IEEE Robotics and Automation Letters (RAL) paper presented at the
2020 IEEE/RSJ International Conference on Intelligent Robots and Systems (IROS)
October 25-29, 2020, Las Vegas, NV, USA (Virtual)

Convergence Analysis of Hybrid Control Systems in the Form of Backward Chained Behavior Trees

Petter Ögren

***Abstract*— A robot control system is often composed of a set of low level continuous controllers and a switching policy that decides which of those continuous controllers to apply at each time instant. The switching policy can be either a Finite State Machine (FSM), a Behavior Tree (BT) or some other structure. In previous work we have shown how to create BTs using a backward chained approach that results in a reactive goal directed policy. This policy can be thought of as providing disturbance rejection at the task level in the sense that if a disturbance changes the state in such a way that the currently running continuous controller cannot handle it, the policy will switch to the appropriate continuous controller. In this letter we show how to provide convergence guarantees for such policies.**

I. Introduction

Behavior Trees (BTs) were created by computer game programmers as a way to improve modularity and reactivity in the control policies of so-called Non-Player Characters (NPCs) in games [1]. Since then, BTs have been receiving an increasing amount of attention in Robotics [2]–[9]. The reason is that robotics share many high level planning and control problems with game AI, while at the same time, the low level problems, such as force control, grasping and sensing, are extremely challenging in robotics, and almost trivial in virtual worlds. Therefore, game AI designers have encountered the high level problems of composing a large set of low level behaviors into a rational system, earlier than robot researchers. BTs is a tool to handle such compositions in a modular and efficient way.

Feedback is one if the key concepts in control theory, where many kinds of disturbances can be handled by iteratively measuring the state of the system and applying the appropriate control action. In a robot control system, there is often a set of continuous controllers, performing tasks such as moving to a given pose, or grasping an object, and a task switching structure responsible for switching between those continuous controllers. For such systems, we want the disturbance rejection to take place at both the continuous level, handling measurement errors and model uncertainties, and at the switching level, handling events such as objects being moved by external agents, or accidentally dropped by the robot itself.

BTs, especially those using the backward chained design proposed in [8], can provide disturbance rejection on the task level, as described above. An example of such a design can be seen in Figure 1. The detailed workings of a BT will be described below, but before going into details, we can note that the design in Figure 1 would provide the following functionality for a mobile manipulator aiming to move to an object, grasp it, move to a goal region and drop the object there.[1] If, while grasping, the object is moved outside the reach of the gripper, the robot switches to first moving closer to the object, and then switches back to the grasping task. If the object slipps from the grasp while moving towards the goal, the robot stops moving and starts grasping. Finally, if the object slipps from the grasp, but accidentally rolls away and stops in the goal area, the robot stops moving and goes to idle-mode, as the given task is now complete. An extended version of Figure 1 can be found in Figure 5, where additional concurrent objectives are added, together with the possibility of achieving subgoals in different ways.

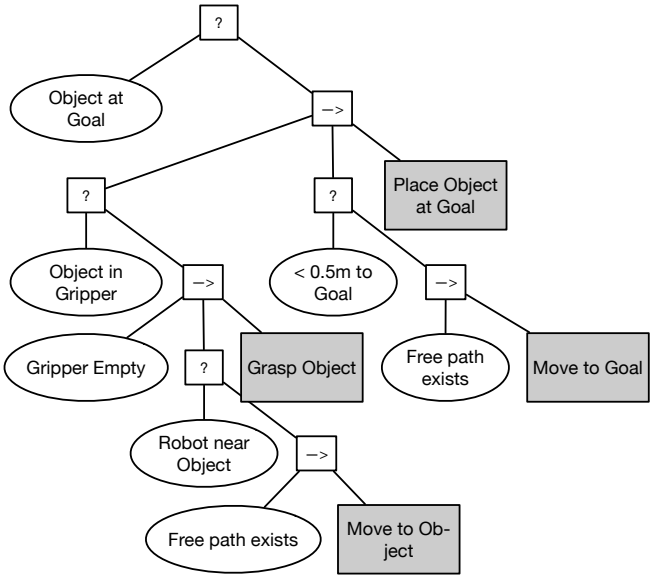


Fig. 1: A BT including the four actions *Move to Object*, *Grasp Object*, *Move to Goal*, *Place Object at Goal*, designed in a way to provide disturbance rejection at the task level. An extended version, including additional objectives and alternative ways to achieve subgoals, can be found in Figure 5.

The objective of doing disturbance rejection using feedback is to achieve convergence to a goal state, with a large region of attraction, even in presence of disturbances and

This work was supported by SSF through the Swedish Maritime Robotics Centre (SMaRC) (IRC15-0046), and FOI project 7135.

P. Ögren is with the Robotics, Perception and Learning Lab., School of Electrical Engineering and Computer Science, Royal Institute of Technology (KTH), SE-100 44 Stockholm, Sweden, petter@kth.se

[1] A video clip illustrating the execution can be found at: https://youtu.be/h6JsYbi5EmI

uncertainties. Thus, a natural question to ask is: *What can be said about converge and region of attraction for the design in Figure 1?*

The main contribution of this letter is that we answer the question above, by making an extension to the state space formalism proposed in [3] and use it to do a convergence analysis of the backward chained design proposed in [8]. Furthermore, we show how the requirements of the convergence proof, in terms of constraints to be preserved, can serve as design specifications when implementing the different components.

The structure of this letter is as follows. First, in Section II we present the related work. Then, in Section III we give a background on BTs and the result in [3]. The main results are then presented in Section IV, followed by an example in Section V, and conclusions in Section VI.

II. Related Work

The problem of convergence analysis of BTs has been addressed in a number of recent papers, [4], [5], [10], [11]. In general, all of them analyze a linear chain of actions, each satisfying the preconditions of the following. This is different from the design proposed in [8] and analyzed here, where several different strategies for achieving a condition can be added, several different preconditions for an action can be listed, and separately achieved, and several top level objectives can be set.

