Planning for Automated Testing of Implicit Constraints in Behavior Trees

Uwe Köckemann[1], **Daniele Calisi**[2], **Guglielmo Gemignani**[2], **Jennifer Renoux**[1], **and Alessandro Saffiotti**[1]

[1]Center for Applied Autonomous Sensor Systems
Örebro University, Sweden.
[2]Magazino GmbH, Munich, Germany.
{uwe.kockemann, jennifer.renoux, alessandro.saffiotti}@oru.se, {calisi, gemignani}@magazino.eu

Abstract

Behavior Trees (BTs) are a formalism increasingly used to control the execution of robotic systems. The strength of BTs resides in their compact, hierarchical and transparent representation. However, when used in practical applications transparency is often hindered by the introduction of implicit runtime relations between nodes, e.g., because of data dependencies or hardware-related ordering constraints. Manually verifying the correctness of a BT with respect to these hidden relations is a tedious and error-prone task. This paper presents a modular planning-based approach for automatically testing BTs offline at design time, to identify possible executions that may violate given data and ordering constraints and to exhibit traces of these executions to help debugging. Our approach supports both basic and advanced BT node types, e.g., supporting parallel behaviors, and can be extended with other node types as needed. We evaluate our approach on BTs used in a commercially deployed robotics system and on a large set of randomly generated trees showing that our approach scales to realistic sizes of more than 3000 nodes.

Introduction

Behavior Trees (BTs) have been used for almost 20 years to model complex agents in the computer game industry (Isla 2005; Nicolau et al. 2017; Sekhavat 2017). BTs offer an alternative way to encode complex behaviors of non-playing characters and provide more readability and better support for *modular* design compared to the widely used Finite State Machines. Because of these characteristics, the field of AI and Robotics has also started to give attention to BTs (Colledanchise and Ögren 2017; Bojic et al. 2011; Ögren 2012; Marzinotto et al. 2014; Iovino et al. 2022), and BTs are today increasingly used as a tool to encode robot control strategies both in academia and industry.

When used in practical robotic systems, BTs typically need to be enriched with additional capabilities, and several extensions to their initial formulation have been presented in order to accommodate, e.g., sensing (Yang et al. 2021), multi-agent coordination (Agis, Gottifredi, and García 2020), or passing data between nodes (Gemignani 2021). While these extensions increase the representation power of BTs, they also make their design more error prone. The latter capability, in particular, introduces implicit constraints among nodes in the BT which are not directly visible in the tree's topology. For instance, suppose that a node A in the tree runs a task that computes a pose estimate, and a node B in another part of the tree uses that pose estimate. A BT designer must make sure that any execution of the tree will run node A before node B. In another example, a given robotic system may require that a sensor initialization task, done in node A, is always performed before running any task, say in node B, that uses that sensor.

Figure 4 later in the paper shows a fragment of a BT actually used at Magazino where such dependencies are indicated. When dealing with trees of notable size, a BT designer might easily overlook possible execution sequences that violate implicit constraints. As a matter of fact, such inconsistencies are often present in robotic systems based on BTs, including commercially deployed ones, which may result in runtime errors leading to potentially significant productivity losses. This paper addresses this problem.

We present a planning-based approach that automatically tests data dependencies and ordering constraints on BTs offline, before these trees are deployed. The critical move is to translate each dependency into a planning problem, whose solution is a trace of a possible execution that violates that dependency if it exists. Since each planning problem focuses on a specific dependency, we can prune large portions of the BT before converting it into a planning problem. As a result, our approach can deal with large trees with more than 3000 nodes, such as those typically found in real deployed applications.

The use of automated planning for testing BTs brings about a number of important advantages. First, the generated plan precisely describes the sequence of events that reproduces a constraint violation, if any; this may give concrete hints to the BT designer on the origin of the problem and how to fix it. Second, we can support additional control flow nodes used by BT variations by simply adding new planning operators without modifying any algorithm. Finally, according to the empirical findings reported below, domain-independent planning algorithms perform very well on this problem both when an inconsistency is present and when the tree is entirely consistent.

In this paper, we consider a concrete use case based on a real-world application: the BTs used by Magazino GmbH

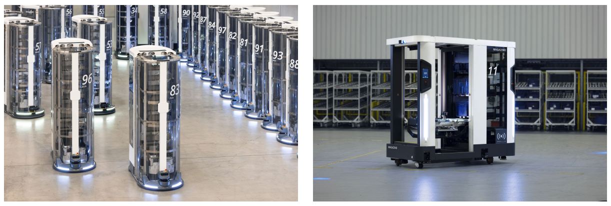
Figure 1: The TORU (left) and SOTO (right) robots.

to encode the behavior of its robots. Magazino GmbH is a Germany company that develops autonomous mobile pick-and-place robots for production lines and warehouses in e-commerce applications. Such robots, called TORU and SOTO, are shown in Figure 1. These robots can pick single items from shelves and operate as part of a swarm, working alongside humans. The behaviors of the robots are described via a modified version (Tenorth 2016) of the classical BTs formalism (Marzinotto et al. 2014). This version has a richer data passing mechanism (Gemignani 2021), that will be detailed in the background section, and is the main object of the automated testing described in this paper. We emphasize, however, that the approach described in this paper is not limited to the formulation of BTs used by Magazino: other control flow nodes and decorators can be modeled simply by defining the corresponding planning operators.

The remainder of this paper is organized as follows. First, we discuss related work. Then we introduce the necessary background on BTs and automated planning. Based on this, we introduce BT testing as a decision problem which is then turned into a planning problem. Then we briefly discuss how pruning is performed and how parallel nodes can be converted to non-parallel nodes for the purpose of testing data dependencies. Finally, we illustrate our approach on BTs used by Magazino, and we show a systematic evaluation on randomly generated BTs that supports the scalability of our approach.

