IEEE Robotics and Automation Letters (RAL) paper presented at the
2020 IEEE International Conference on Robotics and Automation (ICRA)
31 May - 31 August, 2020. Paris, France

A Framework for Formal Verification of Behavior Trees with Linear Temporal Logic

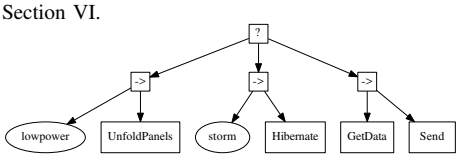
Oliver Biggar[1], Mohammad Zamani[1]

***Abstract*— Despite the current increasing popularity of Behavior Trees (BTs) in the robotics community, there does not currently exist a method to formally verify their correctness without compromising their most valuable traits: modularity, flexibility and reusability. In this paper we present a new mathematical framework in which we formally express Behavior Trees in Linear Temporal Logic (LTL). We show how this framework equivalently represents classical BTs. Then we utilize the proposed framework to construct an algorithm to verify that a given BT satisfies a given LTL specification. We prove that this algorithm is sound. Importantly, we prove that this method does not compromise the flexible design process of BTs, i.e. changes to subtrees can be verified separately and their combination can be assured to be correct. We present an example of the proposed algorithm in use.**

I. Introduction

Behavior Trees (BTs) are tree-based task-switching structures which are being used in game development, robotics and AI as a useful alternative to Finite State Machines (FSMs) [1]. Their increasing popularity in these fields—and particularly in industry—has been attributed to their modularity, flexibility and reusability [1]–[7]. BTs structure agent behaviors as trees, with leaf nodes representing Actions and Conditions and internal nodes controlling the order of execution (see Section III and Fig.1). This tree structure makes BTs ideal for large-scale system design, summarized by three main strengths: (1) "Changes in one part of the system [do] not affect other parts" [1] (*flexibility*); (2) atomic actions and subtrees present a common interface, so "individual behaviors can be reused in the context of a higher-level behavior" [8] (*reusability*); (3) "the designer [of one sub-behavior] does not need to know which sub-behavior will be performed next" [3] (*modularity*).

Fig. 1. An example Behavior Tree, constructed to control a Mars-Rover-inspired robot to obtain data and send it to Earth. This example is used in Section VI.



With these strengths in mind, a recent research goal [2] has been to adapt BTs to systems where more formal guarantees of the desired properties are required [9]–[12]. In this paper, we extend a standard model-checking problem [13] to BTs: how to verify formally whether a given BT satisfies a given Linear Temporal Logic (LTL) [14] specification. In theory, BTs are little different to FSMs (indeed one can show that they are equivalent to a subset of FSMs [3]) so methods for verification of FSMs could be adapted for BTs. Classical model-checking methods however, require constructing a Transition System [13] which requires low-level knowledge of behavior in every state, violating modularity and deconstructing the tree structure that gives BTs their flexibility and reusability. In this paper we show that verification can be done without compromising these properties. To formalize this, we seek an analysis framework for verifying the correctness of a BT that satisfies the following three principles (derived from the three strengths listed before).

1) (Reusability) Subtrees and individual leaf nodes should have a common interface.
2) (Modularity) The verifier should need only high-level details of behaviors.
3) (Flexibility) Refinements of one subtree should be verified by considering only that subtree.

The first two principles set out how a verification method should represent BTs, and the third describes how the verification should be able to be conducted. We believe all other existing approaches to this problem fail to satisfy one or all of these principles, as we shall discuss in Section II.

The main contribution of this paper is a new framework and method for verification of BTs that allows correctness to be checked without violating the three principles above. We propose an alternate mathematical interpretation and notation framework based on LTL and show that it equivalently represents classical BTs [3]. Moreover, we show that this framework allows us to reason about BT structures correctly using LTL [14]. We present each subtree as a mathematical object we call a 'behavior' (*reusability*) and use this to reason about compositions under operators inspired by the classical 'Sequence', 'Fallback' and 'Parallel' control flow nodes. In this framework we describe not only the required specifications, but also the properties of individual behaviors as LTL formulas, and recursively combine these to describe the BT as an LTL formula. In this way, we allow behaviors to be described by a high-level LTL formula that is not dependent on implementation (*modularity*) and in the process reduce the problem of verifying whether a given BT satisfies an LTL specification to the problem of language containment for LTL formulas. We present this as a method by which we can check whether a given BT controller satisfies an LTL specification. If it is not satisfied, the method generates a

[1]Both authors are with the Land Division, Defence Science and Technology Group, Australia. oliver.biggar1@dst.defence.gov.au mohammad.zamani@dst.defence.gov.au

counterexample of a legal BT execution which violates the specification, which provides useful feedback to the designer. We prove that this method is sound for *any* BT structure and *any* LTL formula representing the required specification. We prove also that this verification process is flexible; if we replace a node with a subtree, and verify that this subtree alone satisfies the properties of the original node, then the correctness of the *entire* tree is maintained (*flexibility*).

The result is a powerful automated analysis tool that enables BT designers to subject their designs to incremental, automated review, and so contributes to both the fields of formal verification and BTs in robotics. This allows BTs to be used in systems where correctness is important, without compromising their most desirable traits. We demonstrate this by showing the use of this method on a realistic example. Importantly, this also allows formal verification to be used in systems where flexibility, modularity and reusability are important.

The structure of this paper is as follows. In Section II, we discuss the related work. In Section III, we describe the classical description of BTs and briefly discuss LTL. In Section IV we present the proposed mathematical BT framework, and describe and prove how this framework equivalently represents classical BTs. We present our verification algorithm and related proofs of soundness, flexibility and complexity in Section V. We present an example of applying the presented method in Section VI.