In [10], it was noted how BTs generarlize the Teleo-Reactive approach proposed in [12]. The convergence proof of Teleo-Reactive designs from [12] was also carried over to BTs, and it was shown how Fallback compositions of Sequenced Precondition-Action pairs ensures convergence to the goal state if each actions satisfies a precondition with higher priority (to it's left in the Fallback composition).

An analysis of convergence conditions was also made in [4], [5], with the objective of allowing the user to specify the high level goals, then invoking a HTN planner to optimize the execution, including looking for opportunities of parallell execution, of the tasks to be done. An extended version of BTs (eBTs) was defined to bridge the gap between BTs and HTN-planning. The work presented here goes beyond the work in [4], [5] in that it uses a continuous state formalism, instead of one where the state is composed of a set of conditions that are either true or false.

In [11], the authors define a version of BTs called Robust Logical-Dynamical Systems (RLDS) that they use to prove convergence in the presence of disturbances on the task level, such as someone closing a drawer that was opened by the robot. The work is similar in spirit to [10]. The authors define a set of tasks, each having a runnable condition deciding when it can be run, and each pushing the state towards satisfaction of the runnable condition of the next task. They define a priority ordering, with more downstream tasks having higher priority, to enable the system to skip tasks if external events enable it. They also open up for a set of reactive evasive manoeuvre tasks, having highest priority, even more downstream than the goal. Furthermore, they describe a way to compute implicit conditions that need to hold for future actions. These conditions are added as running conditions for all actions. Thus, [11] is closely related to this work, but this letter goes beyond the work in [11] by similarly identifying conditions that need to hold, but with a higher resolution in time and space. Alternative ways to achieve the same objective can be included, with separate constraints. For example, after picking up an object the robot needs to move to the goal region without accidentally dropping the object, or moving into the designated un-safe zone. But after placing the object, there is obviously no need to be careful not to drop it, whereas the importance of staying out of the un-safe zone remains. A detailed comparison to the results of [11] can be found in Section IV-D.

For papers not dealing with BTs, the results are closely related to the ideas of [13], using a switching scheme to enlarge the region of attraction of a control policy. The metaphor of a funnel for a feedback policy that moves a larger set of initial conditions into a smaller set of final conditions was used. By requiring that the final conditions was inside the initial end of another funnel (controller) of lower index, the authors were able to guarantee that the switching iteratively progressed to controllers of lower index until finally the overall goal region was reached. This letter goes beyond [13] by applying binary conditions, such as *object in gripper*, and separate tasks, such as *grasp object* to handle the complexity of the world. Thus allowing the structured combination of controllers developed to perform completely different subtasks. Furthermore, the notion of constraints being implied across the different tasks is also different from [13].

III. Background

In this section, we give a brief overview of the state space formulation of BTs that was introduced in [3], as well as the design proposed in [8].

A. State space formulation of BTs

Definition 1 (Behavior Tree): A BT is a three-tuple

$$\mathscr{T}_i = \{f_i, r_i, \Delta t\}, \tag{1}$$

where $i \in \mathbb{N}$ is the index of the tree, $f_i : \mathbb{R}^n \to \mathbb{R}^n$ is the right-hand side of an ordinary difference equation, Δt is a time step and $r_i : \mathbb{R}^n \to \{\mathscr{R}, \mathscr{S}, \mathscr{F}\}$ is the return status that can be equal to either *Running* ($\mathscr{R}$), *Success* ($\mathscr{S}$), or *Failure* ($\mathscr{F}$). Let the Running region (R_i), Success region (S_i) and Failure region (F_i) correspond to a partitioning of the state space, defined as follows:

$$R_i = \{x : r_i(x) = \mathscr{R}\},\ S_i = \{x : r_i(x) = \mathscr{S}\},\ F_i = \{x : r_i(x) = \mathscr{F}\}.$$

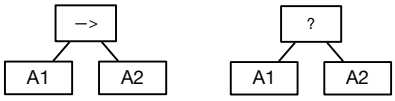
Finally, let $x_k = x(t_k)$ be the system state at time t_k, then the execution of a BT $\mathscr{T}_i$ is a standard ordinary difference equation

$$x_{k+1} = f_i(x_k), \tag{2}$$

$$t_{k+1} = t_k + \Delta t. \tag{3}$$

TABLE I: The four fundamental node types of a BT.

Node type	Symbol	Succeeds	Fails	Running
Sequence	$\rightarrow$	If all children succeed	If one child fails	If one child returns running, and the previous ones succeed
Fallback	?	If one child succeeds	If all children fail	If one child returns running, and the previous ones fail
Action	box	When execution succeeds	When impossible to succeed	During execution
Condition	oval	If true	If false	Never



(a) (b)

Fig. 2: A Sequence node (a) and a Fallback node (b).

The return status r_i will be used when combining BTs recursively, as explained below.

The four fundamental node types[2] of a BT are listed in Table I. An *Action* node directly implements f_i, r_i as above, with $R_i \neq \emptyset$ being the set where the action controls the agent. A *Condition* also implements f_i, r_i, but with $R_i = \emptyset$. Thus a condition never controls the agent, but checks a proposition leading to the execution of some other action. These two node types do not have children, and are thus leafs of the tree. A BT can also be created by composing other BTs, using the two fundamental compositions node types: *Sequence* and *Fallback*, illustrated in Figure 2(a) and 2(b) respectively. The *Sequence* is used when the progression to the next task depends on the success of the previous, such as when drinking from a bottle only makes sense if you succeeded in opening it. If you failed in opening it, or are still trying to open it, there is no point in starting to drink. The *Fallback* on the other hand, is used when progression to the next task is only needed in case of failure of the previous. If you fail to drink from the bottle, you might want to try drinking directly from the tap. But there is no point in doing that if you succeed with drinking from the bottle, or are still trying to drink from the bottle. We will now formally define the two composition types.

