: *running* (processing is ongoing), *success* (the node has achieved its objective), or *failure* (anything else). This subsection presents a formal syntax and semantics for behavior tree behavior.

We define the standard two types of execution nodes, (*Action* and *Condition*), but we only define three of the four standard types of control nodes: *Sequence*, *Selector*, and *Decorator* [11]. The *Parallel* node type is not implemented, which

has some consequences for how quickly missions can execute (discussed in the next section). The syntax of the nodes define the structure of the BT. We use N to denote a node of arbitrary type and let the syntax make clear the actual type in the semantics. Node syntax uses prefix notation, ($N$ argument_list), where the node operation type is followed by the arguments on which the node operates. For example, ($\sigma$ $N^L$ $N^R$) represents a sequence node ($\sigma$) that operates on left and right child nodes ($N^L$ and $N^R$, respectively).

An action node, ($\alpha$ id) where id is a unique identifier, executes the action or plan that is referred to by id in the world. A condition node, ($\kappa$ $\phi$) where $\phi$ is a quantifier free first order logic formula, evaluates some property of the world. A sequence node, ($\sigma$ $\mathrm{N}^L$ $\mathrm{N}^R$), is a logical conjunction with *short-circuit* semantics. Short-circuit semantics evaluate operands in order stopping when the expression outcome can be logically concluded. For simplicity, we restrict ourselves to at most two operand nodes, a left node and right node, in the sequence and other similar node types. This restriction does not affect expressiveness. A selector node ($\lambda$ $\mathrm{N}_0$ $\mathrm{N}_1$) acts as a disjunctive proposition with short-circuit semantics. A decorator node, ($\delta$ id N), follows a user defined rule identified by "id" that decides its status from its child.

BT semantics are defined recursively over the syntax of the tree with a *tick* function. A *tick* is recursively sent from a controller, which is external to the behavior tree, to the root tree node, which passes the *tick* to its children using a depth-first-search (DFS) tree traversal from left to right.

In the BT semantics, system state and dynamics are abstracted to be a function $s : AP \mapsto \{success, failure, running\}$ mapping atomic propositions to analogues for true, false, and unknown respectively. We omit the details of the ternary propositional logic to not distract from the presentation. Only decorator nodes keep an internal state, so the set $\mathrm{AP}_b$ contains the internal states of all decorator nodes in the behavior tree.

For many roboticists and agent designers, defining state as a function differs from the conventional approach as defining it as a vector of state variables. We use the function-based representation of state because it makes expressing behavior tree semantics considerably easier than using a state vector-based representation. Note that the vector-based representation can be obtained by assigning an order to elements of $\mathrm{AP} = \{p_1, p_2, \ldots\}$ representing state $\vec{s} = [s(p_1), s(p_2), \ldots]$ as vector of atomic proposition values returned by the state function $s$.

Each BT node returns two things: its status ($b$) and the updated state function ($s$), yielding the tuple $(b, s)$. (Returning the updated state function is equivalent to returning a "next state vector".) Only decorator nodes update internal behavior tree state and only action nodes update external world state. Behavior tree semantics are defined recursively, with two base cases: action nodes, Eq. (1), and condition nodes, Eq. (2). For condition nodes, we indicate with $[\![\phi]\!]_s$ the evaluation of the propositional formula $\phi$ in the logic given the state $s$. The notation ($[\![\phi]\!]_s$ $s$) can be read as "the operator $[\![\phi]\!]_s$ gives meaning to state $s$." The $L$ and $R$ superscripts indicate the left and right children, respectively, of the sequence and selector nodes.

$$tick\ s\ (\alpha\ \mathrm{id}) = run\ s\ \mathrm{id} \tag{1}$$

$$tick\ s\ (\kappa\ \phi) = (\llbracket \phi \rrbracket_s\ s) \tag{2}$$

$$tick\ s\ (\sigma\ \mathrm{N}^L\ \mathrm{N}^R) = \begin{cases} (b^L, s^L) & \textbf{if } b^L \in \{fail, running\} \\ & \text{where } (b^L, s^L) = tick\ s\ \mathrm{N}^L \\ tick\ s^L\ \mathrm{N}^R & \textbf{otherwise} \end{cases} \tag{3}$$

$$tick\ s\ (\lambda\ \mathrm{N}^L\ \mathrm{N}^R) = \begin{cases} (b^L, s^L) & \textbf{if } b^L \in \{success, running\} \\ & \text{where } (b^L, s^L) = tick\ s\ \mathrm{N}^L \\ tick\ s^L\ \mathrm{N}^R & \textbf{otherwise} \end{cases} \tag{4}$$

$$tick\ s\ (\delta\ \mathrm{id}\ \mathrm{N}) = compute\ s\ \mathrm{id}\ \mathrm{N} \tag{5}$$

Actions, (1), rely on an external *run* function to execute the indicated plan from the given state. That function results in a new updated state that is propagated downstream (e.g., from leaf nodes to parent nodes) in the semantics. Sequences, (3), and selectors, (4), *tick* the second child depending on the return status of the *tick* on the first child according to their respective semantics. Decorators, (5), rely on an external *compute* to execute the indicated rule that decides whether or not the child sees the *tick* and what the return status should be. Like the *run* function, *compute* may result in a new updated state that is propagated downstream in the semantics.

Let $\mathbb{T}$ denote a specific behavior tree. The language of $\mathbb{T}$ is all possible sequences resulting from some set of valid initial states. To define this language, we assume a controller function that calls *tick* at a fixed frequency, passes *tick* to the root note, and repeats until the tree resolves to a final end state. We identify the tree as a controller function, $(b,\ [s_0, s_1, \ldots]) = \mathbb{T}(s_0)$, where the subscripts associated with the states represent changes in state. The first tick always uses the starting state $s_0$ with subsequent ticks using the resulting state from the previous tick. The tree returns the final state of the tree ($b \in \{success, fail\}$) and the finite sequence of states observed after each call to *run* and *compute* while ticking the tree ($[s_0, s_1, \ldots]$).

We make three assumptions about the relationship between the behavior tree and the external world:

- the *run* and *compute* functions eventually succeed or fail,
- the tick frequency is such that the previous tick on the tree always returns before the next tick, and
- the tick frequency is fast enough that external changes in the world (e.g., changes not caused by *run*) are observed in the tree.

We now define the *traces* and the *language* of a BT.

**Definition 1 (Traces).** *The set of traces generated by* $\mathbb{T}$ *given a set of initial states* $S_0$ *is*

$$\Gamma(S_0, \mathbb{T}) = \big\{[s_0, s_1, \ldots] \mid s_0 \in S_0 \wedge (b, [s_0, s_1, \ldots]) = \mathbb{T}(s_0) \ \wedge \\ (b == \text{success} \ \vee b == \text{failure})\big\}$$

*where the trace only includes external states from* $\text{AP}_w$ *and omits internal states from* $\text{AP}_b$.

Observe that the notation $[s_0, s_1, \ldots]$ represents a finite trace with one or more states. Traces are finite because the tree terminates with success or failure. Also observe that the tree is deterministic and that traces are defined with respect to a set of starting states $S_0$. The set of starting states models nondeterminism as uncertainty over starting states, meaning that a starting state is chosen nondeterministically and the trace that results is deterministic.

Stated simply, the traces make up the set of all external state sequences that can be produced by a BT when the BT resolves to success or fail. We define the *language* of a BT as the subset of trajectories that resolve to success.

**Definition 2 (Language of a BT).** *The language of a BT* $\mathbb{T}$ *given a set of initial states* $S_0$ *is*

$$L(S_0, \mathbb{T}) = \big\{[s_0, s_1, \ldots] \mid s_0 \in S_0 \wedge (b, [s_0, s_1, \ldots]) = \mathbb{T}(s_0) \ \wedge \\ b == \text{success}\big\} \tag{6}$$

The remainder of the presentation omits $S_0$ from the notation preferring $L(\mathbb{T})$ with the implicit assumption that it is defined over some set of initial world and decorator node states.

### 3.2 $LTL_f$ Semantics

$LTL_f$ formulas are constructed from a finite set of propositional variables drawn from the set of atomic propositions, AP combined using *logical operators* and *temporal modal operators*. Conventional logical operators are used: $\neg$, $\vee$ , and $\wedge$. The temporal modal operators are **X** ("neXt"), **U** ("Until"), **F** ("Finally", meaning at some time in the future), and **G** ("Globally"). Unary operators take precedence over binary, the **U**ntil operator takes precedence over the $\wedge$ and $\vee$ operators, and all operators are left associative.

