2019 IEEE 15th International Conference on Control and Automation (ICCA)
Edinburgh, Scotland, July 16-19, 2019

Autonomous task planning and acting for micro aerial vehicles

Menglu Lan[1], Shupeng Lai[2], Tong Heng Lee[2] and Ben M. Chen[2,3]

***Abstract*— Micro aerial vehicles (MAVs) are ideal platforms for various aerial applications such as search and rescue, building inspection and parcel delivery, due to their superior mobility. It is natural and meaningful to investigate on how the MAVs can be employed to autonomously carry out complicated tasks in real applications. In this paper, we present a task planning and acting framework to extend the autonomy level of MAVs further. We use linear temporal logic (LTL) to formally specify the task requirements and solve the reactive synthesis problem through online iterative planning with a determinized system model. Moreover, special attention is given to the interaction between the task and motion levels, where we use behavior tree as an intermediate interface to execute the plan, feedback the execution outcome and facilitate the high-level task re-planning process. The proposed framework has been tested with a high-fidelity simulation using an actual quadrotor dynamics model.**

I. Introduction

It is natural to design and implement an autonomous vehicle in a hierarchical protocol stack, as depicted in Fig. 1. Each block of the stack works at different abstraction levels. In this paper, we are interested in the design of the task planning block and its interactions with the lower-level motion planning block for micro aerial vehicles (MAVs). We assume that the inner-loop blocks for the task planning

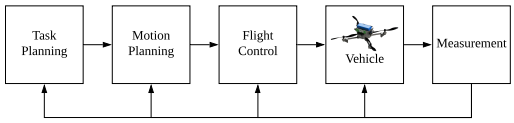


Fig. 1. The block diagram of the interested MAV system.

block (e.g., motion planning, flight control) have already been settled. In other words, we assume that the vehicle has the basic navigation capability to fly safely among waypoints, which is not an unpractical assumption since obstacle avoidance in a dynamic environment is a matured skill that many MAVs platforms have already mastered. [1], [2], [3].

Our work aims to extend the autonomous capabilities of current MAVs by developing a task and acting framework. We are interested in an abstract system that consists of a controllable plant (the MAV) and an uncontrollable environment. The main sources of the uncertainty we considered are task-level events that may happen at any time during the task execution, such as low battery power or receiving a return home command from the user. For the uncertainty of the geometric obstacles, we leave it to the lower-level motion layer to handle. By abstracting the existing lower-level motion skills into a set of primitive behaviors, we can model the entire system as a non-deterministic transition system. Also, we note that many MAV tasks involve temporal constraints, such as inspecting several sites in a user-defined order. Linear temporal logic (LTL) is a convenient tool to formally specify the desired system properties that evolve, such as safety, progress, and response. Our problem is then reduced to synthesize a control policy given a task specification described by an LTL formula. Unlike the common approach that models the reactive synthesis problem into a two-player game and normally focuses on a fragment of the LTL such as GR(1) [4] [5], we propose an alternative approach based on the idea of interleaving the planning and execution and performing iterative online re-planning with a determinized system model. We then can apply automaton-based techniques which are typically for non-reactive synthesis problem [6]. Moreover, motivated by the work in [7], we reformulate the problem into a standard *classical planning* problem, allowing us to use any off-the-shelf classical planners to compute the infinite plan without assuming the LTL is co-safe [8].

A key difference between our work and the typical work in task planning community is that, we are not only interested in plan generation, but also interested in the *acting* side of the generated plan. Specifically, we are interested in the interactions between task level and motion level. Analyzing the interactions between task and motion level is not a trivial task. Symbolic-based task planners work with a highly abstract discrete model which often completely ignores the complex geometric and dynamics details presented in the continuous domain. Such abstraction may lead to an infeasible plan for the lower-level motion algorithm [9]. The situation gets worse in a traditional one-way, top-down task-to-motion framework. A naive way to address the issue is to directly discretize the underlying full state space and apply a graph search algorithm, which is possible in theory. However, in practice, the resulting search space is too large to be solved in a reasonable time. For real-time applications, abstraction and hierarchy remain important techniques to reduce the state space dimension and thus boost the computation efficiency. From this perspective, though the integration of the task and motion planning is a must step towards a higher-level

[1]Graduate School for Integrative Science & Engineering, National University of Singapore. Email: lanmenglu@u.nus.edu
[2]Department of Electrical and Computer Engineering, National University of Singapore.
[3]Department of Mechanical and Automation Engineering, Chinese University of Hong Kong.
The work of Ben M. Chen was supported in part by the Key Program of National Natural Science Foundation of China (Grant No. U1613225).

autonomy, it is important to keep the hierarchical structure instead of fully merging them into one layer.

Hence, similar to the idea in [10], we present a domain-independent interface to provide necessary communications between the task layer and motion layer, as an effort to partially address the integration issue. Different from [10], we use a behavior tree (BT) as the underlying model for the interface. The BT is a tree model that groups a set of modular behaviors, which was originally developed in the game industry to model the complex behaviors of the non-player characters. Recently, there are growing interest to apply the BT for real robotic applications [11] [12] [13] because of its hierarchical, modular structure and natural execution policy. Different from the work in [14] which directly synthesizes a BT from a fragment of LTL through a specific algorithm, our work focuses on how to represent a general high-level task plan into a BT and uses the BT as an independent acting interface, which allows us to use a wider range of task planning algorithms. Specifically, in our work, the BT servers as a refinement acting engine, which gradually refines the high-level task plan into more primitive actions. It also automatically monitors the execution status of the task plan. Any low-level action failure would automatically trigger an update of high-level task abstraction and further trigger a re-planning procedure if necessary.

The rest of the paper is organized as follows: Section II introduces some background information including classical task planning and LTL. Section III presents the design framework of our task-level planning system. Section IV presents the behaviour-tree based interface. We provide a simulation of a common MAV surveillance task using actual quadrotor dynamics model in Section V. Finally, we draw some concluding remarks in Section VI.