II. Related Work

A variety of previous work has addressed different facets of formal verification problems for BTs. In [9], the authors present a mathematical approach intended to unify previous descriptions of BTs and provide a framework from which more rigorous reasoning about BTs can be done. Their approach allows proofs of important properties of BTs such as robustness to be constructed. Their framework similarly provided a common framework for subtrees (maintaining *reusability*) but intended for functional analysis based on state spaces while ours is intended for discrete analysis using logic. Verification against a given specification was not covered, and low-level details of Actions was required (reducing *modularity*). A different mathematical formalism based on process algebra is presented in [12], where the topic of verification is left for future research. In [10], a formalism based on description logic is presented as a method to interface BTs with their environment but verification of correct behavior is left for future work. In [11], the authors consider the problem of correct-by-construction synthesis of a BT based on an LTL specification. This work did not discuss verifying already-constructed BTs and the approach did not retain the desirable properties of BTs expressed above. It assumed complete atomic knowledge of the system and didn't provide a high-level description of the Actions of the agent (violating *modularity*). Also the structure of the LTL specification determined the structure of the BT, so subtrees were not treated identically (reducing *reusability*). In addition the BT was defined by internal states, making it less human-readable for a designer.

III. Behavior Trees and Linear Temporal Logic

A. Classical Description of Behavior Trees

A BT is a control architecture structured as a directed rooted tree (see Fig.1). A BT's execution begins at the root node, which sends signals (called 'ticks') to its children, if any. A node is executed when it receives ticks. Internal nodes tick their children when ticked, and are called *control flow nodes* and leaf nodes are called *execution nodes*. When ticked, any child node returns `RUNNING` if its execution is in progress, `SUCCESS` if it has achieved its goal, and `FAILURE` otherwise. Typically, there are four types of control flow nodes (Sequence, Fallback, Parallel, and Decorator) and two types of execution node (Action and Condition) [3]. Note that Fallback is sometimes called Selector. A Condition (drawn as an ellipse) checks some value, returning `SUCCESS` if true and `FAILURE` otherwise. An Action node (drawn as a rectangle) represents an action taken by the agent. Sequence nodes (drawn as a $\rightarrow$ symbol) tick their children from left to right. If any children return `FAILURE` or `RUNNING` that value is returned by Sequence, and it returns `SUCCESS` only if every child returns `SUCCESS`. Fallback (drawn as a **?**)is analogous to Sequence, except that it returns `FAILURE` only if every child returns `FAILURE`, and so on. The Parallel node (symbol $\rightrightarrows$) has a success threshold M, and ticks all of its N children; returning `SUCCESS` if M return `SUCCESS`, `FAILURE` if $N-M+1$ return `FAILURE` and `RUNNING` otherwise. The Decorator node returns a value based on some user-defined policy regarding the return values of its children. For a more detailed discussion, we refer the reader to [3].

B. Linear Temporal Logic

Linear Temporal Logic (LTL) [14] is an extension of propositional logic which includes qualifiers over linear paths in time. It has long been used as a tool in formal verification of programs and systems. Given a set AP of atomic propositions, the syntax of LTL is given by the following grammar [13]:

$$\varphi ::= \ p \mid \neg p \mid \varphi \vee \varphi \mid \bigcirc \varphi \mid \varphi \mathscr{U} \varphi$$

where $p \in AP$. The temporal operators are *next* $\bigcirc a$, which indicates a is true in the subsequent state, and *until* $a \mathscr{U} b$ indicating a is true until a state where b is true. We can derive other common operators from these as follows: *and* $\varphi \wedge \psi := \neg(\neg\varphi \vee \neg\psi)$, True $:= p \vee \neg p$, False $:= \neg$True, *implies* $\varphi \Rightarrow \psi := \neg\varphi \vee \psi$, *eventually* $\Diamond\varphi :=$ True$\mathscr{U}\varphi$ and *always* $\Box\varphi := \neg\Diamond\neg\varphi$. An LTL formula is satisfied by a sequence of truth assignments over AP, which can be considered an ω-word [13] over the alphabet given by 2^{AP}. If ϕ is an LTL formula, $L(\phi)$ is the ω-regular language containing all ω-words that satisfy ϕ. If ϕ and ψ are LTL formulas, then $\phi \models \psi$ if $L(\phi) \subseteq L(\psi)$ and we say ϕ *entails* ψ. This means in any sequence where ϕ holds, ψ also holds. We refer the reader to [13] for a more thorough discussion of LTL.

IV. The proposed framework

In this section we present the definitions that make up our framework and show how these relate to classical Behavior Trees. In our formulation, we generalize execution nodes and subtrees into a single mathematical object we call a behavior, and describe control flow nodes as operators over behaviors. We describe the execution of behaviors by LTL formulas. To avoid confusion in this section we will capitalize classical nodes such as Action and Sequence, but use lowercase for behaviors and the equivalent operators (sequence, fallback) defined in our framework.

We begin with some assumptions about classical BTs.

Assumption 4.1: We assume that the ticking of the classical tree is effectively instantaneous. In other words, the environment doesn't change between when we tick the root and when we tick any child node.

This is reasonable for most practical systems. With this assumption in place, we avoid considering the case where a BT takes the wrong action because the value of a variable changed while it was traversing the tree.

Assumption 4.2: We assume any subtree returns exactly one value every time it is ticked.

The process of checking the state of the world occurs only within ticks, so if an action begins executing (returning `RUNNING`) and then fails before the next tick, we assume that this will not be detected until the next tick. If the ticks are sufficiently frequent, both of these assumptions do not limit the reactiveness of BTs. This assumption also assumes that if a subtree is ticked at any point it must return either `RUNNING`, `SUCCESS` or `FAILURE`. Our framework is applicable to any BT that satisfies these two assumptions.

A. Behaviors and classical BTs

TABLE I
Behavior abstractions for the Mars Rover example

Behavior	Success	Failure	Guarantee
Get Data	$data$	False	$\Diamond data$
Send Data	False	$\neg data$	$\Diamond(sent \wedge \neg data)$
Unfold Panels	False	False	$charging \wedge (day \Rightarrow \Diamond \neg lowpower)$
Hibernate	False	False	$hibernating$

Table I contains some examples of behaviors used in Section VI to *model* (see Definition 4.7) the nodes in Fig.1. We will use them to explain the following definitions.

Definition 4.3: Given a finite set V of Boolean variables $v_1, v_2, \ldots, v_n$ that completely describe our system and its environment, we say that the *state space* of our problem, written $\mathscr{S}$, is a subset of $\{\text{True}, \text{False}\}^n$ containing all possible assignments of values to variables. We call an element s of the state space a *state*, and write s_{v_i} to mean the value of the variable v_i in s.