A state is defined as a subset of atomic propositions that are true at time $i$, $s_i \subseteq \text{AP}$. $LTL_f$ formulas operate on a trace, denoted by $\tau = [s_0, s_1, \ldots]$ that consists of a sequence of states. Trace segments are indexed using the following notation: $\tau[i] = \mathbf{s}_i$ and $\tau[i:j] = [\mathbf{s}_i, \ldots, \mathbf{s}_j]$. Let $m$ denote the maximum length of a finite $LTL_f$ formula. Thus, a full trace is $\tau[0:m]$ and the "suffix" of a trace beginning at time $i$ in $0 < i \leq m$ is $\tau[i:m]$.

The semantic interpretation of an $LTL_f$ formula, $\psi$, is given using the *satisfaction relation*, $\models$, which defines when a trace satisfies the formula. A trace suffix $\tau[i:m]$ satisfying an $LTL_f$ formula $\psi$ is denoted using prefix notation by $\models \tau, i\ \psi$ and is inductively defined (using the $\equiv$ to indicate "defined") as follows:

- $\models \tau, i\ A \equiv A \in \tau[i]$ and $A \equiv \texttt{true}$
- $\models \tau, i\ (\neg\psi) \equiv (\not\models \tau, i\ \psi)$
- $\models \tau, i\ (\wedge\ \psi_1\ \psi_2) \equiv \models \tau, i\ \psi_1$ and $\models \tau, i\ \psi_2$.
- $\models \tau, i\ (\vee\ \psi_1\ \psi_2) \equiv \models \tau, i\ \psi_1$ or $\models \tau, i\ \psi_2$.
- $\models \tau, i\ (\mathbf{X}\ \psi) \equiv \models \tau, (i+1)\ \psi$.
- $\models \tau, i\ (\psi_1\ \mathbf{U}\ \psi_2) \equiv \models\ \tau, k\ \psi_1$ and $\models\ \tau, j\ \psi_2$ where $\exists j \in \{i, i+1, \ldots, m\}$ such that $\forall k \in \{i, i+1, \ldots, j-1\}$.
- $\models \tau, i\ (\mathbf{G}\ \psi) \equiv \models\ \tau, k\ \psi$ where $\forall k \in [i, m]$.
- $\models \tau, i\ (\mathbf{F}\ \psi) \equiv \models\ \tau, k\ \psi$ where $\exists k \in [i, m]$.

Observe that traces in $LTL_f$ are made up of boolean elements not the ternary elements used in behavior trees. The following section constructs a specific $LTL_f$ formula that is satisfied by traces generated by a formula-specific BT whenever the behavior tree returns success. A key step is mapping between traces generated by a successful BT and a trace evaluated by an $LTL_f$ formula.

## 4 Constructing a Behavior Tree from an $LTL_f$ Task Formula

This paper restricts attention to goal specifications to *task* structures that follow a postcondition, precondition, and action (PPA) structure. PPA structures check the postcondition before trying an action, and have proven useful for agent-based and robotics applications [11]. The resulting subset of $LTL_f$ that uses the PPA structure is called *PPA-LTLf*. This section shows that a BT can be constructed such that the traces generated by the BT satisfy a formula expressed using the *PPA-LTLf* structure.

### 4.1 Task Formula and Task Behavior Tree

**PPA-Style Task Formula** The basic structure of a PPA-structured achievement goal can be represented using the propositional logic formula $PPATask = \vee\{PoC\{\wedge[PrC][Action]\}$ where $PoC$, $PrC$, and $Action$ are postcondition, precondition, and action propositions, respectively. The $Action$ proposition is true iff and only if the action satisfies the post-condition, so the formula could have been rewritten as $PPATask = \vee\{PoC\{\wedge[PrC][PoC]\}$. If execution of the expression is performed from left to right then the $\vee$ operator means that the action will not be executed if the postcondition is already satisfied. If the postcondition is not satisfied, then the $\wedge$ operator ensures that action is only executed when the precondition is satisfied. To help extend the formula to more general conditions, the two operands of the $\vee$ operator are delineated with curly braces and the two operands of the $\wedge$ operator with square braces.

A more general goal includes global and task constraints,

$$\begin{aligned} \psi &= \wedge\Big\{\mathbf{G}\ GC\Big\}\Big\{\ \vee\ \Big[PoC\Big]\ \Big[\wedge\ \Big(PrC\Big)\ \ \Big(\mathbf{U}\ (TC)\ (Action)\Big)\Big]\Big\} \\ &= \wedge\Big\{\mathbf{G}\ GC\Big\}\Big\{\ \vee\ \Big[PoC\Big]\ \Big[\wedge\ \Big(PrC\Big)\ \ \Big(\mathbf{U}\ (TC)\ (\wedge\ PoC\ GC)\Big)\Big]\Big\} \end{aligned} \tag{7}$$

where PoC, PrC, GC, and TC are boolean formulas that encode task postconditions, task preconditions, global constraints, and task constraints propositions, respectively. The two forms of the equation emphasize that an *Action* is satisfied if and only if both the post condition and the global constraint are satisfied. Large curly braces delineate the two operands of the leftmost $\wedge$ operator, which requires that the global constraint is always satisfied *and* the postcondition is eventually satisfied. Large square braces delineate the two operators of the $\vee$ operator where the first operand represents the postcondition and the second operand represents that precondition/action pair. The simple postcondition has been replaced with a check that the global constraint and postcondition are simultaneously satisfied. The precondition is delineated by the large parentheses. The action has been replaced by an until operator, which says that the task constraint must hold until the action has satisfied the postcondition and global constraint. This formula can be simplified since there are redundant conditions (e.g., the requirement that the global constraint be satisfied globally and the requirement that the action satisfy the global constraint), but this form allows a direct mapping onto a behavior tree.

**BT for a PPA-Style Task** Figure 1 shows the *task behavior tree*, denoted by $\mathbb{T}_\psi$, constructed from the task formula $\psi$ in Equation (7). Let $L(\psi)$ denote the set of all traces that satisfy $\psi$. We will show that the behavior tree is a sound implementation of the task formula, which means that $L(\mathbb{T}_\psi) \subseteq L(\psi)$.

We now describe how the components of the task formula are implemented in the BT. We do this by describing each node as we encounter it while performing a depth first search tree traversal of the tree. We provide a description of how signals flow between components of the tree, relating the signals to the BT semantics in Eqs. (1)–(5). The edges in the tree are labeled with the signals that correspond to a successful execution of the BT. The downward curved arrows represent states passed to children and the upward curved arrows represent the return statuses and states passed to parents. The subscripts indicate changes in state. The return status of *success* and *failure* are denoted in shorthand form by $b = c$ or $b = f$, respectively.

**Controller.** The downward arrow at the top of the tree represents the controller that generates ticks at a fixed frequency. The $s_0$ label next to the arrow is the initial state.

**Sequence node $\boldsymbol{\sigma_{\mathbf{PPATask}}}$.** This sequence node implements the leftmost *and* operator $\wedge\{\mathbf{G}\ GC\}\{\cdot\}$ of Eq. (7). Its left child is the condition node $\kappa_{GC}$ that evaluates the global condition ($[\![GC]\!]_s\ s$) from Eq. (2). This condition node receives state $s_0$ from its parent and returns success $c$, indicating that the global constraint was satisfied. The condition node does not modify state, so it returns $s_0$.

