Formalizing Stateful Behavior Trees

Serena S. Serbinowska
0000-0002-9259-1586
Vanderbilt University
Nashville TN, USA
serena.serbinowska@vanderbilt.edu

Preston Robinette
0000-0002-4906-2179
Vanderbilt University
Nashville TN, USA
preston.robinette@vanderbilt.edu

Gabor Karsai
0000-0001-7775-9099
Vanderbilt University
Nashville TN, USA
gabor.karsai@vanderbilt.edu

Taylor T. Johnson
0000-0001-8021-9923
Vanderbilt University
Nashville TN, USA
taylor.johnson@vanderbilt.edu

Behavior Trees (*BT*s) are high-level controllers that are useful in a variety of planning tasks and are gaining traction in robotic mission planning. As they gain popularity in safety-critical domains, it is important to formalize their syntax and semantics, as well as verify properties for them. In this paper, we formalize a class of *BT*s we call Stateful Behavior Trees (*SBT*s) that have auxiliary variables and operate in an environment that can change over time. *SBT*s have access to persistent shared memory—often known as a blackboard—that keeps track of these auxiliary variables. We demonstrate that *SBT*s are equivalent in computational power to Turing Machines when the blackboard can store mathematical (unbounded) integers. We also identify conditions where *SBT*s have computational power equivalent to finite state automata, specifically where the auxiliary variables are of finitary types. We present a domain specific language (DSL) for writing *SBT*s and adapt the tool BehaVerify for use with this DSL. This new DSL in BehaVerify supports interfacing with popular *BT* libraries in Python, and also provides generation of Haskell code and nuXmv models, the latter of which are used for model checking temporal logic specifications for the *SBT*s. We include examples and scalability results where BehaVerify outperforms another verification tool (MoVe4BT) by a factor of 100.

1 Introduction

A Behavior Tree (*BT*) is a high-level controller that shares similarities with a hierarchical state machine, yet distinguishes itself by offering greater flexibility and modularity in defining behaviors. At its core, a *BT* organizes various behaviors within a tree structure, where leaf nodes encapsulate distinct behaviors and higher-level nodes define the control flow. This hierarchical arrangement facilitates the design of complex behaviors and is both scalable and adaptable to changing circumstances or requirements.

*BT*s were originally created for video game development and were devised to enhance the autonomy and realism of Non-Playable Characters (NPCs). An NPC is an entity within a video game that operates under the control of the game's Artificial Intelligence (AI). *BT*s are useful for specifying such behaviors. The explainability and versatility of *BT*s have also led to their widespread adoption in areas like robotics and AI. Accordingly, *BT*s have been used for a variety of tasks, such as for controlling wheeled-legged robots [10] and bipedal locomotion robots [16], in vision measurement systems of road users [24], and the management swarms [18, 20]. Additional applications can be found in this survey [19].

As *BT*s continue to be adopted to address new and existing challenges in various domains, especially in real-world, safety-critical domains such as robotics, it is increasingly important to formalize their structure and behaviors. Such formalization is crucial for the verification of safety and liveness specifications, ensuring the systems behave reliably and as intended under all conditions. Toward this end, we provide

To appear in EPTCS.

a formalization we call Stateful Behavior Trees (*SBT*s). *SBT*s are a class of *BT*s that have auxiliary variables and operate in an environment. The primary contributions of this work are the following.

1. We formalize a novel class of models we call *SBT*s that operate in an environment and have global variables stored in persistent shared memory.
2. We demonstrate equivalence of *SBT*s to Turing Machines and Finite State Automata under syntactic assumptions, which is of critical importance for model checking *BT*s.
3. We present a domain specific language (DSL) for writing *SBT*s implemented in an entirely reworked software tool called BehaVerify [25].
4. We compare the entirely reworked BehaVerify [25] to MoVe4BT [23] in different verification examples and outperform in each; in one, BehaVerify outperformed by a factor of over 100.

2 Related Work

In this section, we discuss relevant literature, focusing on the verification of *BT*s and domain specific languages (DSLs) for *BT*s. We highlight the contributions of this paper within these contexts.