II. Preliminaries

A. Transition system

A *weighted transition system* $\mathcal{T}$ is a tuple $(S, Act, \rightarrow_{\mathcal{T}}, S_0, AP, \mathcal{L}, c)$ where S is a set of states, Act is a set of actions, $\rightarrow_{\mathcal{T}} \subseteq S \times Act \times S$ is a transition relation, $S_0 \subseteq S$ is a set of initial states, AP is a set of atomic propositions, $L : S \rightarrow 2^{AP}$ is a labeling function, and c is a weight function which maps the transition into a positive real number. $\mathcal{T}$ is called a *finite* transition system if S, Act and AP are finite. A run $r_{\mathcal{T}}$ of the transition system $\mathcal{T}$ is an infinite sequence of states $s_0 s_1 s_2 \dots$ such that $s_0 \in S_0$ and $s_i \in S$ for all i. A run $r_{\mathcal{T}}$ generates a $word$ $\mathcal{L}(s_0)\mathcal{L}(s_1)\mathcal{L}(s_2)\dots$ consisting of sets of true atomic propositions at each state.

B. Classical planning

The model used for *classical planning* is a tuple $\hat{P} =< S, s_0, S_g, Act, \rho, cost >$, consisting of a finite set of states S, an initial state $s_0 \subseteq S$, a set of goal states $S_g \subseteq S$, a finite set of actions Act with $Act(s)$ denoting the sets of applicable actions in state $s \in S$, and a deterministic state-transition function $\rho : S \times A$ with $\rho(s, a)$ denoting the successor state when applying action $a \in Act(s)$ in the state s, and $cost : S \times A \mapsto [0, \infty)$ is a cost function. If the cost function is not given explicitly, then $cost(s, a) = 1$ whenever $\rho(s, a)$ is defined. A *plan* or a *solution* to a classical planning problem is a sequence of actions $\pi =< a_0, a_1, \dots, a_n >$ that generates a state sequence $s_0, s_1, \dots, s_{n+1}$ such that $a_i \in Act(s_i)$, $s_{i+1} = \rho(a_i, s_i)$ and $s_{n+1} \in S_g$.

The planning model described above can be specified in a compact form through a set of *state variables*. In this paper, we use the common STRIPS-Like representation which is based on the logical propositions. Specifically, let $P =< F, I, A, G >$ defines the planning problem $\hat{P} =< S, s_0, S_g, A, f, cost >$, where F is the set of propositions, I is the set of clauses over F defining the initial situation, A is the set of actions. An action a has precondition given by a set of literals, and a set of conditional effects $C \rightarrow L$, where C is a set of literals and L is a positive literal and G is the set of literals defining the goal situation.

C. LTL syntax and semantics

Let AP denote a set of atomic propositions. Linear temporal logic formula is formed by combining propositions $p \in AP$ with logical and temporal operators, according to the grammar:

$$\phi ::= p \mid \neg\phi \mid \phi \vee \phi \mid \bigcirc \phi \mid \phi \, \mathcal{U} \, \phi \tag{1}$$

$\neg$ and $\vee$ are standard negation and disjunction. Other logical connectives, conjunction ($\wedge$), implication ($\Rightarrow$) and equivalence ($\Leftrightarrow$), are defined as standard. $\bigcirc$ is the temporal operator Next, and $\mathcal{U}$ denotes operator Until. With these two temporal operators, we can define other two useful temporal operators: Eventually ($\Diamond\phi := True \, \mathcal{U}\phi$) and Always ($\Box\phi := \neg\Diamond\neg\phi$).

The semantics of an LTL formula ϕ is interpreted over a linear structure. Let σ denote an infinite sequence of truth assignments to the propositions $p \in AP$ and $\sigma(i)$ denote the set of propositions that are true in position i. For an interpretation σ, we recursively define when an LTL formula ϕ is true at an instant i (written as $\sigma, i \models \phi$) :

- $\sigma, i \models p$ iff $p \in \sigma(i)$
- $\sigma, i \models \neg\phi$, iff $\sigma, i \not\models \phi$
- $\sigma, i \models \phi_1 \vee \phi_2$, iff $\sigma, i \models \phi_1$ or $\sigma, i \models \phi_2$
- $\sigma, i \models \bigcirc\phi$ iff $\sigma, i + 1 \models \phi$
- $\sigma, i \models \phi_1 \mathcal{U} \phi_2$, there exist $k \geq i$ such that $\sigma, k \models \phi_2$ and for all $i \leq j \leq k$, we have $\sigma, j \models \phi_1$

D. LTL and Büchi automaton

This is a closed relationship between LTL and Büchi automaton. It has been proven that any LTL formula can be automatically represented in a non-deterministic Büchi automaton [15] that accepts the traces that satisfies the LTL formula. A Büchi automaton is a tuple $\mathcal{BA}_\phi = (Z, Z_0, \Sigma, \delta, Z_F)$, where Z is a finite set of states, Z_0 is a set of initial states, $\Sigma = 2^{AP}$ is the input alphabet, $\delta \subseteq Z \times \Sigma \times Z$ is a non-deterministic transition relation, and $Z_F \subseteq Z$ is a set of accepting states.

Let $w = w_0 w_1 w_2 \dots$ denote the *input word* of the automaton. A *run* of the automaton *over* an input word w is a sequence of $r_{\mathcal{BA}} = z_0 z_1 z_2 \dots$ such that $z_0 \in Z_0$,

and $(z_i, w_i, z_{i+1}) \in \delta$ for $i \in \mathbb{N}$. A run r is accepting iff $lim(r) \cap Z_F \neq$, where the $lim(r)$ is the set of states that occur in r infinitely often. The run is accepted iff it gets into Z_F infinitely many times. In other words, at least one accepting state has to be visited infinitely often.

E. Abstractions

We assume that a complex task can be accomplished by performing a set of basic skills or functionality that were developed over a long period of research time. To achieve high-level model-based task planning, the first step is to provide an abstract model for each skill. We use the idea of action template to abstract the low level motions. An *action template* is a tuple $a = (Head(a), Pre(a), Eff(a), cost(a))$, where $Head(a)$ is a syntactic expression of the form $act(\theta_1, \theta_2, \ldots, \theta_k)$ where act is a symbol representing the action name and $\theta_1, \theta_2, \ldots, \theta_k$ are associated parameters, $Pre(a) := (p_1, p_2, \ldots, p_n)$ denotes a set of preconditions that need to be satisfied before the action is applied. $Eff(a) := (e_1, e_2, \ldots, e_m)$ denotes a set of effects after the action is applied, $cost(a)$ is the action cost.