Definition 4.4: A *behavior* B is a 3-tuple

$$B = (B_{success},\ B_{failure},\ B_{guarantee})$$

where $B_{success}$ and $B_{failure}$ are Boolean formulas, $B_{guarantee}$ is an LTL formula and $B_{success} \wedge B_{failure} \equiv \text{False}$. Where it is not confusing, we may write $B_s,\ B_f,\ B_g$ as a shorthand. We shall denote the set of all behaviors over $\mathscr{S}$ by $\mathscr{B}$.

A behavior is designed to abstract the data given in a classical Action node, and in fact any classical BT (principle of *reusability*). Conceptually, we consider $B_{success}$ to be the proposition that would need to be satisfied for a classical BT to return `SUCCESS`. $B_{failure}$ works similarly. We refer to $B_{success}$ and $B_{failure}$ as the success and failure conditions, respectively. $B_{guarantee}$ describes 'what happens if this behavior is executed' in a sense we will make precise in Definition 4.7. To motivate this definition conceptually, consider the Action node GetData in Fig.1. We model this by the behavior **Get Data** in Table I. Suppose this node returns `SUCCESS` if it has data, which we represent by the proposition $data$. To represent this, we have $\text{succ}(\textbf{Get Data}) = data$, so GetData returns `SUCCESS` when $data$ is True. Similarly suppose GetData can always run, so doesn't ever return `FAILURE`. We specify this by $\text{fail}(\textbf{Get Data}) = \text{False}$. Now assume that we know if GetData executes then it will eventually have some data. We could model this by $\text{guar}(\textbf{Get Data}) = \Diamond data$. We will formalise these ideas with the following definitions.

Definition 4.5: Given a classical BT T and state space $\mathscr{S}$, a *physical run* of T is a sequence of states $s_0, s_1, \cdots \in \mathscr{S}$ constructed by taking the values of the variables in $\mathscr{S}$ at a sequence of time instants of a possible physical execution of a system controlled by T. We will write $\text{Ph}(T)$ for the set of all such physical runs.

Fundamentally, in any verification method we would like to be able to reason about the set of physical runs of the system, because these describe everything that can happen when the system is actually executed. For classical BTs, a physical run can be thought of as recording the values of all variables every time the root of the BT is ticked.

Definition 4.6: Given a behavior B and state space $\mathscr{S}$, a *logical run* of B is a sequence of states $s_0, s_1, \cdots \in \mathscr{S}$ such that $\Box(B_{success} \vee B_{failure} \vee B_{guarantee})$ holds. We will write $\text{Lg}(B)$ for the set of all possible logical runs of B.

A logical run is the analogue of a physical run for behaviors. We will use the following definitions to compare behaviors and classical BTs.

Definition 4.7: Given a classical BT T, a behavior B and state space $\mathscr{S}$, we say B *models* T if in any state $s \in \mathscr{S}$ in a physical run r of T,

- $B_{success}$ is true in s iff T returns `SUCCESS` in s.
- $B_{failure}$ is true in s iff T returns `FAILURE` in s.
- $B_{guarantee}$ holds in the subsequence r_s beginning at s if T returns `RUNNING` in s.

If the 'if' in the third criterion is an 'if and only if' we call B and T *equivalent*.

Later in Theorem 5.1 we prove that if a behavior models a classical BT, then every physical run of the BT is a logical run of the behavior. This allows us to reason about BTs using behaviors, which is useful because the set of physical runs is difficult to specify. Also, by modelling BTs as behaviors we avoid requiring complete low-level knowledge about the physical system, in accordance with the principle of *modularity*. Finding behaviors to model given Action

nodes only requires being able to express the conditions under which that node returns SUCCESS or FAILURE as propositional formulas of the variables in the state space. If that is possible, then setting True as the guarantee constructs a behavior which trivially models the node.

B. Condition nodes

Definition 4.8: A *condition* C is a behavior where $C_{success} \vee C_{failure}$ is a tautology.

A classical Condition node is a node that never returns RUNNING. Instead it tests a sensor or checks a variable, and immediately returns SUCCESS or FAILURE on that basis. This corresponds to our definition of a *condition*. Since a condition C has the property that $C_f \vee C_s$ is a tautology, its guarantee C_g is never relevant. Instead one of its success or failure conditions must at any point be satisfied. Hence we see that any condition C is equivalent to a classical Condition node which returns SUCCESS if and only if C_s. Note that since a condition is thought of as a proposition not a behavior, we may use the following shorthand. By 'the condition p' where p is a proposition, we mean the behavior given by $(p, \neg p, \text{True})$.

C. Negation

Definition 4.9: Negation, (written $\sim$), is a unary operator $\sim: \mathscr{B} \to \mathscr{B}$. Given a behavior B, $\sim B = (B_{failure},\ B_{success},\ B_{guarantee})$. In other words, $\sim B$ is the behavior obtained by switching B_s and B_f. This has a similar conceptual role to Boolean negation $\neg$ in propositional logic.

In our framework, negation is the direct counterpart of a classical Decorator node with a single child which swaps the output of its child from SUCCESS to FAILURE and vice versa.