Our Prior Work In our prior work we presented BehaVerify [25]. That version of BehaVerify [25] took as input a Py Trees [27] object and walked the tree to create a nuXmv [5] model. The created nuXmv [5] model was incomplete; it had composite and decorator nodes, but the leaf nodes were 'stubs' for the user to fill in. The same was true of variables. The new version of BehaVerify [25] utilizes a Domain Specific Language (DSL). It takes as input a *BT* specified using the DSL and produces as output a nuXmv [5] model, a Py Trees [27] implementation of the *BT*, or a Haskell implementation of the *BT*. Crucially, the nuXmv [5] model is now complete; there are no 'stubs' for the user to fill in as all the variables and leaf nodes are fully and completely generated.

Existing Behavior Tree Frameworks To our knowledge, there are no existing DSLs for *BT*s, but there are several related libraries and frameworks. [15] lists a variety of different DSLs, but we believe these would more correctly be classified as library implementations of *BT*s (e.g. BehaviorTree.CPP [1] and Py Trees [27]). PROMISE [14] is a DSL inspired by *BT*s, but it is not a DSL for *BT*s. MoVe4BT [23], which we compare against, uses an xml style for specifying *BT*s.

Verifying Behavior Trees There are several existing formal verification works for *BT*s. [2] utilizes SPOT [12] for verification of *BT*s, but is limited to atomic propositions and boolean operators. Furthermore, the examples provided seemed to take over an hour to run for very small trees. [6] does runtime verification for a fragment of Timed Propositional Temporal Logic (TPTL), but not design-time model verification. We compared against BTCompiler in our previous paper [25]. To the best of our knowledge, the only other existing and available tools for model verification of *BT*s are ArcadeBT [17] and MoVe4BT [23].

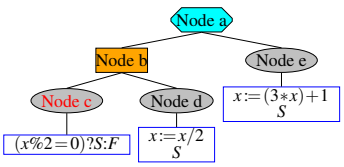
ArcadeBT [17] is an automatic verification method for *BT*s that verifies safety properties by encoding the *BT* using Linear Constrained Horn Clauses (LCHCs). To do this, ArcadeBT [17] includes an implementation of *BT*s in C++ that can be automatically converted to LCHCs and verified using Z3 [22]. The tool has been evaluated on trees with up to 18 nodes. By comparison, our new DSL and implementation in BehaVerify handles trees with 20000 nodes (see Section 7) and supports verification of linear temporal logic (LTL) and computation tree logic (CTL) allowing for both liveness and safety to be verified.

MoVe4BT [23] allows for the verification of LTL specifications over nodes, but it cannot verify LTL specifications written as predicates over variables. In contrast, the implementation of *SBT* verification in BehaVerify developed in this paper supports LTL specifications over variables and nodes. MoVe4BT [23] supports nodes with true parallelism, while BehaVerify does not. However, Py Trees [27], a popular implementation of *BT*s that BehaVerify targets, does not support true parallelism. A more detailed comparison of the tools can be found in Section 7, including experimental evaluations that demonstrate BehaVerify is able to verify trees 100 times bigger than MoVe4BT [23] (20000 vs 200 nodes).

BTs with State and Theoretical Foundation for Verification While not universal, it is common for *BT*s to interact with memory, referred to here as a *blackboard*. [4] compares pure *BT*s (*BT*s without a blackboard) to unrestricted *BT*s (*BT*s with a finite blackboard). While the unrestricted *BT*s are strictly more powerful than the pure *BT*s, [3] points out that this violates the 'reactive' nature of *BT*s ([3] states "An architecture is reactive if its decision making depends only on the current state of the environment"). Instead of using blackboards, the authors of [3] advocate for combining *BT*s with Stateful Components, thereby preserving the benefits of *BT*s without loss of computational power. Regardless, practical major implementations of *BT*s (such as Py Trees [27] and its Robotic Operating System (ROS) extension PyTreesRos, BehaviorTree.cpp [1], and Unreal Engine [13]) all feature the blackboard. As such, there is a practical need for a framework that addresses *BT*s with blackboards.

Other researchers take a different approach and treat *BT*s as deterministic functions with control systems, as seen in the works by [29], [9], [28], [26], and [8], with [21] analyzing the potential of *BT*s as alternatives to Controlled Hybrid Dynamical Systems. While these undoubtedly describe *BT*s with state, there are crucial differences between this style and our formalization of *SBT*s. The issue at hand is that 'state' is an ambiguous term; it could refer to either memory or to the environment. For instance, it was not assumed that pure *BT*s do not function in a persistent environment; rather, the assumption was that the *BT* does not leverage its own persistent memory to augment its behavior. In essence, pure *BT*s are functional; if presented with the same set of inputs, they will produce the same outputs. In this sense, the control system approach utilizes pure *BT*s without memory; the state represents the environment rather than memory. In contrast, *SBT*s consider both a blackboard *and* the environment. Moreover, the environment in the control system model was fully under control of the *BT*. While this may be a reasonable assumption in certain contexts, it makes it impossible to model uncertainty (such as the presence of wind for drones). As such, our formalization allows for nondeterministic updates. This is extended to the environment, which is allowed to change and develop according to user defined rules that can utilize nondeterminism.