Let Act denote the set of all possible actions. In our case, $a \in Act$ maps a set of parameters θ to a trajectory $x(t)$ in the state space. For example, we can create a simple action named $\mathsf{actMove}(\theta_0, \theta_g)$, the parameter θ describes two sets θ_0, θ_g in the work space. The action has a simple precondition as $\{\mathsf{in}(\theta_0)\}$ and effects $\{\neg\mathsf{in}(\theta_0), \mathsf{in}(\theta_g)\}$. The action utilizes an underlying trajectory generator to generate state trajectory $x(t)$ such that $x(0) \in \mathcal{E}(\theta_0)$ and $x(T) \in \mathcal{E}(\theta_g)$. Here, $\mathcal{E}$ stands for a mapping from the work space to the corresponding state space. Also, notice that, when a task specification is described by a set of symbols (normally as a combination of a set of predicates), a mapping between the symbols and the underlying work space is also in-explicitly given.

III. Task Planning

In this section, we describe how our task-level planning system works. The core of our task-level planning system is a classical planner. Motivated by the techniques presented in [7], we show how to use a classical planner to solve the LTL synthesis problem.

A. LTL and classical planning

As pointed by [16], depending on the type of abstractions of the original system (such as deterministic or non-deterministic), there are major four approaches to address an LTL synthesis problem, specifically, automata-based synthesis, Markov Decision Process based synthesis, game-based synthesis, and optimization-based synthesis. In our work, we use the automata-based one. The basic idea of the automata-based method is to combine the system automaton and the specification automaton into a product automaton, and then search for an accepting path over the product automaton. The accepting path of a Büchi automaton can be characterized as a *lasso*, i,e., a *prefix* that is a finite path from an initial state to an accepting state, and a periodic *suffix* that is a loop starting from the accepting state reached by prefix, with an ending at the same accepting state. This result is not surprising because the Büchi acceptance requires it to reach at least one accepting state infinitely often. By looking for the lasso in the product automaton, the synthesis problem is then reduced to a special graph search problem. Notice that the automata-based method cannot be applied to a non-deterministic transition system: because by committing to a particular path during the search, the non-determinism is resolved subjectively. In the latter part of this paper, we show we can apply the automata-based method to a non-deterministic system through system determinization and re-planning.

By constructing the product automaton and finding a prefix-suffix structure over it, the LTL synthesis for a deterministic system can be solved by graph algorithms. However, our goal is to use the standard classical planning algorithms to solve the problem since there are many off-the-shelf computationally-efficient algorithms developed over the years to address the state exploration problems. Using an off-the-shelf planner also implies that we can easily benefit from the advances of planning communities. The key challenge of applying a classical planner to an LTL synthesis problem is that the standard algorithm only handles finite path searching, i.e., it solves the reachability problem between two states. To address the issue, we use the techniques developed by [7] to compile the LTL synthesis problem into a standard classical planning problem and further find an infinite plan.

We first present the definition of the product automaton. The *product automaton* $\mathcal{A}_\mathcal{P} = \mathcal{T} \otimes \mathcal{BA}_\phi$ of the abstraction system $\mathcal{T}$ and the specification Büchi automaton ϕ is a tuple $\mathcal{A}_\mathcal{P} = (Z_\mathcal{P}, Z_{\mathcal{P}0}, \delta_\mathcal{P}, Z_{F_\mathcal{P}}, c_\mathcal{P})$, where

- $Z_\mathcal{P} = S \times Z$ is a finite set of states,
- $Z_{\mathcal{P}0} = \{s_0\} \times Z_0$ is a set of initial states,
- $\delta_\mathcal{P} \subseteq Z_\mathcal{P} \times Z_\mathcal{P}$ is a transition relation, which is defined by the rule $\dfrac{(s \to \bar{s}) \wedge (z \xrightarrow{\mathcal{L}(s)} \bar{z})}{(s, z) \to (\bar{s}, \bar{z})}$,
- $Z_{F_\mathcal{P}} = S \times Z_F$ is a set of accepting states,
- $c_\mathcal{P} : \delta_\mathcal{P} \to \mathbb{R}_{>0}$ is a weight function, where $c_\mathcal{P}((s, z), (\bar{s}, \bar{z})) = c(s, \bar{s})$ for all $((s, z), (\bar{s}, \bar{z})) \in \delta_\mathcal{P}$.

Though the definition of the product automaton is given, we do not explicitly construct such an automaton. Instead, it is incrementally constructed during the forward searching process. Specifically, each transition of product automaton is represented as two successive actions. The specification automaton would first take a move, and then the system would take a move. Additional auxiliary propositions and actions are created to facilitate the process.

Now we explain how to search for a prefix-suffix structure using classical planners. The basic idea is to create a new action template, $\mathsf{actLoop}(z)$, to capture the prefix-suffix structure. Intuitively, during the forward state propagation, the action $\mathsf{actLoop}(z)$ is applicable if the current state z is an accepting state. If we can reach the same accepting state again, we successfully find the loop and solve the problem. To remember the current accepting state reached

by the planner, we first make a of the original proposition set AP, denoted as $\mathsf{req}(\mathsf{L})$, for all $L \in AP$. Those copied propositions are initially all false. As a part of effects of action $\mathsf{actLoop}(z)$, if original proposition L is true, then we can let its corresponding copied proposition $\mathsf{req}(\mathsf{L})$ become true. Similarly, a copy set $\mathsf{nreq}(\mathsf{L})$ is created to remember the false propositions of the reached accepting state. To properly define a goal state, another copy set of propositions $\{\mathsf{end}(L)|L \in AP\}$ is created, together with a new action named actEnd. If we are currently in the mode of searching for a loop, then we can apply the actEnd action to see if the current state satisfies the remembered propositions previously. In other words, if we have two types of conditional effects for the action actEnd, $\mathsf{nreq}(L) \wedge \neg L \rightarrow \mathsf{end}(L)$ and $\mathsf{nreq}(L) \wedge \neg L \rightarrow \mathsf{end}(L)$. Hence if all $\mathsf{end}(L)$ are true, the planner reaches the goal.