D. Sequence and Fallback

Definition 4.10: Sequence, written $\to$, is an associative binary operator $\to : \mathscr{B} \times \mathscr{B} \to \mathscr{B}$. If A and B are behaviors then

$$
\begin{aligned}
(A \to B)_{success} &= A_{success} \wedge B_{success} \\
(A \to B)_{failure} &= A_{failure} \vee (A_{success} \wedge B_{failure}) \\
(A \to B)_g &= (\neg A_f \wedge \neg A_s \wedge A_g) \vee (A_s \wedge \neg B_f \wedge \neg B_s \wedge B_g)
\end{aligned}
$$

The idea behind this definition comes from the definition of the Sequence node in a classical BT, which returns SUCCESS when both its children return SUCCESS, FAILURE if either its first child returned FAILURE or its first returned SUCCESS and its second returned FAILURE. These correspond to the definitions of $(A \to B)_s$ and $(A \to B)_f$. If it returns RUNNING, either its first child returned RUNNING (so that child's guarantee holds) or its first returned SUCCESS and its second returned RUNNING. For example consider the behavior **Unfold Panels** from Table I and the condition *lowpower*. Using these rules, we construct $B := (lowpower \to \textbf{Unfold Panels})$,

$$
\begin{aligned}
B_s &= lowpower \wedge \text{False} = \text{False} \\
B_f &= \neg lowpower \vee (lowpower \wedge \text{False}) = \neg lowpower, \\
B_g &= (\neg lowpower \wedge lowpower \wedge \text{True}) \vee (lowpower \wedge \\
&\text{True} \wedge charging \wedge (day \Rightarrow \Diamond \neg lowpower)) \\
&= lowpower \wedge charging \wedge (day \Rightarrow \Diamond \neg lowpower).
\end{aligned}
$$

We note that $\to$ is sometimes used to mean material implication, which could cause confusion. Here, where used, we shall always write material implication as $\Rightarrow$.

Definition 4.11: Fallback, written ?, is an associative binary operator $? : \mathscr{B} \times \mathscr{B} \to \mathscr{B}$. If A and B are behaviors then $A?B = \sim(\sim A \to \sim B)$.

We can construct the fallback definition from 'first principles' as we did above for sequence (i.e. Fallback nodes returns FAILURE if both their children return FAILURE, and so on), but it turns out to be equivalent to this more succinct form. In a classical BT, the Sequence and Fallback nodes are allowed to have any non-zero number of children. We note firstly that a Sequence or Fallback node with a single child is equivalent to the BT consisting of just that child, so we need not consider this case. In our framework, we defined sequence and fallback to be binary operators, but because they are associative we can represent a Sequence node with three children A, B and C as $A \to B \to C \equiv A \to (B \to C) \equiv (A \to B) \to C$, and we can do likewise with Fallback nodes.

Lemma 4.12: Suppose S is a classical BT, with a Sequence node as the root, and L and R its two children. Suppose now we have behaviors B_L and B_R that model L and R respectively. Then $B_L \to B_R$ models S. This Lemma also holds for Fallback nodes.

E. Parallel

The previous binary operators composed two behaviors by selecting which of the two to apply based on some criteria. In some cases, it is useful to parallelize behaviors, which in a classical BT is done using the Parallel node. This node has a number of children N and a success threshold M. The node returns SUCCESS if at least M children return SUCCESS, returns FAILURE if at least $N - M + 1$ children return FAILURE and returns RUNNING otherwise. Here we introduce two new operators and prove that our two proposed parallel operators are sufficient to represent any Parallel node. *Note:* The two parallel operators are based on special cases of a single classical node. This use of the Parallel node is recommended in [15] because it is simple.

Definition 4.13: If A and B are behaviors, then the *parallel sequence* operator, written $\rightrightarrows$, is an associative commutative binary operator $\rightrightarrows : \mathscr{B} \times \mathscr{B} \to \mathscr{B}$, defined as follows:

$$
\begin{aligned}
(A \rightrightarrows B)_s &= A_{success} \wedge B_{success}, \\
(A \rightrightarrows B)_f &= A_{failure} \vee B_{failure}, \\
(A \rightrightarrows B)_g &= (\neg A_s \wedge B_s \wedge A_g) \vee (A_s \wedge \neg B_s \wedge B_g) \\
&\quad \vee (\neg A_s \wedge \neg B_s \wedge A_g \wedge B_g)
\end{aligned}
$$

Remark 4.14: It is straightforward to show that this operator is commutative ($A \rightrightarrows B \equiv B \rightrightarrows A$) since the definition is constructed "symmetrically" in A and B; it is composed from the commutative Boolean operators $\wedge$ and $\vee$.

Definition 4.15: If A and B are behaviors, then the *parallel fallback* operator, written $\overset{?}{?}$, is an associative commutative binary operator $\overset{?}{?} : \mathscr{B} \times \mathscr{B} \to \mathscr{B}$, which is defined in terms of $\rightrightarrows$ as $A \overset{?}{?} B = \sim(\sim A \rightrightarrows \sim B)$.

As associative and commutative operators, we can apply them to sets of inputs and say *the parallel sequence/fallback composition of* S to mean the result of composing all the elements of the set in any order.

When composing classical BTs in parallel we encounter problems of parallelizability. Classical BTs that control similar parts of a system can cause race conditions if they are executed naively in parallel. We therefore add the following definition:

Definition 4.16: Two classical BTs T_1 and T_2 are *commutative* if when run in parallel with T_1 having the higher priority in any conflict the outcome is identical to if T_2 had the higher priority.

In general, determining if classical BTs commute is nontrivial. See [4] for a discussion of using more complex parallelism in classical BTs.

Lemma 4.17: Suppose we have a Parallel node P with N commutative children $C = \{C_1, \ldots, C_N\}$ and success threshold M, and a set of behaviors $H = \{B_{C_1}, \ldots, B_{C_N}\}$ where $\forall i$, C_i is modeled by B_{C_i}. Then there exists a behavior B which is equivalent to P and can be composed from H using $\rightrightarrows$ and $\overset{?}{?}$.

Proof: (Idea) Replace P with a $\overset{?}{?}$ composition of $\rightrightarrows$ compositions of all M-element subsets of H. ∎

F. Other Decorators

The Decorator node can be used very freely to create new control flow nodes. Some example of other common uses include looping n times, or executing its children up to n times. We assert however, that these are merely implementation tools and any Decorator node can be replaced by a subtree consisting of other control-flow nodes with the possible addition of auxiliary variables. As a result, we will not consider these explicitly.

G. This framework equivalently represents classical BTs

Here we prove that combining behaviors using our operators captures the process of combining subtrees in a classical BT. Firstly we note the following result.

Lemma 4.18 (Duality): For all $A, B, C, D \in \mathscr{B}$:

$$\sim\sim A = A$$
$$\sim(A \to B) = \sim A \,?\, \sim B$$
$$\sim(C \rightrightarrows D) = \sim C \overset{?}{?} \sim D$$

As a result, we can rearrange any expression into a form where negation occurs only adjacent to atomic behaviors.