**Selector node $\boldsymbol{\lambda_{\mathbf{PocBlk}}}$.** This selector node implements the *or* operator $\vee\ [PoC]\ [\cdot]$ of Eq. (7). This selector node receives the state $s_0$ from its parent and returns a success status ($c$) to its parent along with the state modified by its descendants. Its left child is the condition node $\kappa_{PoC}$ that evaluates the

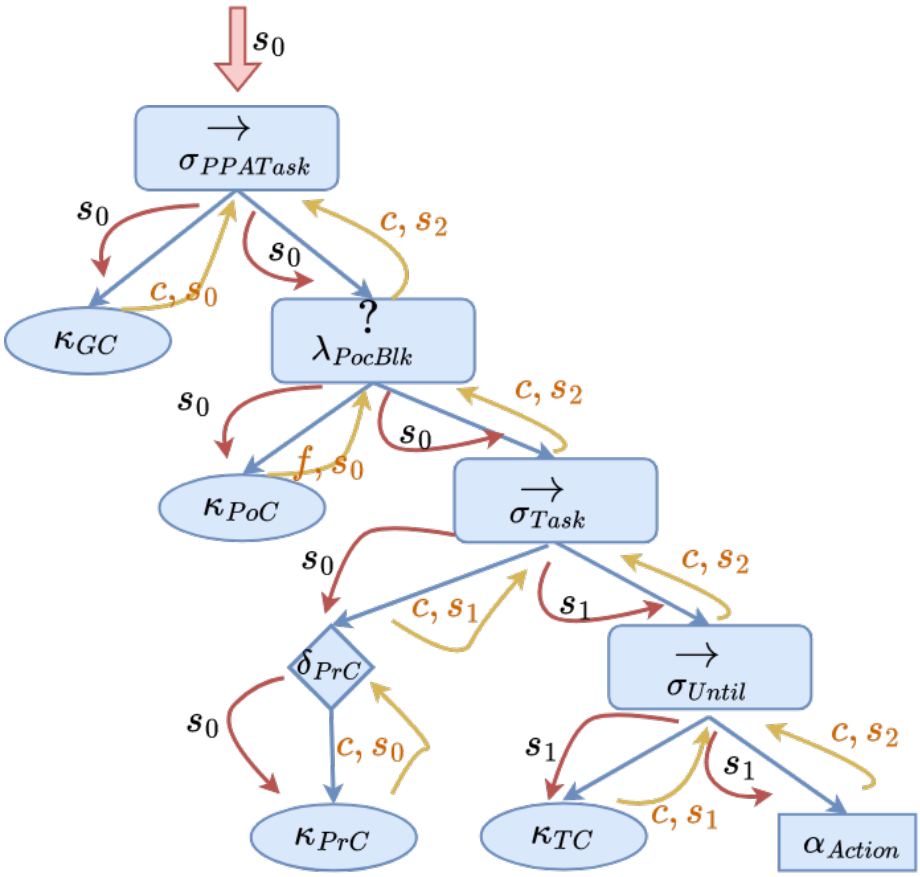


**Fig. 1.** Task BT $\mathbb{T}_\psi$ constructed for PPATask formula $\psi$.

postcondition ($[\![PoC]\!]_s\ s$) from Eq. (2). The signal labels indicate that the child returns an unmodified state $s_0$ and a failure status $f$. The failure status was chosen to illustrate a task that is successful because action is taken in the world.

**Sequence node $\sigma_{\mathbf{Task}}$.** This sequence node implements the *and* portion $\wedge\ (PrC)\ (\cdot)$ of Eq. (7). It's left child checks the precondition and its right child implements the until operator. We describe each child separately.

**Decorator node $\delta_{\mathbf{PrC}}$.** The decorator node and its child check the precondition. The precondition needs to be satisfied only during the tick in which the action begins to execute. Thus, the decorator node always returns *success* if the precondition is satisfied, blocking the need to recheck the precondition. Its child is the condition node $\kappa_{PrC}$ that evaluates the precondition ($[\![PrC]\!]_s\ s$) from Eq. (2). The precondition node receives the state $s_0$, and it returns success $c$ (indicating that the precondition was satisfied) and an unmodified state $s_0$. The

decorator node returns success $c$. The decorator node holds an internal state of the BT, so it changes state from $s_0$ to $s_1$, which indicates that the decorator node perpetually returns success.

**Sequence node $\boldsymbol{\sigma_{\textbf{Until}}}$.** This sequence node implements the until operator **U** $(TC)$ $(Action)$ in Eq. (7). Its left child is the condition node $\kappa_{TC}$ that evaluates the task constraint $([\![TC]\!]_s\ s)$ from Eq. (2). The left child receives the state $s_1$ and returns success $c$ and the unmodified state $s_1$ if the postcondition is satisfied. Its right child is an action node, which we now describe.

**Action node $\boldsymbol{\alpha_{\textbf{Action}}}$.** The action node is constructed so that it returns success only when both the global constraint and the postcondition are satisfied. Formally,

$$\alpha_{Action}[t] = \begin{cases} \textit{success} & \text{if } (PoC \wedge GC = \texttt{true}) \\ \textit{running} & \text{if } (\neg PoC \wedge GC = \texttt{true}) \\ \textit{failure} & \text{otherwise} \end{cases}$$

Note that while the global condition is satisfied then the action node continues to return a *running* status. Observe that the action node returns a modified state $s_2$, which encodes any changes to the external state of the world. This means that each call to the action node returns an updated world state, which assumes that the action node observes the consequences of what it does before returning. Thus, if the action node returns success we are ensured that the global constraint and postcondition are both satisfied, implementing the $\wedge$ *PoC GC* portion of Eq. (7).

### 4.2 BT Success Implies Task Formula Satisfied

Let $\psi$ denote a PPATask formula constructed from Eq. (7), and let $\mathbb{T}_\psi$ denote the behavior tree constructed from the task formula $\psi$ in the pattern of Figure 1. We can define the language of a task formula analogously to the language of a BT.

**Definition 3 (Language of a Task Formula).** *The language of an $LTL_f$ task formula constructed from Eq. (7) is defined as*

$$L(\psi) = \{\tau : \tau \textit{ is a trace that satisfies } \psi\}$$

The following theorem states that the task behavior tree is a *safe substitution* of the $LTL_f$ formula. This means that the formula is an over approximation of the BT so any property we conclude in the formula we can conclude about the BT. Before stating and proving the theorem, recall from Definition 1 that the internal states of the BT are excluded from the definition of BT traces.

**Theorem 1.** *Given a task formula, $\psi$, and the behavior tree constructed from the task formula, $\mathbb{T}_\psi$, $L(\mathbb{T}_\psi) \subseteq L(\psi)$*

*Proof.* The proof has two cases.

*Case 1: There is only one element of the trace.* Each call to the action node $\alpha_{\text{Action}}$ appends an element to the trace, so a successful trace with a single

element produced by $\mathbb{T}(\psi)$ never reaches $\alpha_{\text{Action}}$. Since Eq. (1) says that a successful trace requires one of the children of $\lambda_{PocBlk}$ to return success, this case requires the left child of $\lambda_{PocBlk}$ to return success, which can only occur when the postcondition is satisfied. And since Eq. (3) says that $\sigma_{PPATask}$ requires both of its children to return success, a successful trace with a single element must satisfy both $PoC$ and $GC$. Applying to the task formula in Eq. (7), $\psi$ is satisfied because both $GC$ and $PoC$ are satisfied.

*Case 2: There is more than one element of the trace.* Let the elements of the trace be denoted by $\tau = [\tau_0, \tau_1, \ldots, \tau_k]$, where the trace succeeds on the $k^{\text{th}}$ state change. The first trace element $\tau_0$ cannot satisfy both $GC$ and $PoC$ because otherwise Case 1 would have applied. However, the precondition must have been satisfied by the initial element of the trace because otherwise the decorator node would have returned failure and the action node would never have been called. Thus the $PrC$ component of Eq. (7) is satisfied. The task constraint must have been satisfied for all ticks $0, 1, \ldots, k-1$ since otherwise the left child of $\sigma_{Until}$ would have returned failure. Similarly, the global constraint must have been satisfied for all ticks $0, 1, \ldots, k-1$ since otherwise Eq. (2) says that $\kappa_{GC}$ would have returned failure, causing $\sigma_{PPATask}$ to return failure. Thus, both $TC$ and $GC$ were satisfied until tick $k-1$. The action node appends $\tau_k$ to the trace when it returns success, and this can only occur when the action node observes that both the postcondition and global constraint were satisfied. This means that $\tau_k$ satisfies $PoC$ and $GC$. Consequently, the $\mathbf{U}(TC)$ $(\wedge\ PoC\ GC)$ portion of Eq. (7) is satisfied, which means that $\psi$ is satisfied.