Computational Power Finally, we consider the computational power of the resulting models. [7] informally says *BT*s are the same as Finite State Machines (*FSM*s), but does not explicitly state any assumptions, restrictions, etc. On the other hand, [4] creates a hierarchy of Teleo-reactive programs (TR), Decision Trees (DT), *BT*s, and *FSM*s. They conclude that if you provide a TR, DT, or *BT* with access to a finite blackboard that they can freely read from/write to, then they are equivalent to a *FSM*. Giving a *FSM* access to a finite blackboard does not increase the computational power of the model, as this is the equivalent of adding a finite number of additional states. We take this a step further and consider the power of a *SBT* with an infinite blackboard (a blackboard capable of storing variables of unbounded size) and conclude that such a model has the computational power of a Turing Machine.



Tick	1	1	1	1	1	1	1	2	2	2	2	2	2	2
t	1	2	3	4	5	6	7	8	9	10	11	12	13	14
Active	a	b	c	b	d	b	a	a	b	c	b	a	e	a
Returns	-	-	*S*	-	*S*	*S*	*S*	-	-	*F*	*F*	-	*S*	*S*
x	6	6	6	6	3	3	3	3	3	3	3	3	10	10

Figure 1: A *BT* for the Collatz conjecture (hailstone sequence) consisting of a selector node (a), a sequence node (b), a check (c), and two actions (d, e). We use the ternary operator $i?j:k$ to mean if i then j else k. We use % for modulo. Tick indicates the number of times the tree has been ticked. t is used to track the number of times we have changed active nodes. Active is used to track where we are in the tree. Returns indicates what the active node returns; a - indicates the node did not return. If a node is finished, then it returns one of S, F, or R.

3 Behavior Tree Overview

We will utilize Figure 1 to provide an intuitive explanation of *BT*s. We will then provide additional details.

In Figure 1 we have a basic *BT* for calculating a hailstone sequence from a starting value. To calculate such a sequence, start with a positive integer and use the following rules: if the number is even, divide by 2; otherwise, multiply by 3 and add 1. The root of the tree (a) is a selector node. As the name implies, this node 'selects' a child. In this case, we want to select either 'divide by 2' or 'multiply by 3 and add 1'. (b) is a sequence node; aptly named once more, this node executes a sequence, aborting if a failure is encountered. Here our sequence is 'number is even' followed by 'divide by 2'. (c) is a check node; it checks if a condition is true or false. (d) and (e) are action nodes; they actually do stuff.

Our tree starts when it receives an external signal (a *tick*). This causes the root (a) to become active. Execution will now, mostly, follow a *depth first traversal* (DFT). (a) hasn't yet selected an option, so it follows DFT and makes (b) active. (b) hasn't completed the sequence or encountered a failure, so it follows DFT and makes (c) active. (c) checks if x is even; since 6 is even it returns Success (S). (b) is active again, but it still hasn't completed a sequence or encountered a failure, so it follows DFT and makes (d) active. (d) executes the action and halves the value of x and returns S. (b) is active again and the sequence successfully finished, so S is returned. (a) is finally active again and it has selected an option, so it returns S. The first tick is now over. The tree will now do nothing until it receives another tick. When it does, (a) again becomes active, then (b), then (c). This time, however, x is not even, so (c) returns Failure (F). Thus the sequence failed, so (b) returns F. (a) is now active again, but still hasn't selected an option, so (e) becomes active. (e) executes the action; x becomes 10 and S is returned. (a) has now selected an option, so it returns S. The second tick is now over.

We now provide some more concrete requirements for *BT*s that were not covered by the example. There must be a path from the root to every other node in the tree. Each node has exactly one parent, except the root which has no parent. There are four possible states for nodes: Success (S), Failure (F), Running (R), and Invalid (I). Each node becomes I when a new tick arrives. When a node finishes executing it returns one of S, R, or F. Finally, there are three types of nodes: leaf, decorator, and composite.