Theorem 4.19: Suppose we have a set A of Action and Condition nodes, and a set B of behaviors where each $b \in B$ has an $a \in A$ which it models. Let T be a classical BT over A. Let R be the behavior obtained by composing B in the structure given by T. Then R models T.

Proof: Follows from structural induction using lemmas 4.12 and 4.17 and their counterparts for ? and $\overset{?}{?}$. ∎

If we strengthen the hypothesis in the above theorem to assume each behavior is *equivalent to* its respective classical Action or Condition, then it can be similarly shown that the compositions are equivalent. This justifies that the operators here equivalently capture the structure of BTs.

V. Verification of Behavior Trees

Problem: Given a classical BT T composed of Action and Condition nodes $T_1, \ldots, T_n$ and an LTL formula φ representing the system specification, verify that all physical runs of T satisfy φ.

First we provide an informal sketch of the solution based on the following idea. Construct a formula that is true in all physical runs of the BT, and check if this formula entails the specification formula. To wit, find behaviors $B_1, \ldots, B_n$ which model $T_1, \ldots, T_n$. Combine these behaviors in the structure of T using the operators presented here to obtain behavior B. Then B models T, and we prove below in Theorem 5.1 that every physical run of T is a logical run of B ($\mathrm{Ph}(T) \subseteq \mathrm{Lg}(B)$). Then we simply check whether $\mathrm{Lg}(B) \subseteq L(\varphi)$, which is equivalent to checking that $\square(B_{success} \vee B_{failure} \vee B_{guarantee}) \models \varphi$. This is done by translating the first formula and the negation of the second to automata [13], and checking if the language of their product is empty. If this holds, then $\mathrm{Ph}(T) \subseteq \mathrm{Lg}(B) \subseteq L(\varphi)$ and so we know that every physical run of T satisfies φ. If this entailment does not hold, then any sequence in the language of the product automata is a run in $\mathrm{Lg}(B) \setminus L(\varphi)$. This is a counterexample of a valid logical run which does not satisfy φ. Using this we can refine our model and try again with different behaviors B_i. All the steps involving automata manipulation, including generating the counterexample, can be done using an off-the-shelf tool for LTL model checking such as [16].

Theorem 5.1: If T is a classical BT, and B a behavior that models T, then $\mathrm{Ph}(T) \subseteq \mathrm{Lg}(B)$.

Proof: Suppose for contradiction that $r \in \mathrm{Ph}(T) \setminus \mathrm{Lg}(B)$. Then $\square(B_s \vee B_f \vee B_g)$ does not hold in r, so $\exists i \in \mathbb{N}$ such that $B_s \vee B_f \vee B_g$ does not hold in the subsequence starting at r_i. As r is a physical run, at each instant r_i T has a return value (Assumption 4.2). However, B models T, so if T returns `SUCCESS` then B_s holds; if T returns `FAILURE` then B_f holds, and if T returns `RUNNING` then B_g holds in the subsequence beginning in r_i. However then $B_s \vee B_f \vee B_g$ holds in the subsequence beginning in r_i, which is a contradiction. ∎

Corollary 1.1 (Verification works): Let T be a classical BT composed of the Action and Condition nodes $T_1, \ldots, T_n$, and let φ be an LTL formula. Suppose $B_1, \ldots, B_n$ are behaviors where B_i models T_i. Let B be the behavior given by composing the B_i in the structure of T. If $\mathrm{Lg}(B) \subseteq L(\varphi)$ then $\mathrm{Ph}(T) \subseteq L(\varphi)$.

Proof: By Theorem 4.19, B models T. Then by Theorem 5.1 $\mathrm{Ph}(T) \subseteq \mathrm{Lg}(B)$ which implies $\mathrm{Ph}(T) \subseteq L(\varphi)$. ■

Remark 5.2: The above corollary states that this method is sound, i.e. whenever the method gives a positive result then it is definitely the case that every physical run of the system will obey the specification. However there are two points we should consider here, which are limitations of any system based on modelling. The first is that this only holds with the assumption that the individual behaviors did indeed model the individual Actions. Of course if some Actions acted in ways the modelling behavior did not capture then the overall tree may not behave as expected. The second point is that, even if $\mathrm{Ph}(T) \subseteq L(\varphi)$, this method may still give a negative result if $\mathrm{Lg}(B) \not\subseteq L(\varphi)$. In this case, the generated counterexample can be used to refine the behaviors.

A. Multiple agents and adversarial environments

In the previous section, we considered verifying a classical BT by constructing a behavior which modelled it and verifying that all logical runs of this behavior satisfy a given specification. In general though, considering the set of *all* possible runs is too strict a requirement, because some runs can be considered impossible by imposing requirements on the *environment*. In this section we will discuss one way to do this. The title of this section refers to multiple agents, because our method will involve introducing the environment as a second system acting in the state space, and so the method is generalizable to situations involving multiple agents. In the subsequent discussion we will mostly refer to the second agent as 'the environment', but the results hold even if we consider this to simply be another agent.

Problem: Given classical BTs T and E in a common state space $\mathscr{S}$ and an LTL specification φ, verify that all physical runs of the combined system satisfy φ.

Solution: Find behaviors T_B and E_B modelling T and E. Recall by Theorem 4.19 that if T and E are composed of multiple nodes, T_B and E_B can be constructed by composing behaviors modelling those individual nodes. We prove below that every physical run of the combined system is a logical run of both T_B and E_B. Recall that since a physical run is a sequence of truth values at specific time intervals, a physical run of the combined system is a recording of the values of variables of the system in a world containing both BTs. Then, check whether $\mathrm{Lg}(T_B) \cap \mathrm{Lg}(E_B) \subseteq L(\varphi)$ in the same way as in the previous solution.

Theorem 5.3: Let T and E be classical BTs, and let T_B and E_B be behaviors which model them. Then every physical run of the combined system is a logical run of both T_B and E_B.