Given a deterministic transition system $\mathcal{T} =< S, s_0, Act, \rightarrow_{\mathcal{T}}, AP, \mathcal{L}, c >$, and a Büchi automaton $\mathcal{BA}_\phi$ from the task specification ϕ, we can create a planning problem $P_\phi =< F, I, A, G >$ as shown in Fig. 2, where F denotes the fluents set, I denotes the initial state and G defines the goal state. A is the set of compiled actions, where the detailed definition for each action is given in Fig. 3. Proposition $\mathsf{currBAz}(z)$ indicates whether the current automaton state is z, and $\mathsf{nxtBAz}(z)$ represents the next Büchi automaton state. $\mathsf{accptBAz}(z)$ examines whether the state z is an accepting state in the automaton. Propositions inLp and LpS represent "in loop" and "loop started" respectively. Proposition BAturn is a mode guard proposition. When BAturn is true, the planner apply the transitions from the $\mathcal{BA}_\phi$ by executing the new action actMvBA1. Assuming the input word for the automaton is W, the conditional effects of actMvBA1 are $\{\neg\mathsf{currBAz}(z)\}_{z\in Z} \cup \{W \wedge \mathsf{currBAz}(z) \rightarrow \mathsf{nxtBAz}(z)\}_{z\in Z}$, which describe the transition from state z to state z'. Action actMvBA1 is a helper function for actMvBA2 which is responsible for some book-keeping work. The propositions BAturn, actMvBA1Done are used together with actMvBA1 and actMvBA2 to simulate the move of the specification automaton. The functionality of $\mathsf{req}(L), \mathsf{nreq}(L), \mathsf{end}(L)$ is discussed above. There are also other auxiliary propositions. We also need to modify all actions in original set Act a bit, by adding some extra preconditions and effects, as show in Fig. 4. We use Act_m to denote the set of modified actions and use $a_m \in Act_m$ to denote the modified action from the original action $a \in Act$.

Now we successfully compile the LTL non-synthesis problem into a standard classical planning problem. Such a problem can be solved by a classical planner such as FF [17]. With the help of action $\mathsf{actLoop}(z)$, the infinite plan now becomes finite. Notice that the plan computed from the compilation domain consists of auxiliary actions as well, typically in a form: actMvBA1, actMvBA2, a_m^0, actMvBA1, actMvBA2, a_m^1, ..., $\mathsf{actLoop}(z)$, a_m^k, actMvBA1, actMvBA2, ..., a_m^n, actMvBA1, actMvBA2, actEnd. We can remove the auxiliary actions such as actMvBA1, actMvBA2 during the actual execution.

$$
\begin{aligned}
F :=& \{\mathsf{currBAz}(z), \mathsf{nxtBAz}(z), \mathsf{accptBAz}(z)\}_{z\in Z} \cup \\
& \{\mathsf{actMvBA1Done}, \mathsf{actEndDone}, \mathsf{actNopDone}, \\
& \mathsf{BAturn}, \mathsf{endBA}, \mathsf{inLp}, \mathsf{LpS}\} \cup \{\mathsf{reqLp}(z)\}_{z\in Z} \\
& \cup \{\mathsf{req}(L), \mathsf{nreq}(L), \mathsf{end}(L)\}_{L\in AP} \cup AP \\
I :=& \{\mathsf{currBAz}(z)\}_{z\in Z} \cup \{\mathsf{BAturn}\} \cup s_0 \\
A :=& \{\mathsf{actNop}, \mathsf{actEnd}, \mathsf{actMvBA1}, \mathsf{actMvBA2}\} \cup \\
& \{\mathsf{actLoop}(z)\}_{z\in Z_F} \cup Act_m \\
G :=& \{\mathsf{inLp}, \mathsf{LpS}, \mathsf{endBA}, \neg\mathsf{BAturn}\} \cup \\
& \{\mathsf{end}(L)\}_{L\in AP}
\end{aligned}
$$

Fig. 2. The compiled classical planning problem given a Büchi automaton and deterministic transition system

$$
\begin{aligned}
Pre_{\mathsf{actMvBA1}} :=& \{\mathsf{BAturn}\} \cup \\
& \{\mathsf{actMvBA1Done}, \neg\mathsf{actEndDone}\} \\
Eff_{\mathsf{actMvBA1}} :=& \{\mathsf{actMvBA1Done}, \neg\mathsf{endBA}\} \cup \\
& \{\neg\mathsf{currBAz}(z)\}_{z\in Z} \cup \\
& \{W \wedge \mathsf{currBAz}(z) \rightarrow \mathsf{nxtBAz}(z)\}_{z\in Z} \\
Pre_{\mathsf{actMvBA2}} :=& \{\mathsf{actMvBA1Done}\} \\
Eff_{\mathsf{actMvBA2}} :=& \{\neg\mathsf{actMvBA1Done}, \neg\mathsf{BAturn}\} \cup \\
& \{\neg\mathsf{nxtBAz}(z)\}_{z\in Z} \cup \\
& \{\mathsf{nxtBA}(z) \rightarrow \mathsf{currBAz}(z)\}_{z\in Z} \\
Pre_{\mathsf{actLoop}(z)} :=& \{\mathsf{acptBAz}(z), \mathsf{currBAz}(z)\} \cup \\
& \{\neg\mathsf{BAturn}, \neg\mathsf{LpS}, \neg\mathsf{actEndDone}\} \\
Eff_{\mathsf{actLoop}(z)} :=& \{\mathsf{reqLp}(z), \mathsf{LpS}, \mathsf{currBAz}(z)\} \\
& \cup \{\neg(z = z') \rightarrow \mathsf{currBAz}(z')\}_{z'\in Z} \\
& \cup \{L \rightarrow \mathsf{req}(L)\}_{L\in AP} \\
& \cup \{\neg L \rightarrow \mathsf{nreq}(L)\}_{L\in AP} \\
Pre_{\mathsf{actEnd}} :=& \{\mathsf{inLp}, \neg\mathsf{actEndDone}, \neg\mathsf{BAturn}\} \\
Eff_{\mathsf{actEnd}} :=& \{\mathsf{actEndDone}\} \cup \\
& \{\mathsf{currBAz}(z) \wedge \mathsf{reqLp}(z) \\
& \rightarrow \mathsf{endBA}\}_{z\in Z} \\
& \cup \{\mathsf{req}(L) \wedge L \rightarrow \mathsf{end}(L)\}_{L\in AP} \\
& \cup \{\mathsf{nreq}(L) \wedge \neg L \rightarrow \mathsf{end}(L)\}_{L\in AP} \\
Pre_{\mathsf{actNop}} :=& \{\mathsf{LpS}, \neg\mathsf{BAturn}\} \cup \\
& \{\neg\mathsf{actNopDone}, \neg\mathsf{actEndDone}\} \\
Eff_{\mathsf{actNop}} :=& \{\mathsf{actNopDone}, \mathsf{BAturn}\} \cup \\
& \{\mathsf{LpS} \rightarrow \mathsf{inLp}\}
\end{aligned}
$$