Definition 2: (Sequence Compositions of BTs) Two or more BTs can be composed into a more complex BT using a Sequence operator, $\mathscr{T}_0 = \text{Sequence}(\mathscr{T}_1, \mathscr{T}_2)$. Then r_0, f_0 are defined as follows

$$\text{If } x_k \in S_1: \qquad r_0(x_k) = r_2(x_k),\ f_0(x_k) = f_2(x_k) \tag{4}$$

$$\text{else}: \qquad r_0(x_k) = r_1(x_k),\ f_0(x_k) = f_1(x_k) \tag{5}$$

$\mathscr{T}_1$ and $\mathscr{T}_2$ are called children of $\mathscr{T}_0$. Note that when executing $\mathscr{T}_0$, the first child $\mathscr{T}_1$ in (5) is executed as long as it returns *Running* or *Failure* ($x_k \notin S_1$). The second child of the Sequence is executed in (4), only when the first returns *Success* ($x_k \in S_1$). Finally, the Sequence itself, $\mathscr{T}_0$ returns *Success* only when all children have succeeded ($x \in S_1 \cap S_2$).

Definition 3: (Fallback Compositions of BTs) Two or more BTs can be composed into a more complex BT using a Fallback operator, $\mathscr{T}_0 = \text{Fallback}(\mathscr{T}_1, \mathscr{T}_2)$. Then r_0, f_0 are defined as follows

$$\text{If } x_k \in F_1: \qquad r_0(x_k) = r_2(x_k),\ f_0(x_k) = f_2(x_k) \tag{6}$$

$$\text{else}: \qquad r_0(x_k) = r_1(x_k),\ f_0(x_k) = f_1(x_k) \tag{7}$$

Note that when executing the new BT, $\mathscr{T}_0$ first keeps executing its first child $\mathscr{T}_1$, in (7) as long as it returns returns *Running* or *Success* ($x_k \notin F_1$). The second child of the Fallback is executed in (6), only when the first returns *Failure* ($x_k \in F_1$). Finally, the Fallback itself $\mathscr{T}_0$ returns *Failure* only when all children have been tried, but failed ($x \in F_1 \cap F_2$), hence the name Fallback.

We conclude the state space description of BTs with noting that the theoretical analysis of this letter is valid for the abstraction above. Clearly, a real robot system is much more complex than this and switching between different actions will not happen instantly. However we do believe that the results provide important insights for real systems as well.

B. BTs that Succeed in Finite Time

In control theory, the design objective of a controller is often to make it stable (keeping the state close to a given equilibrium point), or asymptotically stable (converging to an equilibrium point). As BTs already include the notion of a Success region S_i, it makes sense to let the design objective be that the state reaches S_i. Furthermore, since compositions of BTs are also a core part of the tool, we need each sub-tree to reach its S_i in *finite time*. Otherwise, if two sub-trees are in a Sequence composition, and the first converges to S_1 as $t \to \infty$, the second one would never execute and the composition would never reach its goal. Thus we make the following definition.

Definition 4 (Finite Time Successful): A BT is finite time successful (FTS) with region of attraction R', if for all starting points $x(0) \in R' \subset R$, there is a time $\tau'(x(0)) = \tau'(x_0)$ and an upper bound τ_b such that $\tau'(x) \leq \tau_b$ for all starting points $x_0 \in R'$, and $x(t) \in R'$ for all $t \in [0, \tau'(x_0))$ and $x(t) \in S$ for $t = \tau'(x_0)$.

As noted in the following lemma, exponential stability implies FTS, given the right choices of the sets S, F, R.

Lemma 1 (Exponential stability and FTS): A BT for which x_s is a globally exponentially stable equilibrium of the execution (2), and $S \supset \{x : ||x - x_s|| \leq \varepsilon\}$, $\varepsilon > 0$, $F = \emptyset$, $R = \mathbb{R}^n \setminus S$, is FTS. The proof can be found in [14].

C. Backchained BTs

The idea of backchaining BTs was first proposed in [8]. Given a list of available actions, with pre- and postconditions,

[2]Note that there is also other node types, including the Parallel composition and so-called Decorators, but since they are not used in this letter they are omitted from the description.

as shown in Table II, the first step of the algorithm converts the list of actions into a set of so-called Postcondition-Precondition-Action BTs (PPA-BTs) as illustrated in Figure 3, each aimed at satisfying a given condition, but invoking the actions only when that condition is not met.

Assume, as in Table II, that there is a postcondition C_2 that can be achieved by either action A_1 or action A_2, that in turn have preconditions C_{11}, C_{12} and C_{21}, C_{22} respectively. Then we create a PPA-BT aimed at achieving the condition C by composing the actions and conditions in the generic way displayed in Figure 3, i.e., each action A_i in sequence after its preconditions C_{ij}, and these sequences in a fallback composition after the main condition C_2 itself. Finally we create similar BTs for each postcondition C_i of Table II.

Actions	Preconditions	Postconditions
A_1	$C_{11}, C_{12}, \ldots$	$C_2, C_3, \ldots$
A_2	$C_{21}, C_{22}, \ldots$	$C_2, C_4, \ldots$
A_3	$C_{31}, C_{32}, \ldots$	$C_3, C_7, \ldots$
$\vdots$	$\vdots$	$\vdots$

TABLE II: The input to Problem 1 is a set of actions and corresponding pre- and postconditions, as illustrated above.

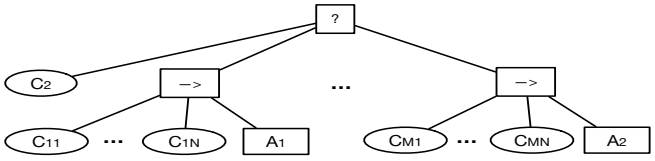


Fig. 3: General format of a PPA-BT. Given Table II we see that the Postcondition C_2 can be achieved by either one of actions A_1 or A_2, which have Preconditions C_{11}, C_{12} and C_{21}, C_{22} respectively. Thus we form the BT above, on a Postcondition-Precondition-Action form (PPA).