In both cases, $\tau \in L(\mathbb{T}_\psi)$ implies $\tau \in L(\psi)$. □

### 4.3 Empirical Verification

It builds confidence to empirically verify that any successful BT execution produces a trace that satisfies the task formula. We simplify the problem by assuming that the constraints and conditions, $\{PoC, PrC, GC, TC\}$ are single propositions, yielding $2^4 = 16$ possible relevant possibilities. A simulation environment was created that caused random transitions between the proposition variables. Each simulation randomly assigned a starting state. Traces of length five were evaluated, yielding a set $16^5$ possible unique traces. A BT was created using the PyTrees [30]. The environment has a *step* method that implements the *tick*.

The task formula was implemented using the python *flloat* library [15]. We ran $2^{20}$ independent trials with random traces, and measured both when the BT returned success or failure and when the trace was satisfied or not.

The confusion matrix from the experiments is shown below. Note that for all 26.81% successful trace generate by the BT, all of them were valid traces. The status of BT and LTLf parse are in complete agreement for $2^{20}$ random traces.

## 5 BT for $LTL_f$ Task-based Missions

This section defines a mission as a temporal sequence of PPA-style tasks constructed from a subset of $LTL_f$ . The set of possible missions is defined using a

| | | **BT Status** | |
|---|---|---|---|
| | | Success | Failure |
| $\boldsymbol{LTL_f}$ **Satisfied** | True | 26.8% | 0 |
| | False | 0 | 73.2% |

**Table 2.** Confusion matrix of PPATask BT experiments.

context free grammar. A behavior tree is then constructed from the grammar. Since the PPATask BT is a safe substitution for the $LTL_f$ formula, a mission can be written in $LTL_f$ and then implemented directly with the PPATasks tree without risking violating the $LTL_f$ formula. We use this safe substitution property to prove by induction that every successful mission generated by the behavior tree satisfies the $LTL_f$ formula describing the mission. The result of this process is the construction of a behavior tree that is a *safe substitution* for the corresponding mission $LTL_f$ formula.

### 5.1 Constructing a Mission Behavior

The temporal sequence is specified using $LTL_f$ operators, but not all temporal sequences are permitted. Combining PPA-style tasks using the LTL operators is constrained by both (a) the nature of tasks and (b) the task BT implementation of a PPA-style task. This subsection presents the mission grammar, describes what can and cannot be done using the mission grammar, and constructs a behavior tree for a mission $LTL_f$ formula.

**Mission Grammar** The mission grammar is expressed in Productions (M1)–(M6)) using prefix operator notation. Productions (M1) and (M3) enforce precedence, which from highest to lowest is **F**inally, **U**ntil, and $\vee$. Productions (M2) and (M4) enforce the left associativity required of the binary LTL operators. Production (M6) means that a Mission is composed of one or more PPA-style tasks. The parentheses in Productions (M5)–(M6) ensure that tasks in a multi-task mission are distinct.

$$\langle \text{Mission} \rangle ::= \langle \text{L1} \rangle \tag{M1}$$
$$::= \vee \langle \text{Mission} \rangle \langle \text{L1} \rangle \tag{M2}$$
$$\langle \text{L1} \rangle ::= \langle \text{L2} \rangle \tag{M3}$$
$$::= \mathbf{U} \langle \text{L1} \rangle \langle \text{L2} \rangle \tag{M4}$$
$$\langle \text{L2} \rangle ::= \mathbf{F} \left( \langle \text{Mission} \rangle \right) \tag{M5}$$
$$::= \left( \langle \text{PPATask} \rangle \right) \tag{M6}$$

**Mission Assumptions and Restrictions** First, the mission grammar does not allow precise task timing. This can be seen by the absence of both the $\wedge$ and Ne**X**t operators from $LTL_f$ . Semantically, $\wedge$ means simultaneous completion time and $Ne\mathbf{X}t$ means completion on the next state change. If precise timing is required between two tasks, these tasks must be combined in a single task action node. Instead of precise timing, a mission implements task sequencing using **F**inally and **U**ntil.

Second, missions are not optimized for speed. The short circuit semantics for a $\vee$ operator mean that the first operand is evaluated before the second. Parallel computations could potentially be much faster when this is implemented on a behavior tree, but this would require that the two tasks avoid side affects like competing for the same resource, occupying the same physical location, or inducing a race condition. Future work should specify precise conditions for what it means for two tasks to be independent.

**Constructing the Mission BT** Let $\Psi$ (capital $\psi$) denote a mission formula derived from the grammar above and expressed in $LTL_f$ form. The mission BT $\mathbb{T}_\Psi$ is constructed from the expression tree in the following manner.

**Leaf nodes of $\mathbb{T}_\Psi$.** Each leaf node is represented by a task tree $\mathbb{T}_\psi$, effectually enforcing the parentheses in Production (M6).

$\vee$ **Operator**. Production (M2) says that some missions can be accomplished in multiple ways, *either* by performing the combination of tasks in the tree descending from the left branch *or* the right branch. The right-hand side of Production (M2) is $\vee\langle\text{Mission}\rangle\langle\text{L1}\rangle$, which is implemented as a subtree rooted at a selector node, $\lambda_{\langle\text{Mission}\rangle}$ as shown in Figure 2.

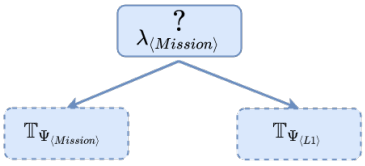


**Fig. 2.** Mission GBT $\mathbb{T}_{\langle\text{Mission}\rangle}$ where $\langle\text{Mission}\rangle ::= \vee\langle\text{Mission}\rangle\langle\text{L1}\rangle$.

**Until Operator**. Production (M4) says that some missions require (a) the task determined by applying productions rooted at $\langle\text{L2}\rangle$ to succeed (b) *until* the task determined by applying productions rooted at $\langle\text{L3}\rangle$ succeeds. Figure 3 shows how the until operator in this production, $\mathbf{U}\langle\text{L2}\rangle\langle\text{L3}\rangle$, is implemented as a subtree rooted at a sequence node type, $\sigma_U$. The sequence node ensures that the left most subtree succeeds before the right subtree is executed.

**Finally Operator**. Production (M6) uses the finally operator. The semantics of the finally operator say that if the child subtree, $T_{\langle\text{Mission}\rangle}$, fails then it should be given another chance to succeed. The finally operator is implemented

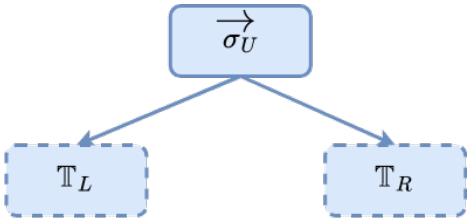


**Fig. 3.** Mission BT $\mathbb{T}_{\langle \text{Mission} \rangle}$ for $\mathbf{U}\ \Psi_{\langle \text{L1} \rangle}\ \Psi_{\langle \text{L2} \rangle}$.

using a decorator node, $\delta_{\mathbf{F}}$, that returns success as soon as its child subtree $\mathbb{T}_{\langle \text{Mission} \rangle}$ returns success. If the child subtree $\mathbb{T}_{\langle \text{Mission} \rangle}$ returns failure, then the decorator node returns running and resets the internal states (i.e., the decorator nodes) of all descendant task trees.

Note that the $\delta_{\mathbf{F}}$ decorator node separates out the subtree descending from the ⟨Mission⟩ nonterminal from the rest of the BT, effectually enforcing the parentheses in Production (M5).

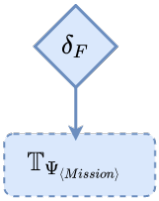


**Fig. 4.** Mission GBT $\mathbb{T}_{\langle \text{Mission} \rangle}$ for $\mathbf{F}\ \Psi$.

In summary, there are two outputs from the finally decorator node: the status returned to its parent and the *reset* command to the task tree if $\mathbb{T}_{\langle \text{Mission} \rangle}[t] = \mathit{failure}$. Because the reset command can cause infinite retries, we include a time-out condition if execution time meets a threshold $T_\theta$. The decorator behavior is defined as

$$\delta_{\mathbf{F}}[t] = \begin{cases} \mathit{success} & \text{if } \mathbb{T}_{\langle \text{Mission} \rangle}[t] = \mathit{success} \\ \mathit{failure} & \text{if } (t \geq T_\theta) \\ \mathit{running} & \text{otherwise} \end{cases}$$

**Mission BT Semantics** Mission semantics are expressed in terms of PPATask success or failure. We omit a precise discussion of mission semantics in the interest of space, but we briefly describe the semantics of a mission tree here. Each mission BT has a controller that issues *tick* to the root node of the tree. This *tick* is passed down the tree in a left-to-right depth first search traversal. The

leaf nodes of a mission tree are composed of PPA Tasks, which are themselves BTs. Like with the task trees, the BT implementations of the binary operators used in a mission behavior tree, $\vee$ from M2 and the **U** from M4, do not evaluate their right child until their left child resolves to success or failure. Consequently, leaf nodes are executed from left to right in the tree and, except for the leftmost leaf node, no leaf node is executed until all leaf nodes to its left resolve to success or failure.