Fig. 3. The details of the new actions added during the compilation process.

$$
\begin{aligned}
Pre_{a_m} :=& Pre_a \cup \\
& \{\neg\mathsf{BAturn}, \neg\mathsf{actEndDone}, \neg\mathsf{actNopDone}\} \\
Eff_{a_m} :=& Eff_a \cup \{\mathsf{BAturn}\} \cup \{\mathsf{LpS} \rightarrow \mathsf{inLp}\}
\end{aligned}
$$

Fig. 4. Modification applied to all $a \in Act$ to get a new action set Act_m

B. Iterative planning and reactive synthesis

One of the biggest challenges in robotics is that the environment is often dynamic and partially unknown in prior. A reactive planner can be used to address the issue. Our approach to the reactive synthesis problem is based on the common practical approach used in robotics, specifically, iterative planning. In the real application, it often does not make much sense to plan a full policy that can handle every environment changes. Instead, it is more meaningful to plan a strategy based on the current state, with some assumptions regarding the environment, and further update the planning model and re-plan a strategy when the assumption is violated. Such an idea is also frequently used in the motion planning layer when handling dynamic obstacles. Instead of accounting all possible locations of the obstacles and synthesis a huge policy beforehand (which is extremely computational intensive), the motion planning algorithms often directly assume the unknown area is obstacle-free and then plan a trajectory based on the current state. The vehicle executes the generated trajectory, senses the environment along the way and updates the abstraction (e.g., the occupancy grid map) and generates a new plan if necessary until it reaches the goal.

We apply a similar idea to the task level as well. Instead of using a powerful synthesis tool to synthesis a policy that accounts for all environment uncertainty, we determinize the original non-deterministic transition system by using the current observed state of the environment. Then we can search for a lasso sequence over the product automaton of the determinized transition system and specification automaton, and generate a linear plan as discussed above. When the monitored current state differs from the predicated task-level state, the system is re-determinized based on current observation, and we start a new planning cycle from the current state.

C. Advantages

There are several advantages of using a classical planner and re-planning strategy to handle the reactive synthesis problem.

- By determinizing and compiling the problem into a standard classical planning problem, we can use any existing classical planning algorithm to solve the problem and thus can easily benefit from the advances of classical planning community.
- Using a planning approach, the product automaton is incrementally built during the forward searching process and does not require an explicit representation. In our approach, the representations of system transitions system and specification automaton are essentially decoupled in the planning domain. So during the execution, if the system abstraction needs modification, this is no need to re-calculate the specification automaton and re-compile it into planning domain.
- The compilation process for each action is standard and independent from the specification automaton. In other words, even if task specification changes, there is no need to re-compile the system dynamics.
- For a task problem that does not require temporal constraints and only cares about the reachability problem, such as delivering a set of parcels from one place to another, we can directly use the classical planner to solve the problem without using LTL approach.
- We use re-planning to solve the uncertainty issue, and hence the resulted task plan can still be represented in a linear form, instead of a policy represented by a finite state machine. It is easier to refine a linear task plan hierarchically, compared with a finite state machine.
- Planning community usually uses PDDL and its variants to specify the planning problem. By compiling an LTL synthesis problem into a planning problem, we can also use PDDL to specify the system dynamics, which is easier and more intuitive to work with for non-experts.

IV. BT-BASED INTERFACE

This section presents a behavior-tree based interface between the high-level abstract task plan and lower-level motion behaviors. The major responsibilities of the interface are to 1) gradually refine the abstract task plan into a more detailed one until the plan consists of only primitive motion behaviors; 2) provide the communication and feedback between underlying continuous system and its high-level abstractions, and help to update the high-level abstraction if necessary; 3) monitor the changes of environmental states, detect any violation of current planning assumptions and further request the high-level planner to re-plan.

A. Behaviour tree

The behavior tree is a tree-based model that organizes a group of modular behaviors, where each behavior can be either composite or primitive. It is very much similar to the Hierarchical Task Network (HTN), a well-studied planning framework based on a set of refinement models. The BT framework can be viewed as an extension of the HTN added with additional execution policies. Such an extra feature regarding to execution part allows us to interleave the planning and execution, which is extremely useful for real applications. As a tree structure, BT can easily represent and manage a hierarchy of multi-level abstractions. Also, BT is user-friendly and allows the user to directly specify the mission plan by simply grouping the different behaviors without going through a planner [12].

The execution of the BT starts from the root. Execution permission (the tick signal) is gradually traversing the tree (from the parent node to child node) at a fixed frequency. After receiving the tick signal from the parent node, the child node begins to run and returns the running status to its parent when it receives the tick signal at the next cycle. There are only three types of status, i.e., *RUNNING*, *SUCCESS* and *FAILURE*. The leaf nodes (either being a condition node or an action node) are the nodes that can be directly executed, such as the motion commands. The internal nodes are normally referred to as control flow nodes.