Algorithm 1: Creating a Backward Chained BT

Input: Goals $C_1, \ldots, C_M$

$\mathscr{T}_0 \leftarrow$ `Sequence` $(C_1, \ldots, C_M)$;

while *Exists* $C' \in \mathscr{T}_0$ *such that* `Parent`$(C') =$ `Sequence` *AND* C' *is postcondition of some action* **do**

  $\mathscr{T}_{PPA} \leftarrow$ `CreatePPA-BT`(C_i);

  `Replace` C_i `in` $\mathscr{T}_0$ `with` $\mathscr{T}_{PPA}$

Now, given a set of such atomic BTs, and a list of goals of an agent, we can recursively create a BT to achieve those goals using Algorithm 1. The algorithm is illustrated in Figure 4, and Figure 5.

As an example, first create a Sequence node with the goal conditions *In Safe Area* and *Object at Goal* as children. Then we look at Table II and create PPA-BTs for achieving both of these conditions. Both these BTs are shown in Figure 4, with dotted lines to the conditions they achieve. We then replace the original conditions with the BTs achieving them, as shown in the top of Figure 5. Iterating the procedure we find new preconditions in these BTs and create new BTs to achieve them. See all dotted lines in Figure 4, that are replacing the corresponding conditions as shown in Figure 5.

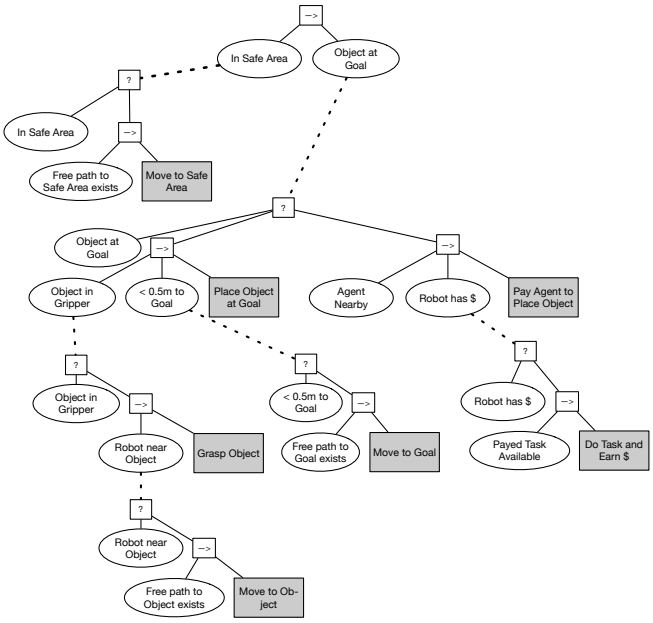


Fig. 4: The dotted lines show how to combine a number of BTs on the PPA-BT form described in Figure 3, with the two overall goals shown at the top, *In Safe Area* and *Object at Goal*. The result can be seen in Figure 5. Note how, for each dotted line, a child of a Sequence node (precondition, something we want to achieve) in the upper BT is replaced by a PPA-BT with that same condition as child of a Fallback node (postcondition, something we know how to achive) in the lower BT.

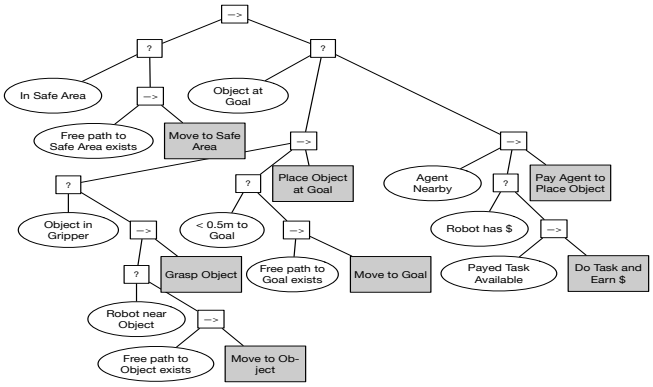


Fig. 5: The resulting BT after applying the design shown in Figure 4.

The main result of this letter is that we provide convergence analysis of such designs, see Theorem 1 below. The result shows that if the actions satisfy some stated properties, the region of attraction is captured by a so-called And-Or tree, as illustrated in Figure 7.

IV. Convergence of Backchained BTs

To appreciate the need for a convergence proof, we start this section with a discussion of things that can go wrong with the proposed design. Then we make some initial assumptions and definitions, followed by the main result. Finally we give some detailed comments on how this result differs from the related work in [10], [11].

A. What can go wrong?

To understand what might go wrong with the design proposed in Algorithm 1 we look at the example in Figure 5. The design is modular in the sense that it locally tries to achieve goals and subgoals. However, there might be global couplings between goals on different levels. In the worst case, the top level goals are mutually exclusive, such as if the goals were *In Safe Area* and *Not In Safe Area*. Clearly, convergence to the satisfaction of both is impossible, but we can still naively apply Algorithm 1. Thus we need to create analytical tools to capture such problems.

There can also be less obvious couplings between goals on different levels. Look again at Figure 5. The robot is tasked to pick and place an object, while staying in the safe area. What if the fire alarm went off, making the entire building where the object and the goal area are located unsafe. Again, clearly there is no feasible solution. In such a case, the robot would first exit the building, getting to the safe area. It would then start to *Move to Object* to pick it up, but as soon as it left the safe area, it would immediately switch to *Move to Safe Area*, and end up chattering on the boundary of the area. To prevent such behavior, and to guarantee convergence when possible, we need a way to identify potential goal conflicts across the whole BT. The solution to the problem will be to identify and propagate information on potential conflicts in terms of additional constraints (see the ACCs of Definition 7) of the actions. *Move to Safe Area* is top priority and will have no such constraints, but *Move to Object* will be extended with the constraint of not violating *In Safe Area*, and *Move to Goal* will have the two constraints *In Safe Area*, and *Object in Gripper*, see Table III. As discussed in Remark 1 below, these constraints can be seen by a human designer as design specifications for the action implementations.