Proof: This is an immediate corollary of Theorem 5.1, as any physical run of the combined system is a physical run of each system individually, and so is in $\mathrm{Lg}(T_B) \cap \mathrm{Lg}(E_B)$. ■

Remark 5.4: At first glance it is not obvious why intersecting the sets of logical runs is the right approach for multiple-agent systems. It may seem like more complicated behavior should be possible with multiple agents compared to a single agent. However, recall that for any behavior B the set $\mathrm{Lg}(B)$ already contains *all* possible logical runs, where all variables can change freely so long as $\Box(B_s \vee B_f \vee B_g)$ holds. Hence when modelling a behavior guarantee it is important to recall that the formula must hold in all circumstances, with any or no constraints on the environment behavior.

B. Maintaining flexibility in verification

Next we show how the proposed verification method allows changes in the tree to be checked locally (principle of *flexibility*). Suppose in modifying a classical BT that was known to satisfy a certain specification, an Action node (or any subtree) in the tree is replaced by a new subtree. We show how, given behaviors modelling all subtrees, ensuring that the modified tree is correct can be done by only checking the relationship between the Action node and the new subtree.This is described formally in Theorem 5.7. In real-world systems it is often useful to construct different aspects of a BT in pieces, or to begin with a simple core structure which is gradually refined. This theorem shows that this is possible with our approach, while correctness is maintained. The following definitions are used in Theorem 5.7.

Definition 5.5: If A and B are behaviors, we say that A *refines* B if $A_s \equiv B_s$, $A_f \equiv B_f$ and $\mathrm{Lg}(A) \subseteq \mathrm{Lg}(B)$. If also $A_g \models B_g$ we say A *strongly refines* B.

Definition 5.6: If K is a subtree in a BT T, and K_B, T_B are behaviors modelling them, then the *preconditions of* K, written pre_K are defined recursively by $\mathrm{pre}_K(K) = \mathrm{True}$, $\mathrm{pre}_K(Q \to A) = \mathrm{pre}_K(Q\,?A) = \mathrm{pre}_K(Q)$, $\mathrm{pre}_K(A \to Q) = \mathrm{pre}_K(A \rightrightarrows Q) = A_s \wedge \mathrm{pre}_K(Q)$, $\mathrm{pre}_K(A\,?Q) = \mathrm{pre}_K(A \,⁇ Q) = A_f \wedge \mathrm{pre}_K(Q)$, where Q is the behavior containing K_B and A is any behavior.

Theorem 5.7 (Verification is flexible): Let T be a BT and K a subtree of T. Let K' be another BT, and let T' be the BT obtained by replacing K with K' in T. Let T_B, K_B, T'_B, K'_B be behaviors modelling T, K, T', K' respectively, and suppose $\mathrm{Lg}(T_B) \subseteq L(\varphi)$ where φ is an LTL specification. Then if K'_B strongly refines K_B, then $\mathrm{Lg}(T'_B) \subseteq L(\varphi)$. If K'_B refines K_B and $\Diamond\Box\mathrm{pre}_K$, then $\mathrm{Lg}(T'_B) \subseteq L(\Diamond\varphi)$.

Proof: First note that for any run $r \in \mathrm{Lg}(B)$, for some behavior B, $r_t \in \mathrm{Lg}(B)$ for any tail r_t. Let $r \in \mathrm{Lg}(T'_B) \cap L(\Diamond\Box\mathrm{pre}_K)$. Then $\exists i \in \mathbb{N}$ such that the tail $r_i \in \mathrm{Lg}(T'_B) \cap L(\Box\mathrm{pre}_K)$. It is straightforward to show that for any behavior A, $L(\Box\mathrm{pre}_K) \cap \mathrm{Lg}(K'_B \to A) \subseteq \mathrm{Lg}(K_B \to A)$ (and likewise for fallback). We now show $L(\Box\mathrm{pre}_K) \cap \mathrm{Lg}(A \to K'_B) \subseteq \mathrm{Lg}(A \to K_B)$. Firstly, $\mathrm{pre}_K \models A_s \implies \Box\mathrm{pre}_K \models \Box A_s$. Then $\Box A_s \wedge \Box((A \to K'_B)_s \vee (A \to K'_B)_f \vee (A \to K'_B)_g) \equiv \Box A_s \wedge \Box((A \to K_B)_s \vee (A \to K_B)_f \vee (A \to K'_B)_g)$. Expanding this definition and simplifying gives $\Box((A_s \wedge K_{B_s}) \vee (A_s \wedge K_{B_f}) \vee (A_s \wedge \neg K_{B_s} \wedge \neg K_{B_f} \wedge K'_g)) = \chi$. Now $\chi \models \Box(K_{B_s} \vee K_{B_f} \vee K'_{B_g}) \models \Box(K_{B_s} \vee K_{B_f} \vee K_{B_g})$ by refinement, so $\chi \models \chi \wedge \Box(K_{B_s} \vee K_{B_f} \vee K_{B_g}) \models \Box((A \to K_B)_s \vee (A \to K_B)_f \vee (A \to K_B)_g)$ as required. If K'_B strongly refines K_B this argument does not require the extra condition $\Box A_s$. The remaining cases $A\,?K'_B$, $A \rightrightarrows K'_B \ldots$ are similar. By induction we conclude that

$\mathrm{Lg}(T'_B) \subseteq \mathrm{Lg}(T_B) \subseteq L(\varphi)$, and $r_i \in L(\varphi) \implies r \in L(\Diamond\varphi)$. ■

The additional fairness requirement of $\Diamond\Box \mathrm{pre}_K$ ensures that the modified subtree is eventually able to make progress without the environment repeatedly causing the ticks to move elsewhere in the tree. This theorem extends easily to the case where an environment behavior E is also specified explicitly. We provide an example of this method of separately verifying subtrees in Section VI.

C. Complexity Analysis

The key step of this method is a check that $\mathrm{Lg}(B) \subseteq L(\varphi)$ for some behavior B and LTL specification φ. This involves translating both to automata, and verifying that the language of the second contains the first. The time complexity of this transformation is known to be exponential in the length of both formulas [13]. We are therefore interested here in the relationship between the number of behaviors n with a given average length of their success conditions, failure conditions and guarantee, and the length of the guarantee of a behavior that is composed of those n behaviors.