Related Work

Despite the increasing popularity of BTs to describe the desired behavior of an intelligent autonomous system, there are still very few tools for formal validation of their correctness that can help BT designers (e.g., Biggar and Zamani (2020); Serbinowski and Johnson (2022)).

Conditional Behavior Trees (CBTs) (Giunchiglia et al. 2019) attach preconditions and effects (as used in classical planning) to execution nodes in behavior trees. This makes it possible to verify that conditions are guaranteed to be satisfied when reaching an execution node, e.g., by converting a conditional BT to a Boolean satisfiability problem (Giunchiglia et al. 2019; Colledanchise et al. 2021). This approach does not perform pruning on BTs beforehand and selector (fallback) control flow nodes require a quadratic number of symbols. This leads to the conversion to SAT requiring time quadratic in the size of the BT, whereas our approach converts a BT to a state in linear time. Moreover, as described in the Interpreting Solution section, our approach not only can detect invalid trees, but can also show examples of exact sequence of events that render them as such. Finally, conditional BTs can be modeled with an extension of the data dependencies we consider in this work. First, we represent each CBT precondition by a data requirement and each CBT effect by a data producer. Negative preconditions can be turned into goals asserting that data is unavailable. Negative effects can be modeled as nodes that make data unavailable (e.g., by turning off a sensor). Then, it is sufficient to test each CBT precondition symbol individually, as we do with data requirements in our approach.

Model checking for BTs has been discussed by Yatapanage, Winter, and Zafar (2010), who address the problem of "slicing" for BTs, a mechanism similar to the pruning approach we describe later. While our pruning approach directly operates on the BT, the slicing-based approach operates on a BT dependency graph. During slicing parts of a BT may become disconnected and have to be re-attached which is not the case of our pruning approach. The BT used to test the approach contains 125 nodes before slicing and was pruned down to 36, 116, and 73 nodes for three tested theories. While Yatapanage, Winter, and Zafar (2010) use linear temporal logic which allows for more complex properties to be expressed, our approach allows more aggressive pruning, while still enabling the verification of BT properties used in the deployment of complex robotics systems.

Background

Behavior Trees

A behavior tree B is a directed graph whose nodes are either the root node $\emptyset_B$, an execution node T (a leaf) or a control flow node F. Each BT has a single root node with a single child and no parents. All other nodes have a single parent. Execution nodes have no children, and control flow nodes have at least one child. A BT is executed by periodically "ticking" the root node with some frequency. When the root node is ticked, it will tick its sole child node. When an execution node is ticked for the first time, it will be executed and return its state. The state of each node is either *pending*, *running*, *success*, or *failure*. An execution node will return *running* until it is completed and then return either *success* or *failure* based on its outcome. The *pending* state is never actually returned but will be useful in later definitions. Control flow nodes behave depending on their type. Every formulation of BTs define two common types of control flow nodes: *sequence* ($\rightarrow$) and *selector* ($?$). A sequence node $\rightarrow$ activates its children in order. When any child returns *running* or *failure* it will return the child's state. On *success* the next child will be activated. When all children have succeeded, the sequence also returns *success*. A selector node $?$ activates its children in order. When any child returns *running* or *success* it will return the child's state. On *failure* the next child will be activated. When all children have failed, the selector also returns *failure*. We use T_B and C_B as the set of all execution and control flow nodes in BT B respectively. The state of a BT during execution is written as a map from each node to its current state $S_B = \{\forall_{n \in T_B \cup C_B} n \leftarrow s | s \in \mathit{pending}, \mathit{running}, \mathit{success}, \mathit{failure}\}$ and we use $S_B(n)$ for the current state of node n in state S_B. The set of all possible states reachable by a given tree is written S_B^*. For any node n we write its type as $type_n$. If n is an execution node, its type is $exec$, and if n is a control flow node, the type is a symbol representing the type of control flow node. For a control flow node n, we write $children(n)$ for the list of its children in execution order. $children(n)_i$ is the ith child of n.

Behavior Tree Extensions

Many different formulations of BTs exist. Such different formalisms are often extended or have hidden dependencies that may introduce constraints between execution nodes.

One possible extension of BTs can introduce new types of control flow nodes. These may support, for instance, loops or parallel execution in various ways. In the current paper, we additionally considered the following set of control flows:

- **ParallelAll**: a *ParallelAll* node activates all children in parallel and returns *success* only if all of them return *success*. If one of the children returns *failure*, the node returns *failure* immediately. Else, if some children return *running*, the node returns *running*.
- **ParallelSelector**: a *ParallelSelector* node activates all children in parallel and returns *success* as soon as one of them returns *success*. If all the children return *failure*, the node returns *failure*. Else, if some children return *running*, the node returns *running*.
- **OnFailure**: an *OnFailure* node activates its children in order. If the first child returns *success*, the node returns success immediately without activating the other children. If the first child returns *failure*, all the other children are activated one after the other, following the *sequence* rule. Independently if the the result of the remaining children is *success* or *failure*, the node returns *failure*. If some children return *running*, the node returns *running*. This composite is typically used to perform operations (e.g., clean up), upon failure of the first child.
- **Finally**: a *Finally* node activates the first child until it reaches the *success* or *failure* state. Then, it activates all the other children one after another until one returns *failure* or all return *success*. At this point, *Finally* returns the state returned by the first child. This composite is typically used to perform operations, no matter the return value of its first child.
- **Inverter**: an *Inverter* node inverts the return state of the child. If the child returns *success* or *failure*, the node returns *failure* or *success*, respectively. If the child returns *running*, the node also returns such state.

A second extension of the formalism allows execution nodes to produce and consume data. In our use-case, for example, a variable of a node can be defined as being input, output, or both, as well as having other characteristics: see Figure 4 for an example. More generally, there may also be other constraints on the order in which certain execution nodes should be executed that are not directly apparent in the BT.