### 5.2 BT Success Implies Formula Satisfied

The *mission trace*, denoted $\tau$, is initialized as the trace produced by the leftmost PPATask leaf node. When the next leaf node resolves to success or failure, the trace from that PPATask BT is appended to the mission trace, and so on.

Define the languages for a mission $LTL_f$ formula $\Psi$ and a mission BT $\mathbb{T}_\Psi$, respectively as follows:

- $L(\Psi) = \{\tau : \tau$ is a mission trace that satisfies $\Psi\}$
- $L(\mathbb{T}_\Psi) = \{\tau$: $\tau$ is a mission trace generated when $\mathbb{T}_\Psi = success\}$

**Theorem 2.** $L(\mathbb{T}_\Psi) \subseteq L(\Psi)$

*Proof:*

The proof is by induction, with multiple induction steps, one for each $LTL_f$ operator appearing in the mission grammar.

**Leaf Nodes.**

The base case is for $\Psi = \psi$, a mission consisting of a single PPA-style task. The base case is proven in Theorem 1.

**Subtrees Rooted in the Finally Operator.** Consider a subtree rooted at $\mathbf{F}(\langle\text{Mission}\rangle)$, and let $\mathbb{T}_{\langle\text{Mission}\rangle}$ denote the subtree that descends from the $\delta_F$ decorator node. The induction hypothesis is that $\mathbb{T}_{\langle\text{Mission}\rangle}$ returns success implies the mission formula $\Psi_{\langle\text{Mission}\rangle}$ is satisfied. The finally operator can only return success if $\mathbb{T}_{\langle\text{Mission}\rangle}$ returns success, even if prior runs of $\mathbb{T}_{\langle\text{Mission}\rangle}$ fail. This means that the trace produced by $\mathbb{T}_{\langle\text{Mission}\rangle}$ eventually satisfies $\Psi_{\langle\text{Mission}\rangle}$, which means that $\mathbf{F}\Psi_{\langle\text{Mission}\rangle}$ is satisfied.

**Induction Step for the Until Operator.** The second induction step is for production (M4), which corresponds to the formula $\mathbf{U}\ \Psi_L\ \Psi_R$, where $L$ and $R$ indicate the formulas encoded in the left and right children of the Until sequence node. The induction hypothesis has two parts: $\mathbb{T}_{\Psi_L}$ = success implies that $\Psi_L$ is satisfied, and $\mathbb{T}_{\Psi_R}$ = success implies that $\Psi_R$ is satisfied. The subtree implementing the until operator, $\mathbb{T}_{\mathbf{U}\ \Psi_L\ \Psi_R}$ is rooted at a sequence node with left and right children, $\mathbb{T}_{\Psi_L}$ and $\mathbb{R}_{\Psi_R}$, respectively. The sequence node returns success only when $\mathbb{T}_{\Psi_L}$ = success up to the time when $\mathbb{T}_{\Psi_R}$ returns success. The fact that $\mathbb{T}_{\Psi_L}$ = success continues to return success while $\mathbb{T}_{\Psi_R}$ executes means that the $LTL_f$ formula corresponding to the $\langle$L1$\rangle$ portion of the production $\mathbf{U}\langle$L1$\rangle\langle$L2$\rangle$ is satisfied for all atomic states up to the time that the $\mathbb{T}_{\Psi_R}$ returns success. When $\mathbb{T}_{\Psi_R}$ returns success, the $LTL_f$ formula corresponding to the $\langle$L2$\rangle$ portion of the production is satisfied. Thus, the until operator is satisfied.

**Induction Step for the Or Operator.** The proof for the $\vee$ operator is omitted since it follows a similar pattern as the proof for the **U** operator, evaluating the trace generated by the left and right children sequentially. □

## 6 Planner-Goal Alignment Example

Given the BT produced from a given mission formula, it is necessary to choose a plan or policy for each action node. Traditional LTL plan synthesis using advanced automata (Rabin and Buchi) requires a suitable environment model and is of high computation complexity. When the goal specifications are valid but sufficient information about the environment is unknown, traditional automata-based solutions are infeasible. However, the BT does not specify what type of planner is required. Instead, off-the-shelf planners can be used to design the plans or policies used in the BT's action nodes. This is demonstrated using two planners: policy iteration, which uses reward functions to represent goals that may or may not align with the the $LTL_f$ formula, and a state-action table that uses the BT return status to update the probability of actions in successful or unsuccessful action sequences.

### 6.1 Problem Formulation

The *Mouse and Cheese* problem is a classic grid world problem [26]. This paper uses a 4x4 grid giving sixteen world locations indexed by $s_{j,k}$. There is one atomic proposition $A_{j,k}$ for each state, where $A_{j,k}$ =*true* indicates that the mouse occupies location $s_{j,k}$. The mouse may not occupy two locations simultaneously, $A_{i,j} == true \rightarrow A_{k,\ell} == false$. Location $s_{4,4}$ contains the cheese, and the mouse automatically picks up the cheese when it occupies that cell. An atomic proposition *Cheese* =*true* indicates that the mouse has the cheese. Location $s_{4,2}$ is dangerous, denoted by atomic proposition $Fire$, and the mouse should avoid this state. Atomic proposition $Home$ indicates whether the agent is at home location $s_{3,1}$. The state vector at time $t$ is the vector of truth values for each atomic proposition $\mathbf{s}_t = [s_{1,1}, s_{1,2}, \ldots, s_{4,4}, Cheese, Fire, Home]$.

Paraphrasing from [26], the agent has four actions: move up, left, right, or down. if the agent bumps into a wall, it stays in the same square. The agent's actions are unreliable, i.e., the ''intended" action occurs with some probability $p_{\text{in}}$ but with some lower probability, agents move at the right angles to the intended direction, $1 - p_{\text{in}}$. The problem gets harder for the agent as $p_{\text{in}}$ decreases.

The mouse and cheese problem is a sequential planning problem, requiring the agent to find the cheese and then return it to the home location. This is an achievement goal that requires two separate plans (or policies): (a) find the cheese avoiding fire and (b) return home avoiding fire after finding the cheese. There are many planners that can solve this problem (e.g., MAX-Q and other hierarchical learners), and the point of this section is not to argue for the best way to solve the problem. Rather, the point is to explore how well different types of planners, one for each task, align with the overall mission goal.

The goal of the mouse is to retrieve the cheese while avoiding the fire location and getting back at the home location. An $LTL_f$ formula from the mission grammar for the sequential find-the-cheese-and-return home (C2H) and the corresponding PPA task formulas from Eq. (7) are

$$\begin{aligned}
\Psi^{\text{C2H}} &= \mathbf{U}\ \mathbf{F}\psi^{\text{Cheese}}\ \mathbf{F}\psi^{\text{Home}} \\
\psi^{\text{Cheese}} &= \vee(\wedge\neg \mathit{Fire}\ \mathit{Cheese})(\wedge(\wedge\neg \mathit{Fire}\ \mathit{True}) \\
&\quad (\mathbf{U}\ \mathit{True}\ \wedge\ \mathit{Action}_{\text{Cheese}})) \\
\psi^{\text{Home}} &= \vee(\wedge\neg \mathit{Fire}\ \mathit{Home})(\wedge(\wedge\neg \mathit{Fire}\ \mathit{Cheese}) \\
&\quad (\mathbf{U}\ \mathit{True}\ \wedge\ \mathit{Action}_{\text{Home}}))
\end{aligned}$$

The BT action nodes $\mathit{Action}_{\text{Cheese}}$ and $\mathit{Action}_{\text{Home}}$ execute the plans that lead mouse to the cheese and return home, respectively.