B. Initial Definitions

Preconditions and postconditions are central concepts of Algorithm 1, thus we need to define them in the statespace context.

Definition 5: (Preconditions and Postconditions in Statespace form). If a BT $\mathscr{T}_i$ is FTS, we denote a set of conditions $\{C_j\}_{j\in J}$ such that $\cap_{j\in J} S_j = R_i'$ the preconditions, and a condition C_k such that $S_i \subset S_k$ a postcondition of $\mathscr{T}_i$.

The class of BTs we will analyse is defined as follows.

Definition 6: (Backward chained BT) A Backward chained BT is a BT that is constructed recursively out of PPA-BTs, starting from a set of desired top level goal conditions in a Sequence, as described in Algorithm 1.

Definition 7 (Active Constraint Conditions (ACC)): Given a BT $\mathscr{T}_0$, and an action A_i in that BT, the Active Constraint Conditions $ACC(i)$ of A_i is the sets of conditions, apart from the preconditions of A, that needs to return *Success* (be true) for A to execute.

Examples of ACC can be found in Table III, and Figure 6. Note that we ignore the conditions that need to return *Failure* (be false) for A to execute. Also note that the preconditions is not part of the ACC. The reason is that we do not want actions to violate their ACC, but sometimes we do want actions to violate their preconditions, if that implies reaching their postconditions. An example of this is when the agent is placing an object at a goal location. A precondition is to have *Object in Gripper*, but that precondition is violated in the act of placing the object at the goal location.

Actions A_i	Postconditions of A_i (Objectives)	Active Constraint Conditions $ACC(i)$
Move to Safe Area	In Safe Area	-
Move to Object	Near Object	In Safe Area
Grasp Object	Object Grasped	In Safe Area
Move to Goal	Near Goal	In Safe Area AND Object in Gripper
Place Object	Object at Goal	In Safe Area
Do Task and Earn \$	Robot has \$	In Safe Area AND Agent Nearby
Pay Agent to Place Object	Object at Goal	In Safe Area

TABLE III: All ACCs of the BT in Figure 5. Note how preconditions are not included in $ACC(i)$. Only possible non-local conflicts are captured, such as when executing *Move to Object*, the agent has to satisfy the constraint *In Safe Area*, otherwise, the execution will be interrupted.

Lemma 2 (Finding ACCs): Given a BT $\mathscr{T}_0$, and an action A_i in that BT. Let P be the unique path from A_i to the root. Let N_{0S} be the set of Sequence nodes on P, with the exception of the parent of A (to exclude the preconditions). Let N_1 be the set of children of N_{0S} that are to the left of P, these must return success for A to execute. Let N_{1C} be the conditions in N_1. Let N_{2C} be the conditions that are the first child of Fallbacks that are in N_1. Then $ACC(i) = N_{1C} \cup N_{2C}$.

Proof: By design, using Algorithm 1, P starts with a Sequence (root) node, then has alternating Fallback and Sequence nodes, until it ends with a Sequence node as parent of A. $P = N_{0F} \cup N_{0S} \cup Parent(A) \cup A$. For A to be executed, every child to the left of P in N_{0F} (the Fallbacks) must return Failure, and every child to the left of P in N_{0S} (the Sequences) must return Success. Since we are looking for nodes returning Success we can ignore N_{0F}. The children of N_{0S} are either constraints, that we denote N_{1C} or Fallbacks that we denote N_{1F}. Due to Algorithm 1, the first child of a Fallback in N_{1F} is a constraint, we denote these N_{2C}. Furthermore, N_{1F} only returns Success when N_{2C} does. Thus, $N_{1C} \cup N_{2C} = ACC(i)$ is the set that needs to return success (in additions to the preconditions of A) for A to execute. ■

C. Main Result

Now we define the type of BTs that will be shown to be convergent.

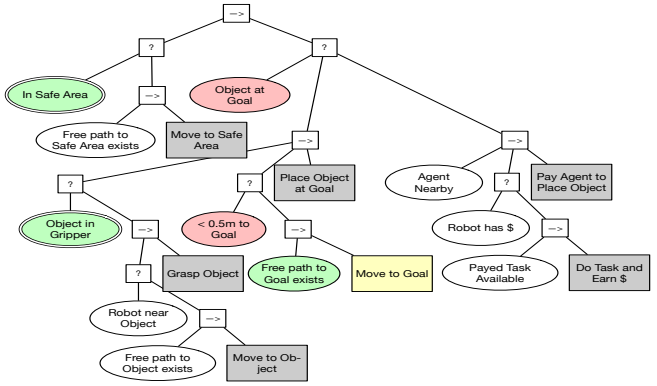


Fig. 6: Example execution of the BT in Figure 5. For *Move to Goal* (yellow) to be executed, all green conditions must be evaluated True and all red conditions must be evaluated False. The green ones include both preconditions (single stroke) and ACC constraints (double strokes). This implies that the agent must chose a path inside the safe area, and move smooth enough to prevent the object from slipping out of the grasp. The corresponding constraints for all actions can be found in Table III. Note that while violating green conditions is often not desired, making red conditions true is in general positive. If, while moving towards the goal, the object somehow ends up at the goal before we deliberately place it there, this is a good thing.

Definition 8 (Non conflicting BTs): A Backward chained BT $\mathscr{T}$, such that each action A_i in $\mathscr{T}$ satisfies the following requirements it is called Non Conflicting. A_i is FTS. The set $(R_i' \cup S_i) \bigcap_{j \in ACC(i)} S_j \neq \emptyset$ is invariant under f_i.