We denote the fact that execution node T produces data x when in state $s \in S$ as

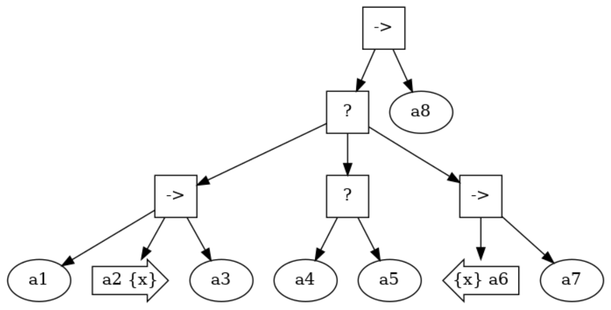

Figure 2: A randomly generated tree with depth 4. Nodes marked $\rightarrow$ are sequences and nodes marked ? are selectors. Arrow shaped nodes are data producers (right arrow) and consumers (left arrow). Producers indicate the produced data on the right of the node name and producers on the left.

$produces(T, x, S)$ or as $produces(T, x)$ if and only if $produces(T, x, \{running, success, failure\})$. S is the set of states in which execution node n produces x. The fact that execution node T requires data x is written as $requires(T, x)$.

We use P_B and R_B as the set of all *produces* and *requires* facts for tree B respectively. The set of all data produced or required in B is denoted D_B. Data required by any execution node has to be produced by a previous execution node in the tree. Producers and consumers can directly be extracted from many BT design tool or taken from CBT models with the method explained above.

If data x is available in a state S_B of a tree, we write

$$available(x, S_B) \equiv \exists_n produces(n, x, S) \in P_B \land S_B(n) \in S$$

Figure 2 shows a BT with one node ($a2$) producing data x (indicated by the right arrow node shape, and a single node ($a6$) requiring x. For this example (and our evaluation later on), we assume that data is produced as soon as a node enters the $running$ state.

While this notion is focused on data dependencies, we can use it also to represent generic ordering constraints between nodes in a tree. Consider, the following cases in which A, B, and C are execution nodes. *(A only after B)*: B produces x and A requires x. *(A only after B or C)*: B and C produce x and A requires x. *(A only after B and C)*: B produces x and C produces y. A requires x and y. (Where x and y are variables not used anywhere else in the tree.)

Planning

A classical task planning problem (Ghallab, Nau, and Traverso 2004) (S_0, G, O) consists of an initial state S_0, a goal state G and a set of operators O. Here we use *Simplified Action Structures (SAS+)* planning (Bäckström and Nebel 1995), where S_0 and G (as well as all preconditions and effects) are state-variable assignments. A single state-variable assignment of value v to variable x is written $x \leftarrow v$. State-variables are convenient for our domain since we often need to track a series of values representing the states when completing control flow nodes. An operator $o = (n_o, P_o, E_o)$ consists of a name n_o, set preconditions

P_o and effects E_o. Both, P_o and E_o are state-variable assignments. A goal or precondition G is satisfied in a state S if $G \subseteq S$. Applying an effect E to a state S replaces the value assignments of all state-variables in S with the ones assigned by E. A plan $\pi = a_1, \ldots, a_n$ is a sequence of operators that can be applied starting at the initial state S_0 leading to the state sequence $S_0, S_1, \ldots, S_n$. A plan is applicable if $\forall_i P_{a_i} \subseteq S_{i-1}$. A plan is a solution if it results in state S_n with $G \subseteq S_n$. In the evaluation presented later, we use a planner that implements a greedy forward state space search with the Causal Graph heuristic (Helmert 2006).

Behavior Tree Testing as a Decision Problem

Before jumping into the planning problem, it is worth formulating BT testing as the following general decision problem, which may be solved in different ways. Given a tree B, is there any data in $x \in D_B$ with a requirement $\mathit{requires}(n, x) \in R_B$ such that can we reach a state s in which n is running but x is not available? We refer to any tree B that satisfies Eqn. 1 as *invalid*, and as *valid* otherwise.

$$\begin{aligned} \exists_{x \in D_B, s \in S_B^*, n \in T_B} \text{ s.t.} \quad & \mathit{requires}(n, x) \in R_B \\ \wedge \quad & \neg \mathit{available}(x, s) \\ \wedge \quad & s(n) \neq \mathit{pending} \end{aligned} \tag{1}$$

Behavior Tree Testing as a Planning Problem

To formulate our decision problem as a planning problem, we need to define states, goals, and operators. The state includes the tree's structure, each node's current state S_B, and some additional variables for modeling more complex control flow nodes. The goal is to reach a node without satisfying a specific data requirement and thus follows almost directly from the decision problem in Eqn. 1. As shown later, we create a dedicated set of goals for each element in R_B. While this means that we have to solve multiple problems, it will allow heavy pruning of the BT which allows our approach to work well even on large instances. Operators cover state changes of nodes from $\mathit{pending}$ to $\mathit{running}$ and from $\mathit{running}$ to $\mathit{success}$ or $\mathit{failure}$. Data is made available by producer nodes. Finally, there is a set of operators for each type of control flow node that models its behavior. This leads to a very modular domain formulation that allows adding new types of control flow nodes easily.