### 6.2 Action-Node Policies via Policy Iteration

This section uses policy iteration to create a policy for $\mathit{Action}_{\text{Cheese}}$ and again to create a policy for $\mathit{Action}_{\text{Home}}$. The reward structure for the cheese task is $\mathbf{r} = (r_{\text{other}}, r_{\text{cheese}}, r_{\text{fire}})$, where the rewards are for for occupying any grid cell other than home or fire, having the cheese, and occupying the fire cell, respectively. The reward structure for the home task is $\mathbf{r} = (r_{\text{other}}, r_{\text{home}}, r_{\text{fire}})$, where the rewards are for for occupying any grid cell other than home or fire, occupying the home cell, and occupying the fire cell, respectively. Restrict attention to situations where $r_{\text{cheese}} = r_{\text{home}} = r_{\text{good}}$, which allows results to be represented using a triple $\mathbf{r} = (r_{\text{other}}, r_{\text{good}}, r_{\text{fire}})$.

Using policy iteration to create policies from rewards encodes goal in two ways: once in the BTs via the return values of the root node, and once in the reward structures themselves. The reward-goal alignment problem is well-known, and the problem is explicit when the goal is encoded directly in the $LTL_f$ formula but indirectly in the reward structures. By construction of the BT, every successful trace satisfies the goal, but many traces fail or time-out depending on the reward structure.

To illustrate, five hundred twelve independent experiments are conducted for various intended action probabilities $p_{\text{in}}$ and rewards values. The dependent variables are *success probability*, which is defined as the number of simulations where $\Psi^{C2H}$ is satisfied divided by the total number of simulations, and *trace length*, which is the length of the trace. Experiments used the following:

| **Parameter** | **Values** |
|---|---|
| $r_{\text{other}}$ | $\{-1.5, -1.4, -1.3, \ldots, -0.2, -0.1, -0.04\}$ |
| $r_{\text{cheese}} = r_{\text{home}}$ | $\{0.1, 0.5, 1.0, 2.0, 5.0, 10.0\}$ |
| $r_{\text{fire}}$ | $\{-10, -5, -2, -1, -0.5, -0.1\}$ |
| $p_{\text{in}}$ | $\{0.4, 0.45, 0.5, \ldots, 0.9, 0.95\}$ |

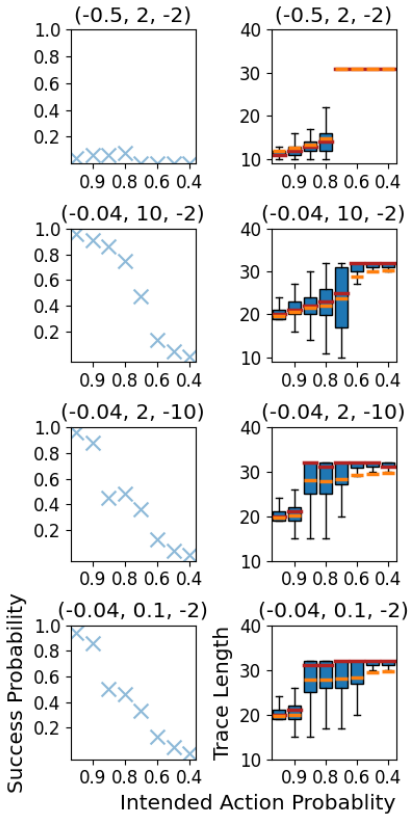


**Fig. 5.** The left column shows the success probability and the right column shows the trace length with various reward structures and intended action probabilities $p_{\text{in}}$. Higher negative rewards in intermediate states leads to lower mission accomplishment rate.

Figure 5 shows the average success probability from a sample of simulations. Each row represents a distinct reward structure, and the x-axis of each subplot represents intended action probabilities $p_{\text{in}}$. Note that when $r_{other}$ has large negative values, the success probability decreases as the $p_{\text{in}}$ decreases, which is consistent with standard results in reinforcement learning. In contrast, the trace length decreases when $r_{other}$ has large negative values because the agent ends up in the fire. The lesson from this experiment is that poor planning in a difficult problem overrides the benefit of the guarantee $\Psi^{C2H}$ is satisfied whenever the BT returns success.

### 6.3 Action-Node Policies with BT Feedback

The property that every successful trace of the BT satisfies the $LTL_f$ formula from which it was derived can be used to learn policies for the action nodes while the BT is running.

**Algorithm** Represent the policy for an action node using a probabilistic state-action mapping, $\pi : S \rightarrow \Delta(A)$, where $S$ is the set of possible states and $\Delta(A)$ is a probability distribution over actions. Thus, the policy is a conditional probability of action given the state, $\pi(s) = p(a|s)$ initialized with uniform action probability.

The learning algorithm lets the BT run a fixed number of episodes ($\xi$), where each episode ends when the BT returns success or failure. Recall that failure can occur if the time limit is reached. During each episode, store the trace as a sequence of time-index state-action pairs $[(s(0), a(0)), \ldots, (s(m), a(m))]$ where $m$ is the trace length. At the end of each episode, update the state action table using

$$p(a(t)|a(t)) \leftarrow \pi(a(t)|a(t)) + \mu^{m-t} * b, \tag{8}$$

where $\pi$ is the policy, (s(t), a(t)) is the state-action pair at time $t$ in the trace $\tau$, $\mu = 0.9$ is the discount factor and $b$ is a binary variable which is translated to +1 (-1) when $\mathbb{T}_\psi$ returns success (failure). After the update, each $p(a|s)$ is renormalized so that $\sum_a p(a|s) = 1$. Since every successful trace is guaranteed to satisfy $\Psi^{\text{C2H}}$, setting $b = 1$ makes actions observed in the trace more likely. It is not true that every failed trace does not satisfy $\Psi^{\text{C2H}}$, the setting $b = -1$ biases exploration to those policies that lead to success.

We can use the fact that $\Psi^{C2H}$ is composed of two PPATasks connected by the **U** operator to use feedback from the PPATask subtree to learn different policies for $\psi^{Cheese}$ and $\psi^{Home}$. Since the two tasks are sequential, the $\psi^{Cheese}$ sub-tree needs to be successful for $\psi^{Home}$ to be learned. Let $p(a|s, C)$ represent the phase where the agent is seeking the cheese, and let $p(a|s, H)$ represent the phase where the agent has the cheese and is learning to return home. The BT nodes $Action_{\text{Cheese}}$ and $Action_{\text{Home}}$ use $p(a|s, C)$ and $p(a|s, H)$, respectively. Divide the trace $\tau$ into two phases, $\tau = [\tau_C, \tau_H]$, where the first part of the trace attempts to perform the Cheese task and the second part attempts to perform the Home task.

Two conditions apply. First, if the cheese subtree never returns success, Eq. (8) updates $p(a|s, C)$ over the entire trace $\tau$. Second, if the cheese subtree returns success, the $\tau_C$ is the part of the trace up to when the task is successful and is used to update $p(a|s, C)$. The remaining part of the trace is $\tau_H$, which uses BT successes of failures to update $p(a|s, H)$.

**Experiment Design and Results** All learning experiments use the following parameters, empirically selected to illustrate successful learning: start a location $s_{(3,0)}$, $m = 50$,$\xi = 200$, and $\mu = 0.9$ unless specified otherwise. Two dependent measures are used: *learning success probability* and *inference success probability.*

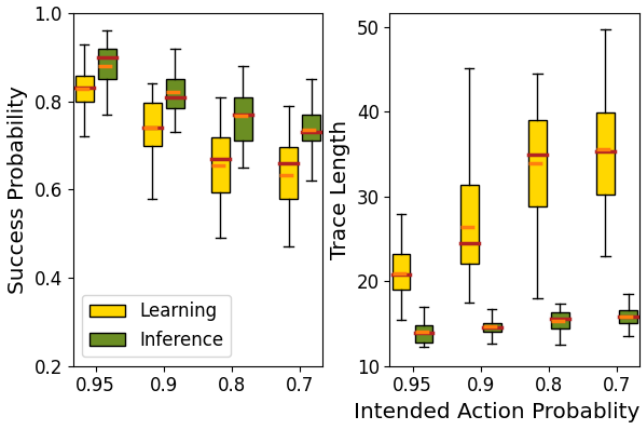


**Fig. 6.** Success probabilities and trace lengths for learning phase and inference phase.

*Learning success probability* is defined as the average success status of mission BT while the policies are being learned divided by the number of episodes. Similarly, the *inference success probability* is the number of successful missions using learned policies, divided by the total number of simulation runs. The agent's starting location is randomized in the inference success evaluations, and 50 simulations were conducted using the learned policies.