According to their node type, it selects a different child to run at each cycle and thus controls the execution of the tree. Essentially, each control flow node is a modular local decision process. It receives and analyzes the feedback from the children and then makes a selection/decision for the next cycle. One can implement any new type of control flow nodes. The most common yet important ones are *selector* and *sequence*. Selector behaves like a logic operator *OR* and runs its children one by one from left to right. It returns *SUCCESS* if one of children returns *SUCCESS*; it only returns *FAILURE* if all children returns *FAILURE*. On the other hand, sequence node behaves like logic operator *AND*, it evaluates its children one by one from left to right and only returns *SUCCESS* if all children return *SUCCESS* and return *FAILURE* if one of the children returns *FAILURE*. There is also a special type of control flow nodes, called *decorator*. The decorator has only one child, and the control policy is user-defined. Common decorators include *loop*, *timer*, *inverter*, *counter*.

B. High level plan representation and refinement

A task plan resulted from the high-level planning system is a sequence of actions, where each action is associated with an action template consisting of a set of preconditions and the set of effects. Each action also maps to a less abstract action (for instance, a fragment of the continuous trajectory). During the plan execution, we gradually refine an abstract action into a less abstract one and finally refine it into some actual commands, i.e., a set of instructions that the system can directly execute. Before actually executing the action, no matter it is a concrete action or an abstract one, we would like to check the corresponding preconditions. After applying the action, we would like to update effects for the planning model of correct abstraction level.

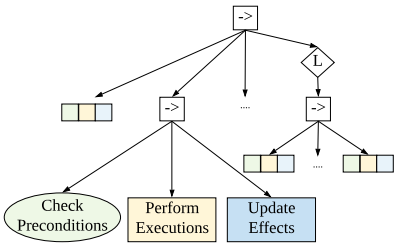


Fig. 5. The behavior tree based LTL plan representation

Graphically, as shown in Fig. 5, we denote the action in a task plan as a colored triple. The green block denotes the set of preconditions that we need to check during the execution. The yellow block denotes the set of refinement executions that we are going to execute. The blue part denotes the set of effects that we want to update after the action being applied. Considering the standard linear representation of the plan, the execution process for each action can be represented as: 1) check the preconditions, 2) perform actual executions and 3) update the effects, which can be easily translated into a behavior-tree representation. As showed in Fig. 5, the root of the tree is a sequence node. Each action in the linear plan now becomes a child branch of this sequence node. The three parts of the action, i.e., preconditions, executions, and effects, are naturally grouped by a sequence node as well. The precondition part is denoted by the condition node of the behavior tree. To represent the prefix-suffix structure of the LTL plan, we use the *loop* decorator (the node with letter L shown in the Figure. 5) to control the execution of the periodic suffix.

C. Iterative planning and abstraction updating

During the plan execution, there often is a need to update the high-level task planning model due to environmental uncertainties. For instance, a particular action may fail, and one has to modify the task model by removing the transition and allow the high-level task planner to re-plan a new action. We use a toy example to illustrate how to use behavior tree to update the high-level abstraction and trigger a re-planning process automatically. Consider a simple action named $\mathsf{move}(W_1, W_2)$, where parameters W_1 and W_2 denote some region in the underlying workspace. It has a single precondition inW_1 and effects $\mathsf{inW}_2, \neg\mathsf{inW}_1$. We first modify the action by adding an extra precondition $\mathsf{trans\_ok\_move}(W_1, W_2)$. The basic motivation of adding such a proposition is that, before the vehicle performs the flight, there is no accurate way to know whether the action can succeed or not. Normally, we often *assume* the action can be applied. Hence, we create a proposition to indicate this *belief*. During the real flight, if the action fails, we then update the belief and make the proposition false. As shown in Fig. 6, the root of this action $\mathsf{move}(W_1, W_2)$ now becomes a *selector*. The left branch is the standard action triple (precondition, execution, effects). If this branch succeeds, then the selector succeeds; otherwise, the selector runs its right branch, which is updating the task-level model by deleting the wrong belief $\mathsf{trans\_ok\_move}(W_1, W_2)$ away. To automatically trigger a re-planning process, we add an *inverter* to make the right tree branch false. As both of children return *FAILURE*, the selector returns failure to its parent (i.e., the root sequence node) as well and thus the re-planing is triggered.

Also, we can add a special checking branch as the leftmost child of the original root node, to monitor the current environment state. If the state does not match the current predicated task-level state, a replanning is triggered.

D. Communications with lower-level motion layer

We continue to consider the example shown in Fig. 6. As mentioned, a composite behavior should be refined into a more primitive behavior that the low-level motion layer can directly execute. Assume that the low-level motion planner can be abstracted into a function $\mathsf{move\_cmd}(c, w)$, where the parameters c and w are exact waypoints in state space. To refine the task-level action $\mathsf{move}(W_1, W_2)$ into motion-level action $\mathsf{move\_cmd}(c, w)$, we are essentially performing another abstraction which abstracts the continuous region

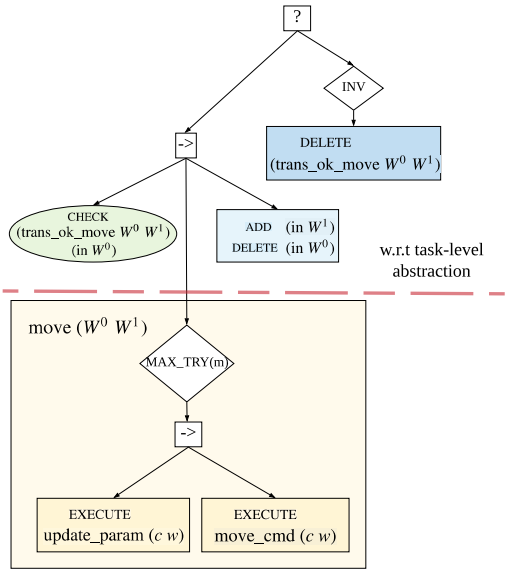


Fig. 6. Behaviour tree-based representation for the action move

(W_1, W_2) into a set of waypoints. The transition between these waypoints is determined by the motion planner online. We create a function named $\mathsf{update\_param(c, w)}$ to *ground* the symbols c and w. Our approach is similar to the one presented by [18], which uses a set of suggesters and corresponding tests to generate valid instantiations for a symbol according to some pre-defined criteria. Besides, we have a database to store the current instantiation of the symbol. If the suggester successfully finds a new value that passes the test, the corresponding value of a symbol in the database would be updated. In our work, both human experts and algorithms can provide information to the database. The symbol grounding process then can be viewed as a query to a dynamic variable database that keeps the most updated value of a symbol. Imagine that the vehicle is required to fly from W_1 to W_2 for some inspection tasks at W_2. The human operator has a rough idea about the location of the building and supplies a default value to the database. However, there is no possible way for the vehicle to know exactly whether the default waypoint is in a collision or not before the actual flight. It can only learn such a geometric detail when it reaches the nearby of W_2. Our dynamic symbol grounding procedure allows us to refine the symbol value during the plan execution continuously. Notice that for the parameter c of the action $\mathsf{move\_cmd}(c, w)$, we always use the current state of the vehicle. In Fig. 6, we add a decorator named $\mathsf{MAX\_TRY}(m)$ to indicate there are maximum m trials for re-instantiations.