State

Table 1 summarizes the construction of the initial state. Most of these variables and values capture the structure and current state of the tree. active is used to track nodes that can be ticked. $\mathit{available}$ is used to track which data is available. Other variables are used for bookkeeping in more complex operators: $\mathit{state}_{\mathit{init}}$ is used to track how nodes in a parallel control flow node are activated. $\mathit{state}_{\mathit{final}}$ is used to remember a state of an earlier child that may be returned later (e.g., for *OnFailure*). $\mathit{positive}$ is used for control flow nodes where the first child node is treated specially (e.g., *OnFailure*). Unlike in the tree state S_B, $\mathit{state}(n)$ is also used to track the index of a control flow node's current child or how many children are finished. This, together with the $\mathit{successor}(i)$ and

Variable	Value	Condition
$\mathit{state}(n)$	$\mathit{pending}$	
$\mathit{state}_{\mathit{init}}(n)$	$\mathit{pending}$	
$\mathit{state}_{\mathit{final}}(n)$	$\mathit{pending}$	
$\mathit{active}(n)$	false	$n \neq \emptyset_B$
$\mathit{active}(n)$	true	$n = \emptyset_B$
$\mathit{type}(n)$	t	$t = \mathit{type}_n$
$\mathit{child}(n, i)$	n_c	$n_c = \mathit{children}(n)_i$
$\mathit{end}(n)$	i	$i = \lvert \mathit{children}(n) \rvert$
$\mathit{produces}(n, x)$	true	$\mathit{produces}(n, x) \in P_B$
$\mathit{requires}(n, x)$	true	$\mathit{requires}(n, x) \in R_B$
$\mathit{successor}(x)$	$x + 1$	$x \in \{1 \ldots m_B\}$
$\mathit{positive}(x)$	$x = 1$	$x \in \{1 \ldots m_B\}$
$\mathit{available}(x)$	false	$x \in D_B$

Table 1: State variables and values used when constructing the initial state from a BT B. In all conditions, we assume $n \in T_B \cup C_B$.

$\mathit{child}(n, i)$ variables, is used to move on to the next child when a child returns its result that allows starting the next child. In the same way, $\mathit{end}(n)$ is used to mark the last child to decide the outcome of a control flow node.

The state's size depends on the number of nodes in the tree and the number of children of each control flow node. Let $m_B = \max_{n \in C_B}(\lvert \mathit{children}(n) \rvert)$ be the maximum number of children of any node in the tree. The size of the state can then be bound by $O(\lvert T_B \rvert + \lvert C_B \rvert m_b)$. If we credit each item $\mathit{child}(n, i) \leftarrow n_c$ in the list of children to the corresponding child n_c and consider that each node has at most one parent that will require such an entry, we can see that the size of the state is bound by $O(\lvert T_B \rvert + \lvert C_B \rvert)$ which is linear in the number of nodes in the tree. The state can be created from a tree by traversing the tree depth-first from the root node in execution order while visiting each node exactly once. We can see that all conditions in Table 1 can be directly read from each node. Control flow nodes require visiting all of their child nodes, but as before, we can distribute this as a constant effort across all children for an overall time complexity of $O(\lvert T_B \rvert + \lvert C_B \rvert)$.

Goal

Given how we construct the state, we can create goals almost directly from Eqn. 1. For each $\mathit{requires}(n, x) \in R_B$:

$$G_{(n,x)} = \{\mathit{available}(x) \leftarrow \mathit{false}, \mathit{state}(n) \leftarrow \mathit{running}\}$$

Any plan that satisfies this goal is an execution trace through the BT that would lead to data x not being available when it is required by node n. Thus it shows that B is invalid and how an invalid state can be achieved.

The definition above creates a goal for each entry in R_B. This leads to $\lvert R_B \rvert$ goals and as many individual planning problems to solve, each focused on testing the validity of B for a single requirement. As mentioned above and shown in our evaluation on randomized problems, these individual problems by themselves become relatively easy to solve since they allow us to prune large parts of the search tree with the method explained below.

Operators

As mentioned above, we have operators for execution nodes (that may produce data), and for each control flow node. For each control flow node, a set of operators typically covers the following cases: (1) If the node is active and pending, set the first child to active. (2) If a child has succeeded or failed, process the result accordingly by either returning a result and skip remaining child nodes or moving to next child node. (3) If the last child succeeds or fails, return the appropriate result.

Below we show the operators for the *sequence* control flow node[1]. These operators are expressed using the *AI Domain Definition Language (AIDDL)* (Köckemann 2020) a general-purpose modeling language that allows us to represent BTs and the SAS+ flavor of planning problems and makes the translation from BTs to states easier to implement. In the example below, the colon : is used for key-value pairs, and the prefix ? indicates a variable. Preconditions and effects are sets of key-value pairs representing state-variable assignments[2]. All example operators share the common condition that the node's type must be *sequence*. If a sequence node is active and pending (1), the *seq-init* operator allows setting the first child to activate and remember the currently active child in the state of the sequence node. If the currently active child succeeds (2), the *seq-advance* operator allows setting the next child to active while deactivating the finished one. If the currently active child fails (2), *seq-fail* can set the sequence to active and its status to failing. Finally, if the last child succeeds (3), *seq-succeed* can set the sequence to active and its status to success.

```
name:(seq-init ?n ?n_c)
preconditions:{
  (type ?n):sequence
  (state ?n):pending
  (active ?n):true
  (child ?n 0):?n_c }
effects:{
  (state ?n):0
  (active ?n):false
  (active ?n_c):true }
```

```
name:(seq-advance ?n ?i ?n_i ?j ?n_j)
preconditions:{
  (type ?n):sequence
  (state ?n):?i
  (child ?n ?i):?n_i
  (active ?n_i):true
  (state ?n_i):success
  (after ?i):?j
  (child ?n ?j):?n_j }
effects:{
  (active ?n_i):false
  (active ?n_j):true
  (state ?n):?j }
```

[1]Full set of operators are available at https://doi.org/10.5281/zenodo.7708806

[2]Equivalent operators could be represented in the Planning Domain Definition Language (PDDL) (Ghallab et al. 1998), but our use of state-variable assignments would make this representation more cumbersome.

```
name:(seq-fail ?n ?i ?n_i)
preconditions:{
  (type ?n):sequence
  (state ?n):?i
  (child ?n ?i):?n_i
  (active ?n_i):true
  (state ?n_i):failure }
effects:{
  (active ?n_i):false
  (active ?n):true
  (state ?n):failure }
```

```
name:(seq-succeed ?n ?i ?n_i ?j)
preconditions:{
  (type ?n):sequence
  (state ?n):?i
  (child ?n ?i):?n_i
  (active ?n_i):true
  (state ?n_i):success
  (after ?i):?j
  (end ?n):?j }
effects:{
  (active ?n_i):false
  (active ?n):true
  (state ?n):success }
```