For the learning experiments, 50 independent simulations were conducted. Figure 6 compares the performance of policies during learning and inference settings. The golden boxes in the left sub-plot of Figure 6 show that learning $p(a|s)$ directly from the return status of the BT produces good policies for the sequential $\Psi^{C2H}$ problem. Similarly, the golden boxes in the right sub-lot of Figure 6 show that the *trace length* increases with a decrease in intended action probabilities. Figure 6 depicts two distinct properties: a) the success probabilities are higher for the inference phase than the learning phase, and b) the trace length increases much more rapidly during the learning phase than the inference as the intended action probability decreases. These properties are seen because, during the inference phase, just one episode is sufficient to test the policy, whereas the learning phase requires many episodes where the uncertainty in the intended action probabilities accumulates with each episode.

Comparing the results from Figure 5 to Figure 6 shows that using the direct feedback from the BT to create policies for the action nodes has higher success probabilities than using policy iteration when the rewards and goal do not align well.

## 7 Fetch Robot Example

This section demonstrates how a BT for an $LTL_f$-based achievement goal works on the *Fetch* robot. A Fetch robot is a mobile robot with a manipulator arm that

has a) 7 degrees of freedom, b) a modular gripper with easy gripper swapping, c) a torso with adjustable height, and d) an ability to reach items on the floor. The camera is at the head of the robot, so during manipulation tasks, its arm blocks its field of view.

### 7.1 Problem Specification

The sequential problem for the robot is a variant of *key-door* [8] problem. A rectangular box of size 1ftx1ft is the *active region* where its vision system is actively scanning, and the area outside the box is a *passive region*. The task has three blocks with three different shapes: the red block is the key, the black block is the door, and the blue block is the prize. The goal for the robot is to a) locate the key and stack it on the top of the door block, b) move both key and door blocks together to the passive zone, and c) locate the prize and carry it to the passive zone. The robot perceives the world through its vision system and interacts with its arm.

An $LTL_f$ formula for *key-door* (KD) mission is

$$\Psi^{KD} = \mathbf{U}(\ \mathbf{F}\ \psi^{Key})(\ \mathbf{U}\ \mathbf{F}\ \psi^{Door}\ \mathbf{F}\ \psi^{Prize})$$

where three PPA tasks have the structure from in Eq. 7

$$\begin{aligned}
\psi^{Key} &= \vee(\wedge \mathit{NoErr}\ \mathit{KeyStacked})(\wedge(\wedge \mathit{NoErr}\ \mathit{IsKeyDoor}) \\
&\quad (\mathbf{U}\ \mathit{VisibleKeyDoor}\ \wedge\ \mathit{Action}_{\mathit{KeyStacked}})) \\
\psi^{Door} &= \vee(\wedge \mathit{NoErr}\ \mathit{KeyDoorPassive})(\wedge(\wedge \mathit{NoErr}\ \mathit{KeyStacked}) \\
&\quad (\mathbf{U}\ \mathit{KeyStacked}\ \wedge\ \mathit{Action}_{\mathit{KeyDoorPassive}})) \\
\psi^{Prize} &= \vee(\wedge \mathit{NoErr}\ \mathit{PrizePassive})(\wedge(\wedge \mathit{NoErr}\ \mathit{PrizeVisible}) \\
&\quad (\mathbf{U}\ \mathit{KeyDoorPassive}\ \wedge\ \mathit{Action}_{\mathit{PrizePassive}}))
\end{aligned}$$

where the *NoErr* proposition checks if the robot is throwing any system errors, *KeyStacked* proposition checks if the key and door block are stacked together, *IsKeyDoor* checks if the key and door block are on the table, *VisibleKeyDoor* checks if the key and door are visible and not overlapping, *KeyDoorPassive* checks if key and door are in passive area of the table, *PrizePassive* checks if the prize block is in passive area, and *PrizeVisible* checks if the prize block is in the active area. The state vector at time $t$ is the vector of truth values for all the atomic proposition described above, $\mathbf{s}_t = [\mathit{NoErr}, \mathit{KeyStacked}, \ldots, \mathit{PrizeVisible}]$.

The action nodes $\mathit{Action}_{\mathit{KeyStacked}}$, $\mathit{Action}_{\mathit{KeyDoorPassive}}$, and $\mathit{Action}_{\mathit{PrizePassive}}$ execute the plans to stack the key on top of the block, move the stack of key and door to the passive area, and move the prize to passive area, respectively. The sensing and plans were created using widely available Fetch robot libraries [37].

### 7.2 Experiment Design

Two different modes of the experiment were conducted: *baseline* and *PPA-Task-LTLf* . For each mode, 25 independent robot trial was done. One trial corresponds to allowing the robot to perform the mission until it returns failure or

success. In the *baseline* mode, the robot tried executing the mission without using $LTL_f$ to BT decomposition, and the plans were connected using if-else code blocks. The second mode *PPA-Task-LTLf* used the $\Psi^{KD}$ specification and the corresponding BT. For each mode, ten trials were completed without any external disturbances. For the other remaining 15 trials, a human physically interrupted the robot by removing or moving blocks. For each trial, the disturbance was only done once. The interruption was uniformly applied at each stage of the mission. Recall that the **F**inally operator from the mission grammar can send a reset signal to its child sub-tree. In all experiments below, if some tasks fail the robot can retry once, which was chosen subjectively as the robot generally accomplishes the mission after one try.

### 7.3 Results

The top part of Table 3 shows the performance of the robot for the key-door problem without using $LTL_f$ to BT decomposition. The robot failed to complete the mission when interrupted in the baseline condition no matter when the interruption occurred. In the *baseline* mode experiments, the robot does not have the ability to resume from where it last failed because the baseline algorithm does not track details of its failures and attempt to correct constraint violations. Since the BT has a modular postcondition-precondition-action (PPA) structure, by design, it can resume from the previous failure point if allowed to retry.

The bottom part of Table 3 shows the robot's performance when mission BT is used. Despite human interruptions at different stages of the mission, most of the time, the robot could complete the mission as it was allowed to do one retry. The robot failed once on the $\psi^{Door}$ task and twice on the $\psi^{Prize}$ task after interruptions due to constraint violations that could not be reversed. The most important constraint violation was a violation of the *NoErr* proposition, which encodes robot system errors that arise when the planning sub-system is unable to generate plans for the current system states. The descriptive data in Table 3 suggest that the mission success rate is higher when the behavior tree implementation of the mission grammar was used, which was one of the reasons for implementing the mission grammar in a behavior tree.

## 8 Summary and Future Work

This paper presented a mission grammar designed for achievement-oriented goals that require temporal coordination among subtasks. The temporal constraints were formalized in the mission grammar, which produced linear-temporal logic formulas from a subset of LTL operators. The grammar for the tasks was constructed to use postcondition-precondition-action structures, allowing the construction of behavior trees that used these structures. Every successful trace produced by the behavior tree satifies the LTL goal.

A key structure of the behavior trees is that the action nodes can create plans using off-the-shelf planners, which is in contrast to many previous work on con-

| **Baseline Conditions** | | | | |
|---|---|---|---|---|
| Status | Normal | Human Disturbances In | | |
| | | $Task^{\text{Key}}$ | $Task^{\text{Door}}$ | $Task^{\text{Prize}}$ |
| **Success** | 10 | 0 | 0 | 0 |
| **Failure** | 0 | 5 | 5 | 5 |
| **Behavior Tree Conditions** | | | | |
| Status | Normal | Human Disturbances In | | |
| | | $Task^{\text{Key}}$ | $Task^{\text{Door}}$ | $Task^{\text{Prize}}$ |
| **Success** | 10 | 5 | 4 | 3 |
| **Failure** | 0 | 0 | 1 | 2 |

**Table 3.** Experiment results under a) the baseline conditions and b) the BT obtained from Mission and Task grammars.

verting LTL formulas into state machines. The examples presented were straightforward demonstrations for how existing planners could be used to implement the action nodes. Some properties of the resulting planners were demonstrated, specifically the risk of reward-goal misalignment if using MDP-based planners, the ability to use the return status of the behavior tree to train state-action policies, and the ability to retry subtasks to produce more resilient behaviors. The most important piece of future work is to encode sophisticated goals for real robots performing complicated tasks, and then identify state-of-the-art planners that are most compatible with the type of feedback provided by the behavior tree.