There are two key parts of this definition. First, A_i being FTS means that it will *locally* satisfy its postcondition. Second, requiring the set $(R_i' \cup S_i) \bigcap_{j \in ACC(i)} S_j$ to be non-empty and invariant under f_i means that it will not violate its $ACC(i)$, which is a *global* property, across the whole BT.

Remark 1 (Controllers that keep ACC sets invariant): From the definition above, it is clear that we need to be able to design controllers $f_i(x)$ such that the set $(R_i' \cup S_i) \bigcap_{j \in ACC(i)} S_j$ is invariant under f_i. To see what this might mean, we look at the example. As noted in Table III, *Move to Goal* has to satisfy *In Safe Area* and *Object in Gripper* If the un-safe area is static, invariance can be guaranteed by planning paths to avoid it. If the un-safe area is moving, an approach to avoid moving obstacles might be needed. Furthermore, to keep *Object in Gripper*, a smoother and slower motion might be needed when executing *Move to Goal*, than when executing *Move to Object* (when the gripper is empty). Finally, we note that a general approach to satisfy controller constraints is the Control Barrier Function approach [15].

Before stating the main Theorem, we need a compact way of describing the region of attraction of the BT. The design of Algorithm 1 is such that actions work to satisfy preconditions of other actions. Several such precondition might be needed, and each of those might be achievable in different ways. It turns out that this fairly complex dependence can be captured

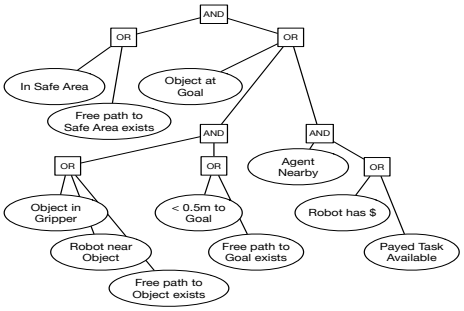


Fig. 7: The resulting And-Or Condition tree after applying Definition 9 to the design shown in Figure 5. As can be seen, if e.g., *Free path to Safe Area exists* AND *Free path to Object exists* AND *Free path to Goal exists*, the problem will be solved.

by a straightforward transformation of the BT into an And-Or tree, as follows.

Definition 9 (And-Or BTs): Given a BT $\mathscr{T}_0$, the corresponding And-Or BT $\mathscr{T}_0^{AO} = AndOr(\mathscr{T}_0)$ can be created by removing all actions, replacing all Sequence nodes with AND operators and Fallback nodes with OR operators.

Note that when all children are conditions, returning only Success/Failure, the Fallback is completely analogous to an OR function and the Success is analogous to an AND function. Thus, $\mathscr{T}_0^{AO}$ above can still be considered a BT. An example of an And-Or BT can be found in Figure 7.

Lemma 3 (Success regions of And-Or BTs): The success region of $\mathscr{T}_0^{AO} = AndOr(\mathscr{T}_0)$ satisfies the following:

$$S_0^{AO} \subset S_0 \cup R_0.$$

Proof: Given a point $x \in \mathscr{T}_0^{AO}$ we need to show that $x \in S_0 \cup R_0$. Since the top level node in $\mathscr{T}_0^{AO}$ is an AND, we know that at least one AND node returns True. If there is only one AND node returning True, we know that all top level conditions are satisfied and $x \in S_0$. If more than one AND node returns True, there is a set of action nodes in $\mathscr{T}_0$ with all preconditions being satisfied. If $x \notin S_0$, one of these is executed and returns running, thus $x \in R_0$

■

Assumption 1: (Reversible Actions) We assume that the set S_0^{AO} of $\mathscr{T}_0^{AO}$ is invariant with respect to the actions $f_i(x)$ of $\mathscr{T}_0$.

In the example, this excludes cases such as when there is first a free passage to the goal, but while getting the object, this path is somehow destroyed or permanently blocked, making it impossible to achieve the overall objectives, or the object is dropped so that there is no free path to it.

Assumption 2: (Restarting Failed Subtrees) While executing $\mathscr{T}_0$, if the sub-BT generated by applying Algorithm 1 to a condition C' returns Failure, i.e., no path to satisfy C' is found, then this sub-BT will not start to execute as a consequence of actions in other parts of $\mathscr{T}_0$.

This excludes infinite loops such as when the agent realizes that there is no free path to the goal, and thus drops the object to let another agent deliver it, but then immediately realizes that it needs to pick it up again to move it to the

goal itself, and then realize again that there is no free path to the goal, and so on. If such problems occur, they can be addressed by re-ordering fallback options such that the option with the least amount of failures has first priority.

We are now ready to state the main result of the letter.

Theorem 1: (Region of Attraction of Non Conflicting BTs) The region of attraction of a Non Conflicting BT $\mathscr{T}_0$ and its success region is a superset of the success region of $\mathscr{T}_0^{AO}$, that is, $S_0^{AO} \subset S_0 \cup R_0'$. Furthermore, $\mathscr{T}_0$ is FTS with $\tau_{b0} \leq \Sigma \tau_{bi}$ for all action A_i in $\mathscr{T}_0$.

Proof: Assume that we start in a state $x \in S_0^{AO}$. By Assumption 1 and Lemma 3, we know that $x(t) \in S_0^{AO} \subset S_0 \cup R_0 \forall t$, there will always be an action in $\mathscr{T}_0$ that executes, or the state is in S_0, returning Success. Thus we need to show that the state will reach S_0 in finite time.

Number all nodes of $\mathscr{T}_0$ using a depth first algorithm, starting with 0 at the root. Number the children in the usual fashion, left to right.

Assume that action A_k, with number k, is executing. We will now show that in finite time, either A_k leads to S_0, or to another action A_m with a higher number $m > k$. Since there are a finite number of nodes, this implies that S_0 is reached in finite time.