Theorem 5.8: If we have n behaviors with average length x composed (without using parallel operators) in a behavior T, then $\mathrm{guar}(T)$ has length which is $O((xn)^2)$. Compositions of n parallel operators have guarantee length which is $O(2^{nx})$. Though we omit this proof, it essentially comes down to observing that as n behaviors are combined with sequence, the total length grows proportional to the sum of integers $n(n+1)/2$. The large upper bound for parallel operators is not so problematic, because of their comparatively rare use [4]. A behavior tree is not an ideal tool for circumstances where it is necessary to run large number of tasks in parallel.

VI. Example

In this section, we work through an illustrative example of our method. The example is inspired by the NASA Space Robotics Challenge [17]. This example is intended to provide a compact example of the real-world considerations of controlling a robot in a dynamic environment; the robot must be able to complete tasks repeatedly, some of which require a complex chain of actions and reasoning to complete, while responding appropriately to unforeseen circumstances caused by its environment.

Implementing the method in this paper is relatively straightforward with access to off-the-shelf model-checking software which can check for LTL entailment. To demonstrate our results, we created a basic implementation of this algorithm using the model-checking library SPOT [16]. The following example uses our implementation. The source code and a Jupyter Notebook containing an interactive form of this example are provided with this paper.

In the NASA Space Robotics Challenge, robots were required to execute complex tasks including repairing a solar array semi-autonomously (due to significant latency representing delay from Earth to Mars). In our example, we shall consider a Mars-rover-like robot exploring the surface of a planet. The robot is intended to collect data from rock samples and send it to Earth. The robot has a solar panel which it can unfold to charge during the day. When a storm comes, the robot must enter a hibernation state in order to protect itself from being damaged. We assume that the robot does not use power while hibernating. The robot should never run out of power and it should never be damaged. Suppose now that the robot is controlled by the classical BT in Fig. 1, which we shall call T. We wish to know if this tree will meet our requirements. To do this, we construct the behaviors shown in Table I, where each behavior (written in bold) is assumed to model the Action node of the same name. Note that in accordance with the principle of *modularity* we only need a high-level understanding of these Actions to construct these behaviors. In the table we have used italicised words to represent atomic propositions. Suppose the environment is modelled by the following behavior E:

$$\begin{aligned}\mathrm{guar}(E) = (&\Diamond\Box \mathit{lowpower} \vee \Diamond\Box\neg \mathit{lowpower}) \wedge \Diamond\Box\neg \mathit{storm} \\ &\wedge(\mathit{dead} \Rightarrow (\mathit{lowpower} \wedge \neg \mathit{charging} \wedge \neg \mathit{hibernating})) \\ &\wedge(\mathit{damaged} \Rightarrow (\mathit{storm} \wedge \neg \mathit{hibernating})) \wedge \Diamond \mathit{day}\end{aligned}$$

and $E_s = E_f =$ False. The requirement $\Diamond\Box \mathit{lowpower} \vee \Diamond\Box\neg \mathit{lowpower}$ states that the value of the variable *lowpower* can only change finitely many times. This means that as long as the robot repeatedly charges, eventually *lowpower* is False forever so the robot is not always charging. We also require *storm* is eventually False forever, so the robot is not always hibernating. The next two lines model the methods by which the environment can legally affect the robot. The first states that the robot can become *dead* only if it has *lowpower* and is neither charging nor hibernating. The second states that the robot can be *damaged* only if there is a storm and it is not hibernating. The last requirement states that *day* is repeatedly True. It is important to recall that by Definition 4.6, all terms are implicitly quantified by the *always* operator ($\Box$) so they hold in every state. The system specification can be expressed as $\varphi = \Box\neg \mathit{dead} \wedge \Box\neg \mathit{damaged} \wedge \Diamond \mathit{sent}$ requiring that the robot is never *dead* or *damaged* and eventually data is sent.

We can now apply the method given in this paper (see Section V) to see if the classical BT T in Fig. 1 meets the specification φ. By using the operators presented in this paper, we obtain a behavior B which models T. We find that this tree *does not* satisfy the requirement φ with the environment given. We produce a counterexample of a logical run where, in the initial state there is *storm*, but the robot chooses to UnfoldPanels and *damaged* is True. With this information, we as the designer choose to swap the order of the $\mathit{lowpower} \rightarrow$ UnfoldPanels and $\mathit{storm} \rightarrow$ Hibernate subtrees, noting that the robot cannot run out of power while hibernating. After making this change, we verify the new tree and determine that it does now satisfy φ. While this example was simple, logical errors of similar type can often occur when designing complex behaviors. This highlights the importance of employing a verification tool in order to progressively verify a design as it evolves.

A key aspect of this framework is that it is *flexible*, so

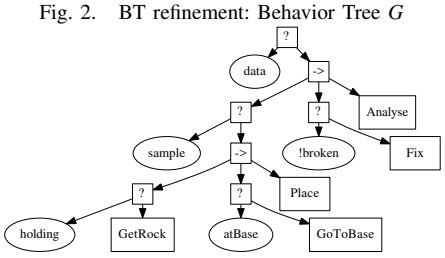
changes can be made to subtrees without compromising the correctness of the entire tree. The following part of the example illustrates this. Let us now modify this example by making the GetData Action more specific. Assume that the robot must actively find rock samples and bring them to a base station to be analysed. Occasionally a component on the base station will break, which the robot must fix before it can analyze any new rocks. We present new behaviors to model these Actions in Table II. Note particularly the

TABLE II
MORE BEHAVIORS

Behavior	Success	Failure	Guarantee
Go To Base	$atBase$	False	$\Diamond atBase \wedge ((holding \wedge \bigcirc holdng) \vee (\neg holding \wedge \bigcirc \neg holding))$
Place	False	$\neg holding$	$atBase \Rightarrow \Diamond(\neg holding \wedge sample)$
Get Rock	$holding$	False	$\Diamond holding$
Analyse	False	$\neg sample$	$\neg broken \Rightarrow \Diamond data$
Fix	$\neg broken$	False	$\Diamond \neg broken$

expression $(holding \wedge \bigcirc holding) \vee (\neg holding \wedge \bigcirc \neg holding)$ in the definition of **Go To Base**. This expresses that the value of $holding$ should not change in any state where we are returning to the base station, i.e. the robot does not drop the rock and does not pick up rocks while travelling. Let G be the BT we propose to replace GetData, and let G_B be the behavior with which we model it. We want G_B to refine the **Get Data** behavior in Table I above, so its success conditions should be $data$, its failure conditions False and its guarantee should satisfy $\Diamond data$. G is shown in Fig. 2. We