V. Simulation

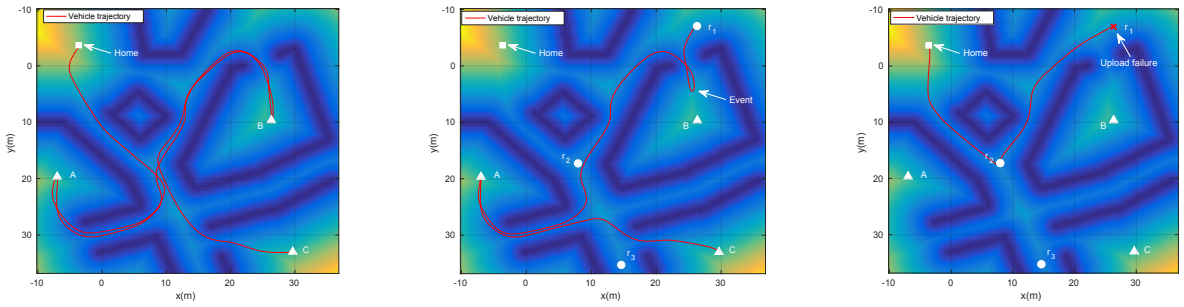
We present a simulation to illustrate our framework. Consider a surveillance task that the MAV is required to loop around three sites A, B and C and search for a target. If the target is found, the MAV needs to communicate back the acquired information to home base. However, due to the payload limit, the vehicle is not able to carry out a powerful communication hardware and thus can only upload the information at one of three uploading sites $r1$, $r2$ and $r3$. Once the information is successfully uploaded, the vehicle is required to return home immediately. Assume that MAV has two basic types of actions (i.e., two action templates). The first one is $\mathsf{actMove}(\theta_1, \theta_2)$, meaning that the vehicle can fly from one place θ_1 to another place θ_2. The precondition is $\{\mathsf{at}(\theta_1)\}$ and the effect is $\{\mathsf{at}(\theta_2), \neg\mathsf{at}(\theta_1)\}$. Notice that θ_1 and θ_2 are template parameters with *parameter type* Place, which can be instantiated to other Place-type symbols such as A, B and C in our case. Another action, is the uploading action $\mathsf{actUploadInfo}(r)$, where parameter r denotes a possible uploading location. The precondition is $\{\mathsf{at}(r), \mathsf{targetFound}, \mathsf{canUpload}(r)\}$, the effect is $\{\mathsf{targetReported}\}$. We then specify the task as an LTL formula:

$$\phi = \phi_{search} \wedge \phi_{upload} \wedge \phi_{return}$$
$$\phi_{search} = \Box((\Diamond p_{\mathsf{A}} \vee p_{\mathsf{TF}}) \wedge (\Diamond p_{\mathsf{B}} \vee p_{\mathsf{TF}}) \wedge (\Diamond p_{\mathsf{C}} \vee p_{\mathsf{TF}}))$$
$$\phi_{upload} = \Box((p_{\mathsf{TF}} \vee p_{\mathsf{TR}}) \rightarrow \Diamond p_{\mathsf{TR}})$$
$$\phi_{return} = \Box(p_{\mathsf{TR}} \rightarrow (\bigcirc p_{\mathsf{H}}))$$

where p_{A}, p_{B}, and p_{C}, p_{H} represent propositions $\mathsf{at(A)}$, $\mathsf{at(B)}$, $\mathsf{at(C)}$ and $\mathsf{at(Home)}$ respectively. p_{TF} and p_{TR} represent propositions targetFound and targetReport.

Initially, we assume that the MAV is at the place Home and there is no targetFound event occurring. We also assume that all three uploading locations can upload the information. The initial state is $s_0 = \{\mathsf{at(Home)}, \neg\mathsf{targetFound}\} \cup \{\mathsf{canUpload}(r1), \mathsf{canUpload}(r2), \mathsf{canUpload}(r3)\}$. A Büchi automaton can be generated using the open source tool LTL2BA provided in [19]. Following the compilation procedures described in Section III-A, we compile the whole problem into a standard classical planning problem in PDDL format. A plan is then calculated using a classical planner and represented as a behaviour tree (see Section IV-B). The symbols are grounded using the ideas of suggesters presented in Section IV-D. The state targetFound is constantly monitored against the actual environment. When the event occurs, we re-update the model and re-plan for the task-level.

A simulation (see Fig. 7) has been conducted to demonstrate the effectiveness of the proposed planner. The simulation is based on a high-order quadrotor model and an indoor environment represented by a grid map. The color of a grid denotes the distance to the nearest obstacle. The darker the color, the shorter the distance. The vehicle is initially cycling among three sites A, B and C to search a target. When the vehicle is on the way to site B for the second time, a target is detected. The original assumption of the initial state ¬targetFound is violated. As a result, a replan process is triggered. The MAV changes the initial plan and flies towards $r1$ to report the necessary information, as shown in Fig. V. However, due to some reasons, the MAV fails to upload information at $r1$. Notice that such a failure can only be learned during the execution not beforehand. The BT automatically updates the task-level model using the similar technique presented in Section IV-D (see Fig. 6). Specifically, the BT deletes the proposition $\{\mathsf{canUpload}(r1)$



(a) (b) (c)

Fig. 7. (a) The MAV loops around three sites to search a target. (b) A target is found, and the initial assumption is violated. A re-planning process is triggered, and the MAV now flies to uploading site $r1$ to report information. (c) During the execution, the MAV finds that the uploading action is unsuccessful. The BT modifies the task-level model, and a re-planning process is triggered again. The MAV flies to $r2$ to upload information. After the uploading, the MAV flies back home.

from the initial state and trigger a re-planning process again. The MAV then selects $r2$ as uploading site and successfully uploads the necessary information. After the uploading, as required, the MAV flies back home. Notice that, during the whole planning and execution process, we only modify the initial state of the compiled problem. There is no need to re-compile other things such as Büchi automaton.