Pruning Trees for Testing

Obviously, the size of trees grows exponentially with their depth (assuming a branching factor greater than one). This increase directly impacts the state's size since we need to keep track of the state of every node in the tree. This means that once we reach a certain size, trees become very difficult to test. In the previous section, we extracted a goal for each node that requires an input x. Multiple execution nodes may produce x, but only a single consumer exists per planning problem. For any sub-tree that does not contain the requiring node or any of the producer nodes, we are only impacted by the outcome of the entire tree not by how a combination of individual execution nodes in the sub-tree achieved the outcome. In Figure 2, for instance, the sub-tree containing $a4$ and $a5$ has no impact on the validity of the tree wrt. x. As a result, we can replace any sub-tree that does not contain a relevant node by a single execution node representing the outcome of said sub-tree. More precisely, given a set of relevant execution nodes $K \in T_B$, we traverse the tree depth-first and replace each sub-tree n for which

$$prune(n, K) = n \notin K \wedge \forall_{c \in children(n)} prune(c, K) \quad (2)$$

evaluates to $true$ with a new execution node. Obviously, the smaller the set of relevant execution nodes, the more sub-trees can be pruned. Assuming $requires(n, x) \in R_B$ we have the set of relevant execution nodes

$$K_{(n,x)} = \{n\} \cup \{n' | produces(n', x) \in P_B\}.$$

With this, we can prune a tree for each goal $G_{(n,x)}$, which leads to a significant reduction in tree (and consequentially state) size. Figure 3 shows the result of applying this pruning rule to the tree depicted in Figure 2.

Dealing with Parallel Control Flow

Parallel nodes create a broad variability on how a tree can be traversed. Therefore, it would be preferable to replace them with non-parallel nodes where possible.

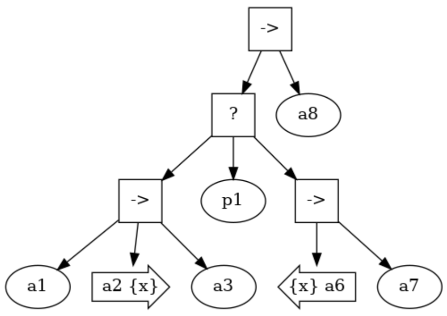

Figure 3: Pruned version of the tree depicted in Figure 2. The node *p1* represents a pruned sub-tree of the original tree.

It turns out that all types parallel nodes considered in this paper can be replaced for data dependencies testing by sequence or selector nodes in the following way.

- If no providing or requiring execution node appears in any child of a parallel node, it will be pruned as explained in the previous section
- If either a requiring execution node or a providing execution node appears in a child of a parallel all/selector node, replace this node with a sequence/selector node (the parallel order does not matter)
- If both a requiring and a providing execution node appear inside the same child of a parallel all/selector node, but not in any other child node, replace this node with a sequence/selector node (the order does not matter)
- If both a requiring and a providing execution node appear in two different children of a parallel all/selector node, replace the node with a sequence/selector and order its children such that the requiring execution nodes appear before the providing execution nodes (the order matters, but we can assume a non-parallel order that requires before it produces)

Note that in the last case the tree may still be consistent, e.g., if the data is provided before entering the parallel node.

Interpreting Solutions

One of the advantages of our approach is that finding a solution to the generated planning problem not only tells us that a tree is invalid, it also tells us exactly how to reproduce invalid behaviors. We can represent a plan as a sequence of action names that lead from the initial state to a state that satisfies the goal. Running our approach on the tree in Fig. 3 produces the following plan.

```
(0) (seq-init (sequence 1) (selector 2))
(1) (sel-init (selector 2) (sequence 3))
(2) (seq-init (sequence 3) a1)
(3) (run-0 a1)
(4) (fail-0 a1)
(5) (seq-fail (sequence 3) n0 a1)
(6) (sel-advance (selector 2) n0
                 (sequence 3) n1 p1)
(7) (run-0 p1)
(8) (fail-0 p1)
(9) (sel-advance (selector 2) n1 p1 n2
                 (sequence 8))
(10) (seq-init (sequence 8) a6)
(11) (run-0 a6)
```

The first three actions initialize control flow nodes (0-2). Then $a1$ runs and fails (3,4), leading to the sequence to fail (5) and the selector to advance (6), $p1$ runs and fails to move to the last node of the selector (7-9). The sequence containing $a6$ and $a7$ initializes (10) and finally $a6$ runs (11) without x being available. Note that in this example, $p1$ represents a sub-tree pruned from Fig. 2. This illustrates another advantage of pruning trees ahead of testing as it leads to shorter and more focused explanations for the reason the tree is invalid. In the original tree, actions 7 and 8 would have to be replaced by a plan that initializes the pruned selector an then fails $a4$ and $a5$. To make the above plan even easier to interpret, we can also filter control flow actions to produce a very compact plan that focuses on the state changes of execution nodes.

Evaluation

We evaluate our approach in two ways. First, we test it on actual sub-trees used at Magazino. Second, we evaluate how the approach scales on randomly generated BTs of increasing depth. BTs and planning problems are represented and solved in the *AIDDL Framework for Integrative AI* [3].

Evaluation on Magazino Trees

The automatic test framework has been run on a selection of BTs currently used by Magazino in production environments. It is not uncommon that these trees present problematic execution of nodes that leads to missing inputs to some nodes. An example is shown in Figure 4, where, depending on the result of the previous nodes, the producer node can be skipped. The problematic executions have been correctly found by the planner: e.g., if the node above the red framed (producer) node returns a FAILURE, the producer node is not executed and the subsequent consumers (blue arrow) can be executed without the required input.