Fig. 2. BT refinement: Behavior Tree G



can construct the behavior G_B by composing the behaviors in Table II in the structure of G using the operators in this paper. They confirm that $G_{B_s} = data$ and $G_{B_f} = \text{False}$. Now note that G_B (trivially) strongly refines the behavior $idle := (data, \text{False}, \text{True})$. If we replace GetData in T by $idle$, we can use the method above to find that all physical runs of the resultant tree satisfy $\Box\neg(dead \vee damaged) \wedge \Diamond\Box\neg(lowpower \vee storm)$, and so $\text{Lg}(T'_B) \subseteq L(\Box\neg(dead \vee damaged))$ by Thm 5.7. Now as $\text{pre}_K = \neg lowpower \wedge \neg storm$, we know $\Diamond\Box\text{pre}_K$ also holds, and so we need only check that G_B refines GetData. We update our environment model to $E' = (\text{False}, \text{False}, \Diamond\Box broken \vee \Diamond\Box\neg broken)$, where we require that the base station can only be broken finitely many times (as before with the variable $lowpower$, this constraint is necessary to ensure the robot can eventually send data) and omit all variables in E that are not otherwise used in G_B, because varying them cannot affect the outcome. Our implementation verified that $\text{Lg}(G_B) \cap \text{Lg}(E') \subseteq \text{Lg}(GetData)$, and so G_B refines **Get Data**. We know then by Theorem 5.7 that $\text{Lg}(T'_B) \subseteq L(\Diamond\varphi)$, and so $\Diamond sent$ holds. But we showed already that $\Box\neg dead \wedge \Box\neg damaged$ holds, and so the original BT (Fig. 1) after replacing GetData by G (Fig. 2) still satisfies φ.

VII. FUTURE WORK

From the framework developed here, directions for future work include automating the construction of BTs to meet a specification, characterizing more completely the sufficient conditions for modifying subtrees correctly or using a different temporal logic to make modeling BTs more intuitive.

VIII. ACKNOWLEDGEMENTS

Our thanks to Michele Colledanchise and the anonymous reviewers for providing valuable feedback and suggestions. We would also like to thank the Australian Defence Science and Technology Group for their funding of this research.

REFERENCES

[1] A. Klöckner, "Behavior Trees for UAV Mission Management," in *Informatik angepasst an Mensch, Organisation und Umwelt*, vol. P-220, 2013, pp. 57–68.

[2] P. Ogren, "Increasing modularity of uav control systems using computer game behavior trees," in *Aiaa guidance, navigation, and control conference*, 2012, p. 4458.

[3] M. Colledanchise and P. Ögren, *Behavior Trees in Robotics and Al: An Introduction*. CRC Press, 2018.

[4] M. Colledanchise and L. Natale, "Improving the parallel execution of behavior trees," in *Proceedings of Int. Conf. on Intelligent Robots and Systems*. IEEE/RSJ, 2018, pp. 7103–7110.

[5] M. Colledanchise, R. N. Parasuraman, and P. Ogren, "Learning of behavior trees for autonomous agents," *IEEE Trans. on Games*, 2018.

[6] E. Coronado, F. Mastrogiovanni, and G. Venture, "Development of intelligent behaviors for social robots via user-friendly and modular programming tools," in *2018 IEEE Workshop on Advanced Robotics and its Social Impacts (ARSO)*. IEEE, 2018, pp. 62–68.

[7] S. Pütz, J. S. Simón, and J. Hertzberg, "Move base flex," in *Int. Conf. on Intelligent Robots and Systems*. IEEE, 2018, pp. 3416–3421.

[8] J. A. Bagnell, F. Cavalcanti, L. Cui, T. Galluzzo, M. Hebert, M. Kazemi, M. Klingensmith, J. Libby, T. Y. Liu, N. Pollard, *et al.*, "An integrated system for autonomous robotics manipulation," in *IEEE/RSJ Int. Conf. on Intelligent Robots and Systems*. IEEE, 2012, pp. 2955–2962.

[9] A. Marzinotto, M. Colledanchise, C. Smith, and P. Ögren, "Towards a unified behavior trees framework for robot control," in *Proceedings of Int. Conf. on Robotics and Automation*. IEEE, 2014, pp. 5420–5427.

[10] A. Klöckner, "Interfacing behavior trees with the world using description logic," in *Proceedings of the Guidance, Navigation and Control Conference*. Boston, MA: AIAA, 2013.

[11] M. Colledanchise, R. M. Murray, and P. Ögren, "Synthesis of correct-by-construction behavior trees," in *Proceedings of the Int. Conf. on Intelligent Robots and Systems*. IEEE/RSJ, 2017, pp. 6039–6046.

[12] R. J. Colvin and I. J. Hayes, "A semantics for behavior trees using csp with specification commands," *Science of Computer Programming*, vol. 76, no. 10, pp. 891–914, 2011.

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

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

[15] A. Champandard, "Enabling concurrency in your behavior hierarchy," *AIGameDev. com*, 2007.

[16] A. Duret-Lutz, A. Lewkowicz, A. Fauchille, T. Michaud, E. Renault, and L. Xu, "Spot 2.0 — a framework for LTL and ω-automata manipulation," in *Proceedings of the 14th Int. Symp. on Automated Technology for Verification and Analysis (ATVA'16)*, ser. Lecture Notes in Computer Science, vol. 9938. Springer, Oct. 2016, pp. 122–129.

[17] Nasa's centennial challenges: Space robotics challenge. [Online]. Available: http://www.nasa.gov/directorates/spacetech/centennial_challenges/space_robotics/