VI. Conclusion

In this paper, we have presented a task and acting framework for micro aerial vehicles. We use linear temporal logic to specify a task formally. By interleaving the planning and execution, we present a classical planner based approach to address the reactive synthesis problem, which can account for task-level environment uncertainties. Specifically, a non-deterministic transition system is determinized, and a classical planning problem is defined over the product automaton of the transition system and specification automaton by a specific compilation process. During the execution, we gradually update our system abstraction and perform re-planning if the planning assumption is violated. Besides, we present a behavior-tree based interface to carry out the task plan and combine the task-level and motion-level autonomy. The behavior tree refines the high-level plan into real concrete commands and automatically monitors the execution process and triggers high-level model updating and re-planning if necessary.

References

[1] J. Chen and S. Shen, "Improving octree-based occupancy maps using environment sparsity with application to aerial robot navigation," in *2017 IEEE International Conference on Robotics and Automation (ICRA)*, May 2017, pp. 3656–3663.

[2] S. Liu, M. Watterson, K. Mohta, K. Sun, S. Bhattacharya, C. J. Taylor, and V. Kumar, "Planning dynamically feasible trajectories for quadrotors using safe flight corridors in 3-d complex environments," *IEEE Robotics and Automation Letters*, vol. 2, no. 3, pp. 1688–1695, July 2017.

[3] S. Lai, K. Wang, H. Qin, J. Q. Cui, and B. M. Chen, "A robust online path planning approach in cluttered environments for micro rotorcraft drones," *Control Theory and Technology*, vol. 14, no. 1, pp. 83–96, Feb 2016.

[4] N. Piterman, A. Pnueli, and Y. Saar, "Synthesis of reactive (1) designs," in *International Workshop on Verification, Model Checking, and Abstract Interpretation*. Springer, 2006, pp. 364–380.

[5] H. Kress-Gazit, G. E. Fainekos, and G. J. Pappas, "Temporal-logic-based reactive mission and motion planning," *IEEE transactions on robotics*, vol. 25, no. 6, pp. 1370–1381, 2009.

[6] S. L. Smith, J. Tumová, C. Belta, and D. Rus, "Optimal path planning under temporal logic constraints," in *2010 IEEE/RSJ International Conference on Intelligent Robots and Systems*. IEEE, 2010, pp. 3288–3293.

[7] F. Patrizi, N. Lipovetzky, G. De Giacomo, and H. Geffner, "Computing infinite plans for ltl goals using a classical planner," in *Twenty-Second International Joint Conference on Artificial Intelligence*, 2011.

[8] M. R. Maly, M. Lahijanian, L. E. Kavraki, H. Kress-Gazit, and M. Y. Vardi, "Iterative temporal motion planning for hybrid systems in partially unknown environments," in *Proceedings of the 16th international conference on Hybrid systems: computation and control*. ACM, 2013, pp. 353–362.

[9] S. Srivastava, S. Russell, and A. Pinto, "Metaphysics of planning domain descriptions," in *2015 AAAI Fall Symposium Series*, 2015.

[10] S. Srivastava, E. Fang, L. Riano, R. Chitnis, S. Russell, and P. Abbeel, "Combined task and motion planning through an extensible planner-independent interface layer," in *2014 IEEE International Conference on Robotics and Automation (ICRA)*. IEEE, 2014, pp. 639–646.

[11] K. R. Guerin, C. Lea, C. Paxton, and G. D. Hager, "A framework for end-user instruction of a robot assistant for manufacturing," in *2015 IEEE International Conference on Robotics and Automation (ICRA)*. IEEE, 2015, pp. 6167–6174.

[12] M. Lan, Y. Xu, S. Lai, and B. M. Chen, "A modular mission management system for micro aerial vehicles," in *2018 IEEE 14th International Conference on Control and Automation (ICCA)*. IEEE, 2018, pp. 293–299.

[13] A. Marzinotto, M. Colledanchise, C. Smith, and P. Ögren, "Towards a unified behavior trees framework for robot control," in *2014 IEEE International Conference on Robotics and Automation (ICRA)*. IEEE, 2014, pp. 5420–5427.

[14] M. Colledanchise, R. M. Murray, and P. Ögren, "Synthesis of correct-by-construction behavior trees," in *2017 IEEE/RSJ International Conference on Intelligent Robots and Systems (IROS)*. IEEE, 2017, pp. 6039–6046.

[15] C. Baier and J.-P. Katoen, *Principles of model checking*, 2008.

[16] H. Kress-Gazit, M. Lahijanian, and V. Raman, "Synthesis for robots: Guarantees and feedback for robot behavior," *Annual Review of Control, Robotics, and Autonomous Systems*, vol. 1, pp. 211–236, 2018.

[17] J. Hoffmann, "Ff: The fast-forward planning system," *AI magazine*, vol. 22, no. 3, p. 57, 2001.

[18] L. P. Kaelbling and T. Lozano-Pérez, "Hierarchical planning in the now," in *Workshops at the Twenty-Fourth AAAI Conference on Artificial Intelligence*, 2010.

[19] P. Gastin and D. Oddoux, "Fast LTL to Büchi automata translation," in *Proceedings of the 13th International Conference on Computer Aided Verification (CAV'01)*, ser. Lecture Notes in Computer Science, G. Berry, H. Comon, and A. Finkel, Eds., vol. 2102. Paris, France: Springer, Jul. 2001, pp. 53–65. [Online]. Available: http://www.lsv.ens-cachan.fr/Publis/PAPERS/PS/Cav01go.ps