**Acknowledgements** This work was supported by the U.S. Office of Naval Research (N00014-18-1-2831). The authors thank Elijah Pettitt, who was an undergraduate research assistant, for programming and running the experiments with the Fetch robot.

## References

1. Ahmadi, M., Sharan, R., Burdick, J.W.: Stochastic finite state control of pomdps with ltl specifications. arXiv preprint arXiv:2001.07679 (2020)
2. Antoniotti, M., Mishra, B.: Discrete event models+ temporal logic= supervisory controller: Automatic synthesis of locomotion controllers. In: Proceedings of 1995 IEEE International Conference on Robotics and Automation. vol. 2, pp. 1441–1446. IEEE (1995)
3. Bacchus, F., Kabanza, F.: Planning for temporally extended goals. Annals of Mathematics and Artificial Intelligence **22**(1-2), 5–27 (1998)
4. Barnat, J., et al.: How to distribute ltl model-checking using decomposition of negative claim automaton. In: SOFSEM. pp. 9–14 (2002)
5. Bertoli, P., Cimatti, A., Pistore, M., Roveri, M., Traverso, P.: Mbp: a model based planner. In: Proc. of the IJCAI'01 Workshop on Planning under Uncertainty and Incomplete Information (2001)

6. Biggar, O., Zamani, M.: A framework for formal verification of behavior trees with linear temporal logic. IEEE Robotics and Automation Letters **5**(2), 2341–2348 (2020)
7. Biggar, O., Zamani, M., Shames, I.: On modularity in reactive control architectures, with an application to formal verification. ACM Transactions on Cyber-Physical Systems (TCPS) **6**(2), 1–36 (2022)
8. Chevalier-Boisvert, M., Willems, L., Pal, S.: Minimalistic gridworld environment for gymnasium (2018), https://github.com/Farama-Foundation/Minigrid
9. Colledanchise, M., Murray, R.M., Ögren, P.: Synthesis of correct-by-construction behavior trees. In: 2017 IEEE/RSJ International Conference on Intelligent Robots and Systems (IROS). pp. 6039–6046. IEEE (2017)
10. Colledanchise, M., Ögren, P.: How behavior trees generalize the teleo-reactive paradigm and and-or-trees. In: 2016 IEEE/RSJ International Conference on Intelligent Robots and Systems (IROS). pp. 424–429. IEEE (2016)
11. Colledanchise, M., Ögren, P.: Behavior trees in robotics and AI: An introduction. CRC Press (2018)
12. Ding, X.C.D., Smith, S.L., Belta, C., Rus, D.: Ltl control in uncertain environments with probabilistic satisfaction guarantees. IFAC Proceedings Volumes **44**(1), 3515–3520 (2011)
13. Fainekos, G.E., Kress-Gazit, H., Pappas, G.J.: Hybrid controllers for path planning: A temporal logic approach. In: Proceedings of the 44th IEEE Conference on Decision and Control. pp. 4885–4890. IEEE (2005)
14. Fainekos, G.E., Kress-Gazit, H., Pappas, G.J.: Temporal logic motion planning for mobile robots. In: Proceedings of the 2005 IEEE International Conference on Robotics and Automation. pp. 2020–2025. IEEE (2005)
15. Favorito, M., Cipollone, R.: flloat (2020), https://whitemech.github.io/flloat/
16. Jensen, R.M., Veloso, M.M.: Obdd-based universal planning for synchronized agents in non-deterministic domains. Journal of Artificial Intelligence Research **13**, 189–226 (2000)
17. Klein, J., Baier, C.: Experiments with deterministic $\omega$-automata for formulas of linear temporal logic. Theoretical Computer Science **363**(2), 182–195 (2006)
18. Lahijanian, M., Wasniewski, J., Andersson, S.B., Belta, C.: Motion planning and control from temporal logic specifications with probabilistic satisfaction guarantees. In: 2010 IEEE International Conference on Robotics and Automation. pp. 3227–3232. IEEE (2010)
19. Maretić, G.P., Dashti, M.T., Basin, D.: Ltl is closed under topological closure. Information Processing Letters **114**(8), 408–413 (2014)
20. Marzinotto, A., Colledanchise, M., Smith, C., Ögren, P.: Towards a unified behavior trees framework for robot control. In: Robotics and Automation (ICRA), 2014 IEEE International Conference on. pp. 5420–5427. IEEE (2014)
21. Parr, R., Russell, S.J.: Reinforcement learning with hierarchies of machines. In: Advances in neural information processing systems. pp. 1043–1049 (1998)
22. Piterman, N., Pnueli, A., Sa'ar, Y.: Synthesis of reactive (1) designs. In: International Workshop on Verification, Model Checking, and Abstract Interpretation. pp. 364–380. Springer (2006)
23. Pnueli, A.: The temporal logic of programs. In: 18th Annual Symposium on Foundations of Computer Science (sfcs 1977). pp. 46–57. IEEE (1977)
24. Pnueli, A., Rosner, R.: On the synthesis of a reactive module. In: Proceedings of the 16th ACM SIGPLAN-SIGACT symposium on Principles of programming languages. pp. 179–190 (1989)

25. Rozier, K.Y., Vardi, M.Y.: A multi-encoding approach for ltl symbolic satisfiability checking. In: International Symposium on Formal Methods. pp. 417–431. Springer (2011)
26. Russell, S.J.: Artificial intelligence a modern approach. Pearson Education, Inc. (2010)
27. Sadigh, D., Kim, E.S., Coogan, S., Sastry, S.S., Seshia, S.A.: A learning based approach to control synthesis of markov decision processes for linear temporal logic specifications. In: 53rd IEEE Conference on Decision and Control. pp. 1091–1096. IEEE (2014)
28. Schillinger, P., Bürger, M., Dimarogonas, D.V.: Decomposition of finite ltl specifications for efficient multi-agent planning. In: Distributed Autonomous Robotic Systems, pp. 253–267. Springer (2018)
29. Sistla, A.P.: Safety, liveness and fairness in temporal logic. Formal Aspects of Computing **6**(5), 495–511 (1994)
30. Stonier, D., Staniasnek, M.: Py-trees (2020), https://py-trees.readthedocs.io/en/devel/index.html
31. Sutton, R.S., Precup, D., Singh, S.: Between mdps and semi-mdps: A framework for temporal abstraction in reinforcement learning. Artificial intelligence **112**(1-2), 181–211 (1999)
32. Tadewos, T.G., Newaz, A.A.R., Karimoddini, A.: Specification-guided behavior tree synthesis and execution for coordination of autonomous systems. Expert Systems with Applications **201**, 117022 (2022)
33. Toro Icarte, R., Klassen, T.Q., Valenzano, R., McIlraith, S.A.: Teaching multiple tasks to an rl agent using ltl. In: Proceedings of the 17th International Conference on Autonomous Agents and MultiAgent Systems. pp. 452–461 (2018)
34. Van Riemsdijk, M.B., Dastani, M., Winikoff, M.: Goals in agent systems: a unifying framework. In: Proceedings of the 7th international joint conference on Autonomous agents and multiagent systems-Volume 2. pp. 713–720. International Foundation for Autonomous Agents and Multiagent Systems (2008)
35. Vasile, C.I., Belta, C.: Sampling-based temporal logic path planning. In: 2013 IEEE/RSJ International Conference on Intelligent Robots and Systems. pp. 4817–4822. IEEE (2013)
36. Vazquez-Chanlatte, M., Jha, S., Tiwari, A., Ho, M.K., Seshia, S.: Learning task specifications from demonstrations. In: Advances in Neural Information Processing Systems. pp. 5367–5377 (2018)
37. Wise, M., Ferguson, M., King, D., Diehr, E., Dymesich, D.: Fetch and freight: Standard platforms for service robot applications. In: Workshop on autonomous mobile service robots. pp. 1–6 (2016)