It should be noted that Figure 4 is a fragment of an actual, much larger BT, used here for illustration purposes. The problematic execution above would not occur in the context of the full BT, because in this specific case the missing input is provided by the parent tree. Bugs similar to this one, however, do occur in actual trees and they often make it to deployment, being unobserved for days or even weeks, because they are often related to infrequent executions or to branches that are activated only to react and recover to exceptional situations. Nevertheless, these bugs have a large impact on the performance of the robot, as, when they occur, the BT execution (and thus the robot functionality) must be suspended. When applying the automatic test framework to the trees provided by Magazino, the algorithm was able to correctly detect some issues related to missing input user data. These were minor issues that would not substantially affect the operation of the robots in the field. This result was

[3]aiddl.org

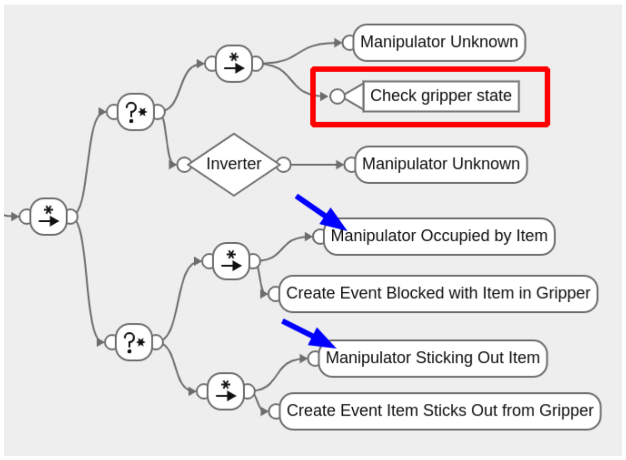
Manipulator Unknown
Check gripper state
Inverter
Manipulator Unknown
Manipulator Occupied by Item
Create Event Blocked with Item in Gripper
Manipulator Sticking Out Item
Create Event Item Sticks Out from Gripper

Figure 4: A fragment of a Magazino BT in which an output is produced by the framed node, and is required by all of the nodes pointed by an arrow. The BT is shown in Magazino custom left-to-right format.

expected, since the tested trees were already validated by a daily use of several hundred robots in production.

Evaluation on Randomly Generated Trees

We conducted four different analyses on random trees: (1) The total time used to check a tree. (2) The distribution of time between the different steps of our approach (pruning, initialization, solving). (3) The efficiency of the pruning (through the proportion of pruned tree). (4) The impact of control flow nodes on planning time. Finally, we look at the outliers (trees for which the planning time was longest) to see the structure of the trees and understand the cause for the increase solving time.

Evaluation was performed on a 12th Gen Intel(R) Core(TM) i9-12900H with 20GB of available memory. The complete analysis is available on Zenodo[4], together with additional results on the initialization and planning steps, not included in this paper for space reasons.

Random Tree Generation We generate random BTs based on the following parameters. The depth of the tree (from 3 to 10), the minimum and maximum number of child nodes for control flow nodes (2 and 3, respectively), the minimum and maximum number of producer nodes (1 and 3 respectively), the number of requiring nodes (1), the probability that a node is an execution node (leaf) before the maximum depth of the tree is reached (0.2), and the type and distribution of control flow nodes used when generating non-leaf nodes (see Table 2). For each setting in Table 2 we generated 100 trees (50 valid, 50 invalid) for each depth increment from 3 to 10 (inclusive), leading to a total of 800 trees for each setting. For the *Basic* set at depth 10 the average tree has 1587.5 nodes with a maximum of 3463 and a minimum of 121 nodes. The numbers of valid and invalid trees are guaranteed by using our approach during generation and making sure we keep generating trees until we have the desired number with each property.

[4]https://doi.org/10.5281/zenodo.7708806

Type	Basic	Advanced	Parallel
sequence	1/2	1/5	20/100
selector	1/2	1/5	20/100
inverter	0	1/5	20/100
on-failure	0	1/5	19/100
finally	0	1/5	19/100
parallel-all	0	0	1/100
parallel-selector	0	0	1/100

Table 2: Type and distribution of control flow nodes.

Results on Generated Trees

Total Time per Tree Type Figure 5 shows the total testing time per type of tree (Basic, Advanced, Parallel) for valid and invalid trees. As expected, the total amount of time taken by our framework increases linearly with the number of nodes. We note that the time remains well below 1 second in almost all cases even for the largest trees. We also note that there is no significant difference in testing time between the valid trees and the invalid trees.

Time Spent per Step We calculated the average time taken in each of three steps for all types of tree. Our approach takes the most time in the initialization step ($mean = 0.09s$), followed by the planning step ($mean = 0.02s$). The pruning step is negligible ($mean = 2.8 \cdot 10^{-5}s$), and the difference between these steps increases with the size of the tree. This result is homogeneous among the three tree types. This can be explained by the fact that initialization includes grounding (i.e., replacing each planning operator including variables by all possible grounded operators that do not include variables), heuristic initialization, compiling away variables that never appear in effects, and replacing composite terms with numerical terms for cheaper comparisons during search. This, combined with our pruning approach, leads comparatively low time needed to actually solve the problem. This result suggests that better grounding strategies might be adopted to reduce this step's total time. A domain dependent way to achieve this would be to perform grounding while traversing the tree when the state is created.

Impact of Pruning Figure 6 show the proportion of the tree that has effectively been pruned.