Let P be the path in $\mathscr{T}_0$ between A_k and A_m. P is unique since $\mathscr{T}_0$ is a tree. Let N_* be the node closest to the root in P. N_* has at least two children, and is therefore either a Sequence or a Fallback. We need to show that $m > k$, which is equivalent to A_m being to the right of A_k in $\mathscr{T}_0$. There are four possible combination of left/right and N_* being a Fallback/Sequence, all these are illustrated in Figure 8, which depicts a general subtree created using Algorithm 1. A_{m1} is to the left, with N_* being a Fallback, A_{m2} is to the left, with N_* being a Sequence, A_{m3} is to the right, with N_* being a Sequence, A_{m4} is to the right, with N_* being a Fallback. Note that in the figure, A_{mi} symbolizes either the action A_m, or a subtree containing A_m.

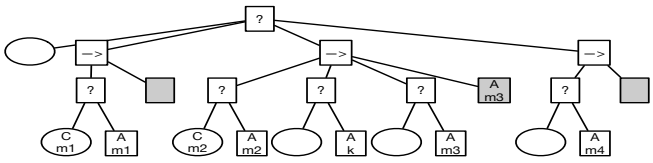


Fig. 8: Illustration of the proof. A_{mi} symbolizes possible locations of either action A_m or a subtree containing A_m. Note that there are two A_{m3}.

The most likely option is that A_k satisfies its postcondition, leading to the execution of A_{m3}. Either way both A_{m3} and A_{m4} have $m > k$ as desired. A_{m2} cannot execute, since C_{m2} was returning Success for A_k to execute, and C_{m2} is a part of $ACC(k)$ which is invariant. Finally, for A_k to execute, the subtree A_{m1} must return Failure. But by Assumption 2, A_{m1} will not change its return status as an effect of A_k. Thus A_{m1} will not execute.

Therefore, each action A_i is followed by another one with higher index until S_0 is reached. Finally, the total execution time is bounded by $\tau_{b0} \leq \Sigma \tau_{bi}$, since each A_i is executed at most once.

■

D. Flattened or Canonicalized BTs

A natural question to ask is how the result in Theorem 1 relates to the convergence proofs of the other works mentioned in Section II. The proofs in [10] and [11] assume that the BT is on the following form: *Fallback(Sequence(C_1,A_1) ... Sequence(C_N,A_N))*. It is clear that the Backward Chained BTs considered here can be written on the flattened form above. However, as seen in Figure 9, this destroys some of the structure. The proofs of [10] and [11] rely on each action satisfying the preconditions of an action to its left (higher priority, closer to goal). But, in Figure 9 it is clear that some of the precondition are actually composed of several conditions, that are in turn achieved by different actions. Thus there is no clear progression from right to left. In fact, if the robot starts out in the un-safe area, there is an initial progression from left to right when reaching the safe area and starting to move towards the object. In [11], the *implicit conditions* correspond to the ACCs of this letter. But in the flattened BT, there is no obvious way to see which ACC should be considered where. Thus we conclude that Theorem 1 is not a special case of the results in [10], [11].

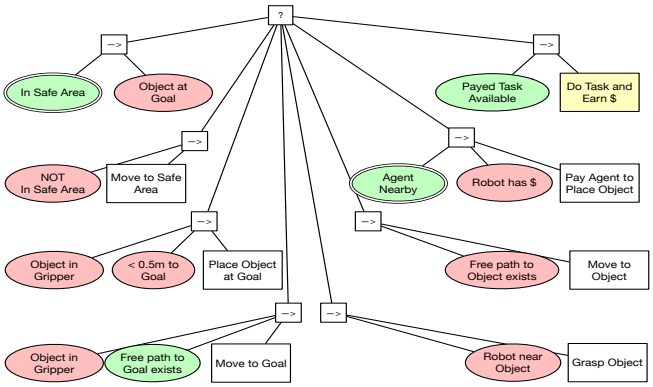


Fig. 9: The BT from Figure 5 written on the flattened form. The green and red conditions return Success and Failure respectively. Note that when *Do Task and Earn $* is executing, there is no clear way to separate the satisfied constraints that can be violated (green single stroke) from the ones that cannot (green double stroke). However, this information is available in the ACC of the Backward Chained BT.

V. The Mobile Manipulator Example

The mobile manipulator example has been a recurring theme of this letter[3]. But to summarize, we start with the high level goal conditions *In Safe Area* and *Object at Goal*. Then we apply Algorithm 1 as illustrated in Figure 4 to get the result of Figure 5. Now, Theorem 1 tells us that

[3]A video illustrating the execution can be found at: https://youtu.be/h6JsYbi5EmI

if the proper requirements are satisfied, we can expect convergence, reaching both high level objectives. Thus, if we start inside $S_0^{AO} \subset S_0 \cup R_0'$ then the goals will be reached within $\tau_{b0} \leq \Sigma \tau_{bi}$. S_0^{AO} is the success region of the And-Or BT, as illustrated in Figure 7, and shows that as long as e.g., *Free path to Safe Area exists* AND *Free path to Object exists* AND *Free path to Goal exists*, the problem will be solved. Another feasible combination is *Free path to Safe Area exists* AND *Agent Nearby* AND *Payed Task Available*. In the former case the robot will move the object, and in the later it will first earn money, and then pay another agent to move the object. The requirements of Theorem 1 is that the BT is non conflicting, see Definition 8, which means that the actions should achieve their postconditions, without violating previously achieved sub-goals, denoted ACC, see Definition 7. These are listed in Table III, and correspond to e.g., staying in the safe area, and in particular moving carefully not to drop the object, when holding it.