Thanks to our pruning strategy, the number of nodes in the pruned tree increases linearly with the depth of the tree, though the number of nodes in unpruned trees increases exponentially (assuming a branching factor greater than one). This analysis confirms the relevance of the pruning step in our approach. To check whether the longer testing time correlated with less efficient pruning, we calculated a Spearman correlation between the proportion of pruned nodes and total testing time in trees of depth 10[5]. Results do not show a sig-

[5]We only selected the largest set of trees to avoid artifacts due

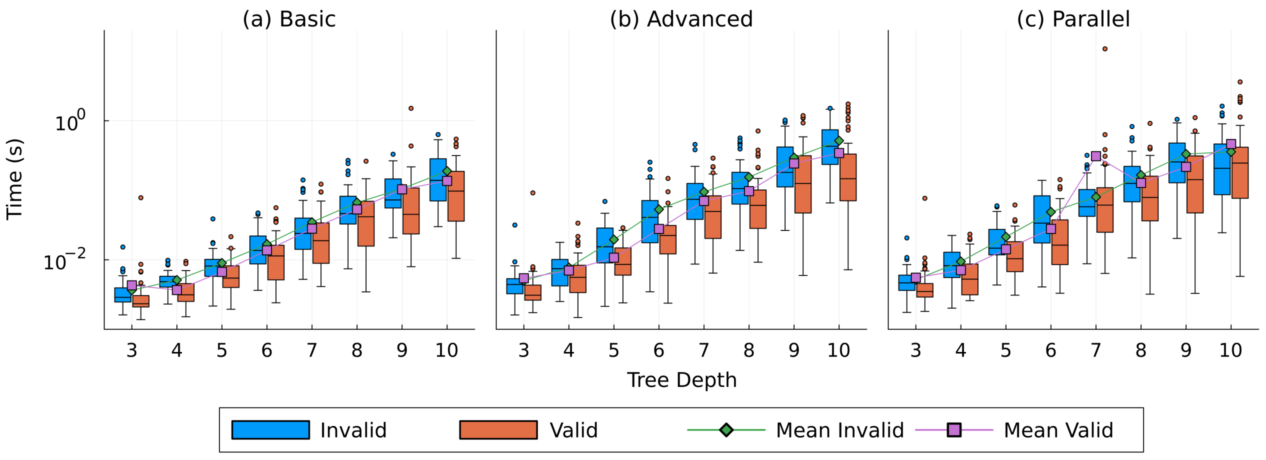

Figure 5: Total testing time over the three type of trees. Note that the y scale is logarithmic.

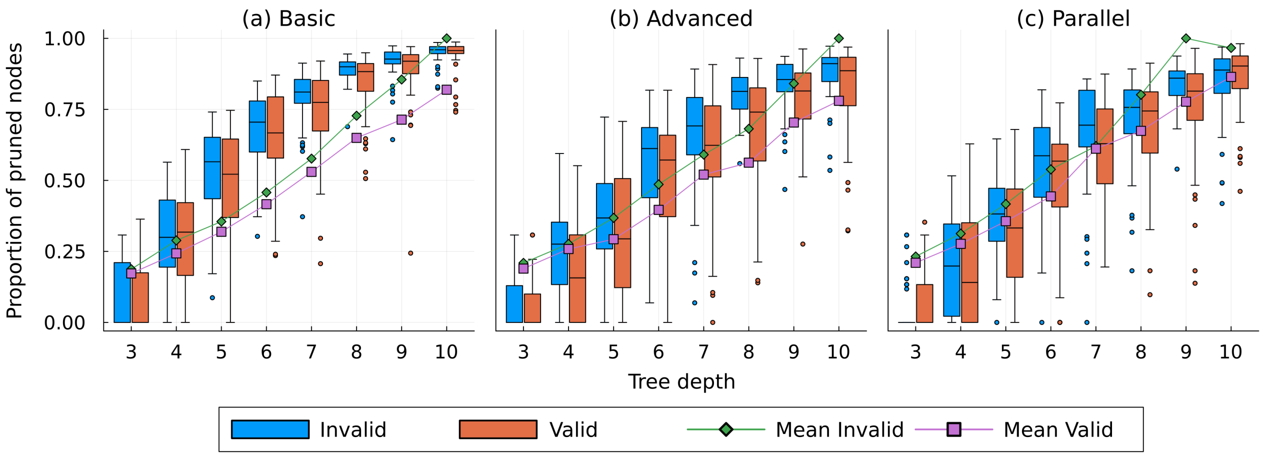

Figure 6: Proportion of the tree effectively pruned.

nificant correlation (Basic: $r = -0.26, p = 0.07$; Advanced $r = 0.007, p = 0.94$; Parallel $r = 0.16, p = 0.1$), indicating that these longer testing time are not due to inefficient pruning.

Influence of Node Types Finally, we evaluate the influence of each control flow node type in the initial BT (before pruning) on the total testing time by calculating a set of Spearman correlations over the trees of depth 10, adjusted for the family-wise error rate with the Holm-Bonferroni procedure. We found out that the only statistically significant correlation is with the parallel nodes ($\rho = 0.26$, $p = 0.006$, $\alpha = 0.006$), indicating that increasing the number of parallel control flow nodes increases the total testing time. This suggests that improved heuristics for handling parallel nodes could be beneficial in reducing the overall planning time. This result must, however, be put in perspective as there is a very small proportion of parallel nodes in the generated trees (2%). The full correlation table is available in the extended result.

to a too small number of nodes in the smaller trees.

Investigating Outliers Finally, we looked at some outliers, specifically problems with the longest testing time for the two largest depths for tree type (total = 6). We found out that all these outliers share a commonality: they all have 3 data producer nodes (the maximum possible), all which are situated very far down in the tree and in different branches. This leads us to think that the structure of the tree, i.e. the number of nodes and the position of producers and consumers, explains the longer testing time of some problems.