Now imagine that the robot has the ability to *create* a free path by removing obstacles, or opening doors. Extending the BT in the standard way will however create a conflict, that we *will* notice when computing the ACCs, see Table IV. The problem is that the robot needs an empty gripper to open the door, but dropping the object violates progress towards placing the object at the goal. The solution is to note that *Free Path to Goal Exists* is actually a precondition to *Place Object at Goal*, of higher priority than *Object in Gripper*. Thus, instead of the dashed option in Figure 10, the dotted one should be preferred, that does not result in such conflicts, see Table IV.

Actions	Objectives	ACC
Drop Object in Gripper (dashed)	Empty Gripper	In Safe Area AND Object in Gripper
Drop Object in Gripper (dotted)	Empty Gripper	In Safe Area

TABLE IV: The new ACCs of the BT in Figure 10. Note how the first (dashed) placement of the new subtree creates a conflict (trying to *Drop Object in Gripper* while keeping *Object in Gripper* as a constraint), while the second (dotted) placement has no such problems.

VI. Conclusions

In this letter, we have shown how to provide convergence guarantees for backchained BTs. Such BTs are more complex than the linear designs studied in earlier work, where each action satisfies preconditions of higher priority ones. Here we allow separate preconditions that are satisfied with different actions, as well as separate actions satisfying the same conditions in different ways. The approach builds upon identifying individual ACCs, global constraints that each action needs to satisfy, using e.g., path planning or CBFs, and we show how an And-Or tree can be used to describe when the design is guaranteed to work.

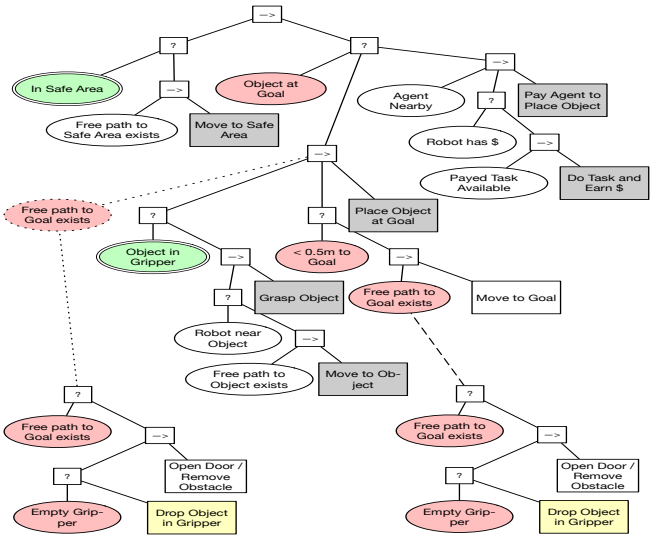


Fig. 10: Two alternative placements of a new subtree. The dashed creates a conflict while the dotted does not.

References

[1] D. Isla, "Handling Complexity in the Halo 2 AI," in *Game Developers Conference*, 2005.

[2] P. Ögren, "Increasing Modularity of UAV Control Systems using Computer Game Behavior Trees," in *AIAA Guidance, Navigation, and Control Conference 2012; Minneapolis, MN; United States; 13 August 2012 through 16 August 2012*, 2012.

[3] M. Colledanchise and P. Ögren, "How Behavior Trees Modularize Hybrid Control Systems and Generalize Sequential Behavior Compositions, the Subsumption Architecture, and Decision Trees," *IEEE Transactions on Robotics*, 2017.

[4] F. Rovida, D. Wuthier, B. Grossmann, M. Fumagalli, and V. Krüger, "Motion Generators Combined with Behavior Trees: A Novel Approach to Skill Modelling," in *IEEE/RSJ International Conference on Intelligent Robots and Systems (IROS)*, 2018.

[5] F. Rovida, B. Grossmann, and V. Krüger, "Extended behavior trees for quick definition of flexible robotic tasks," in *2017 IEEE/RSJ International Conference on Intelligent Robots and Systems (IROS)*, Sept. 2017, pp. 6793–6800.

[6] 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)*, May 2015, pp. 6167–6174.

[7] C. Paxton, A. Hundt, F. Jonathan, K. Guerin, and G. D. Hager, "Costar: Instructing collaborative robots with behavior trees and vision," in *2017 IEEE international conference on robotics and automation (ICRA)*. IEEE, 2017, pp. 564–571.

[8] M. Colledanchise, A. Diogo, and P. Ögren, "Towards blended reactive planning and acting using behavior trees," in *IEEE International Conference on Robotics and Automation (ICRA)*, 2019.

[9] M. Iovino, E. Scukins, J. Styrud, P. Ögren, and C. Smith, "A survey of behavior trees in robotics and ai," *preprint arXiv:2005.05842*, 2020.

[10] M. Colledanchise and P. Ögren, "How Behavior Trees Generalize the Teleo-Reactive Paradigm and And-Or-Trees," in *IEEE International Conference on Intelligent Robots and Systems (IROS)*, 2016.

[11] C. Paxton, N. Ratliff, C. Eppner, and D. Fox, "Representing robot task plans as robust logical-dynamical systems," in *IEEE/RSJ International Conference on Intelligent Robots and Systems (IROS)*, 2019.

[12] N. Nilsson, "Teleo-reactive programs for agent control," *Journal of artificial intelligence research*, vol. 1, pp. 139–158, 1993.

[13] R. R. Burridge, A. A. Rizzi, and D. E. Koditschek, "Sequential composition of dynamically dexterous robot behaviors," *The International Journal of Robotics Research*, vol. 18, no. 6, pp. 534–555, 1999.

[14] M. Colledanchise and P. Ögren, "How behavior trees modularize hybrid control systems and generalize sequential behavior compositions, the subsumption architecture, and decision trees," *IEEE Transactions on robotics*, vol. 33, no. 2, pp. 372–389, 2017.

[15] Y. Emam, P. Glotfelter, and M. Egerstedt, "Robust barrier functions for a fully autonomous, remotely accessible swarm-robotics testbed," in *IEEE Conference on Decision and Control (CDC)*, 2019.