Comparison with Magazino Trees In order to confirm that the generated trees are representative of real cases, we selected a subset of the BTs used by Magazino to control the behavior of its robots. The selection includes larger trees that make them hard to analyze just by visual inspection. Table 3 shows the statistics extracted from this selection. The depth of the trees and the average number of children per composite are consistent with the parameters used to generate the trees described in the previous sections. The distribution of node types is also similar, apart from a smaller number of *finally* and *on-failure* and the presence of other composite and

Statistic	Value
Average depth of the tree	10.22
Average number of composite children	2.7
% sequence	0.3
% selector	0.18
% inverter	0.1
% on-failure	0.03
% finally	0.01
% parallel	0.05
% (other)	0.32

Table 3: Statistics extracted from a selection of the real trees used by Magazino on their robots.

decorator types that have not been considered in this paper as they are slight variations of those defined here.

Conclusion

We have provided a method that verifies that data dependencies in BTs (often implicit) are met during their execution. Our approach converts this problem into an automated planning problem that attempts to find a sequence of events in a BT that reaches a node without the required data available. We introduced a pruning algorithm that replaces irrelevant sub-trees with execution nodes to reduce the complexity of the overall tree before planning. The approach was evaluated on various sets of randomly generated trees that reach similar numbers of execution nodes and have a comparable average number of child nodes to the tree used at Magazino.

The presented approach focuses on validating one data requirement at a time which allows pruning significant amounts of the input tree. However, this limits the expressivity of the properties we can test. Thus one direction for future work is to investigate if and how we can test for more complex properties, such as the full set of features of Conditional BTs, while preserving the overall performance on realistically sized BTs. In addition, we intend to extend the range of control flow node operators in order to deploy the approach for usage on real behavior trees at Magazino.

Acknowledgements

This work has been supported by the European Union's Horizon 2020 research and innovation programme under grant agreement No. 101016442 (AIPlan4EU).

References

Agis, R. A.; Gottifredi, S.; and García, A. J. 2020. An event-driven behavior trees extension to facilitate non-player multi-agent coordination in video games. *Expert Systems with Applications*, 155: 113457.

Bäckström, C.; and Nebel, B. 1995. Complexity Results for SAS+ Planning. *Comput. Intell.*, 11: 625–656.

Biggar, O.; and Zamani, M. 2020. A Framework for Formal Verification of Behavior Trees With Linear Temporal Logic. *IEEE Robotics and Automation Letters*, PP: 1–1.

Bojic, I.; Lipic, T.; Kusek, M.; and Jezic, G. 2011. Extending the JADE Agent Behaviour Model with JBehaviourTrees Framework. In *Proceedings of the 5th KES International Conference on Agent and Multi-Agent Systems: Technologies and Applications*, 159–168. ISBN 9783642219993.

Colledanchise, M.; Cicala, G.; Domenichelli, D. E.; Natale, L.; and Tacchella, A. 2021. Formalizing the Execution Context of Behavior Trees for Runtime Verification of Deliberative Policies. In *2021 IEEE/RSJ International Conference on Intelligent Robots and Systems (IROS)*, 9841–9848.

Colledanchise, M.; and Ögren, P. 2017. Behavior Trees in Robotics and AI: An Introduction. *CoRR*, abs/1709.00084.

Gemignani, G. 2021. Controlling an apparatus, e.g., a robot, with a behavior tree. Patent Number EP 4040293A1.

Ghallab, M.; Howe, A.; Knoblock, C.; McDermott, D.; Ram, A.; Veloso, M.; Weld, D.; and Wilkins, D. 1998. PDDL - The Planning Domain Definition Language. Technical report, CVC TR-98-003/DCS TR-1165, Yale Center for Computational Vision and Control.

Ghallab, M.; Nau, D.; and Traverso, P. 2004. *Automated Planning: Theory and Practice*. Morgan Kaufmann.

Giunchiglia, E.; Colledanchise, M.; Natale, L.; and Tacchella, A. 2019. Conditional Behavior Trees: Definition, Executability, and Applications. In *2019 IEEE International Conference on Systems, Man and Cybernetics (SMC)*, 1899–1906.

Helmert, M. 2006. The fast downward planning system. *Journal of Artificial Intelligence Research*, 26(1): 191–246.

Iovino, M.; Scukins, E.; Styrud, J.; Ögren, P.; and Smith, C. 2022. A survey of Behavior Trees in robotics and AI. *Robotics and Autonomous Systems*, 154: 104096.

Isla, D. 2005. Handling complexity in the Halo 2 AI. In *Game Developers Conference (Vol. 12)*, volume 12.

Köckemann, U. 2020. The AI Domain Definition Language (AIDDL) for Integrated Systems. In *KI 2020: Advances in Artificial Intelligence*, 348–352.

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

Nicolau, M.; Pérez-Liébana, D.; O'Neill, M.; and Brabazon, A. 2017. Evolutionary Behavior Tree Approaches for Navigating Platform Games. *IEEE Transactions on Computational Intelligence and AI in Games*, 9: 227–238.

Ögren, P. 2012. Increasing Modularity of UAV Control Systems using Computer Game Behavior Trees. In *Proceedings of AIAA Guidance, Navigation, and Control Conference*.

Sekhavat, Y. 2017. Behavior Trees for Computer Games. *International Journal on Artificial Intelligence Tools*, 26.

Serbinowski, B.; and Johnson, T. T. 2022. BehaVerify: Verifying Temporal Logic Specifications for Behavior Trees. In *Proceedings of the 20th International Conference on Software Engineering and Formal Methods (SEFM)*, 307–323.

Tenorth, M. 2016. Controlling process of robots having a behavior tree architecture. Patent Number EP 3214510B1.

Yang, S.; Mao, X.; Wang, S.; and Bai, Y. 2021. Extending Behavior Trees for Representing and Planning Robot Adjoint Actions in Partially Observable Environments. *Journal of Intelligent & Robotic Systems*, 102(2): 36.

Yatapanage, N.; Winter, K.; and Zafar, S. 2010. Slicing behavior tree models for verification. In *IFIP International Conference on Theoretical Computer Science*, 125–139.