Leaf Nodes Leaf nodes are nodes that do not have any children, and they either check a condition (check node) or do an action (action node). Check nodes return S if the associated condition is true and F otherwise; they do not do anything else. Action nodes can perform various actions. Furthermore, they are allowed to return S, F, or R and can utilize conditions to determine what to return. In Figure 1, (c) is a check node while (d) and (e) are action nodes.

Decorator Nodes A decorator node always has exactly one child. For our purposes, decorator nodes are used to change the output of a child node without requiring the child node to be modified. Some common decorator types are inverter, which swaps S and F, and R_Is_F which turns R into F.

Composite Nodes Composite nodes control the execution flow through a BT. There are three types of composite nodes: selector, sequence, and parallel. The children are ordered and for convenience we will use a left-to-right order.

1. Selector or fallback nodes, try to 'select' a child. A child is 'selected' if it returns S. Each child is activated in order, from left to right, until one of them returns S or R. At that point, the selector returns the same status. If every child returns F, the selector returns F.
2. Sequence nodes are identical to selector nodes, except S and F are swapped. While this is true and useful to note, it is more practical to think of them in a more distinct manner. Sequence nodes are used to execute a sequence to the end or a failure point. Each child is activated in order, from left to right, until one of them returns F or R. At that point, the sequence returns the same status. If every child returns S, then the sequence returns S.
3. Parallel nodes will not appear in this paper, but it is still important to mention them. As the name implies, parallel nodes activate all their children simultaneously. We do not support this behavior and neither does Py Trees [27], the Python implementation that BehaVerify targets. Instead, our parallel nodes activate each child in order, one at a time, left to right. Unlike selector and sequence nodes, there is no early termination condition for parallel nodes; each child *will* be activated. Once all the children have returned, the parallel node consults a policy to determine what to return.

4 Formal Definition of Stateful Behavior Trees

Here we provide a formal definition of a SBT. In service of this task, we start by defining a tree. A rooted tree is a triple (V,r,E) such that

- V is a finite set representing the vertices of the tree.
- $r \in V$ is a vertex representing the root.
- Let VS be the set of all finite sequences $vs = [v_0, v_1, \ldots, v_n]$ such that $\forall j,k \in \mathbb{Z}$ s.t. $(0 \leq j,k \leq n), (v_j, v_k \in V \wedge (v_j = v_k \implies j = k))$. That is to say, the elements of the sequence are unique vertices. If a and b are elements in a sequence, we will use $<,>,\leq,\geq$ to indicate relative order of the sequence. For example $a < b$ means that a appears before b in the sequence.
- $E : V \mapsto VS$ is a function from vertices to sequences of vertices (the children). It must also meet the following requirements
  - $\forall v \in V, r \notin E(v)$ (the root has no parent).
  - $\forall v,v' \in V, v \neq v' \implies E(v) \cap E(v') = \varnothing$. Each vertex has at most one parent.
  - $\forall v \in V, \exists [v_0,v_1,\ldots,v_n]$ s.t. $v_0 = r \wedge v_n = v \wedge \forall j \in \mathbb{Z}$ s.t. $0 \leq j < n, v_{j+1} \in E(v_j)$. There exists a path from the root to each vertex.

  These conditions ensure that the tree is actually a tree.

4.1 Stateful Behavior Tree

A SBT is a tuple $(V,r,E,S_{SBT},s_{SBT},\Sigma_{SBT},\delta_{SBT})$ such that

- (V,r,E) is a tree.
- S_{SBT} is a set representing the possible states of the blackboard of SBT. We discuss the implications of this set being infinite vs finite in Subsection 4.2.
- $s_{SBT} \in S_{SBT}$ is the initial state of the blackboard.
- Σ_{SBT} is a set representing the possible inputs (the environment).
- ST is the set of all functions $st : V \mapsto \{S,R,F,I\}$. Each $st \in ST$ is a function that maps each vertex to a status. ST is not an element of the tuple; it arises from the elements.
- $\delta_{SBT} : V \times ST \times S_{SBT} \times \Sigma_{SBT} \mapsto 2^{V \times ST \times S_{SBT}}$. Here $2^{V \times ST \times S_{SBT}}$ is the power set of $V \times ST \times S_{SBT}$. The function maps to sets to allow for the expression of nondeterminism. This function must also obey the following:

$$\forall v,v' \in V, \forall st,st' \in ST, \forall s,s' \in S_{SBT}, \forall a \in \Sigma_{SBT}, (v',st',s') \in \delta_{SBT}(v,st,s,a) \implies$$
$$\left(\begin{array}{c} (v = r \wedge st(v) \neq I \implies (v' = r \wedge s = s' \wedge (\forall v'' \in V, st'(v'') = I))) \wedge \\ (v \neq r \implies \forall v'' \in V, (v'' = v) \vee st(v'') = st'(v'')) \wedge \\ (v' = v = r \vee v' \in E(v) \vee v \in E(v')) \wedge \\ (v \in E(v') \implies st'(v) \neq I) \wedge \\ (v' \in E(v) \implies st(v) = st'(v) = st(v') = I) \wedge \\ (v' \in E(v) \implies \forall v'' \in E(v), st(v'') \neq I \vee v' \leq v'') \end{array} \right)$$

  We will refer to v as the active node and v' as the next node while explaining the above. If the root is active and not I, then we reset the status of the tree without changing anything else. In all other cases, only the status of the active node can be updated. The next node is either the root, the child of the active node, or the parent of the active node. If the next node is the parent of the active node, then the next status of the active node will not be I. If the active node is the parent of the next node, then the status of the active node and the next status of the active node are I and the status of the next node is I. If the active node is the parent of the next node, then all children that appear earlier in the sequence of active node's children are not I. These rules ensure that we move through the tree in the appropriate order.

Let $[a_0,a_1,...]$ be a sequence of inputs from Σ_{SBT}. Then a SBT trace is a sequence $[(v_0,st_0,s_0),(v_1,st_1,s_1),...]$ such that $v_0 = r$, $s_0 = s_{SBT}$, $\forall v \in V, st_0(v) = I$, and $\forall j \in \mathbb{Z}, j \geq 0 \implies (v_{j+1},st_{j+1},s_{j+1}) \in \delta_{SBT}(v_j,st_j,s_j,a_j)$.

4.2 Translating Stateful Behavior Trees to Finite State Machines

Assuming that SBT has a finite alphabet set and a finite set of states, we will translate it into a nondeterministic Finite State Machine (FSM). A FSM is a tuple $(S_{FSM},s_{FSM},\Sigma_{FSM},\delta_{FSM})$.

- S_{FSM} is a set of states and $s_{FSM} \in S_{FSM}$ is the initial state.
- Σ_{FSM} is a set of possible inputs.
- $\delta_{FSM} : S_{FSM} \times \Sigma_{FSM} \mapsto 2^{S_{FSM}}$ is the transition function.

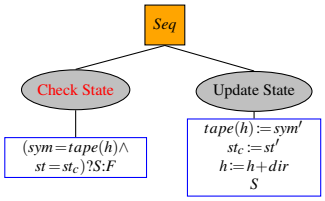


Figure 2: Assume that $f(sym,st)=(sym',st',dir)$, where f is the transition function for the Turing Machine. Then this means that if the Turing Machine is in state st and reads sym from the tape head, it will write sym' to the tape head, transition to st', and move the tape head according to dir. The subtree captures this behavior.

Let $[a_0,a_1,...]$ be a sequence of inputs from Σ_{FSM}. Then a *FSM* trace is a sequence $[s_0,s_1,...]$ such that $s_0=s_{FSM}$ and $\forall j\in\mathbb{Z}, j\geq 0 \implies s_{j+1}\in\delta_{FSM}(s_j,a_j)$.

Let $SBT=(V,r,E,S_{SBT},s_{SBT},\Sigma_{SBT},\delta_{SBT})$. Assume S_{SBT} and Σ_{SBT} are finite. Furthermore, let ST be the set of all functions $st: V\mapsto\{S,R,F,I\}$ (as defined earlier). Since V is a finite set, ST is a finite set. Then $V\times S_{SBT}\times ST$ is finite as well. Let n be the number of elements in $V\times S_{SBT}\times ST$. Create a one-to-one mapping $BTSM$ from $V\times S_{SBT}\times ST$ to the integer interval $[0,1,\ldots,n-1]$. Let $\delta_{FSM}:[0,1,...,n-1]\times\Sigma_{SBT}\mapsto 2^{[0,1,\ldots,n-1]}$ be defined such that

$$\forall v\in V,\forall s\in S_{SBT},\forall st\in ST,\forall a\in\Sigma_{SBT},\forall(v',s',st')\in\delta_{SBT}(v,s,st,a)$$
$$BTSM(v',s',st')\in\delta_{FSM}(BTSM(v,s,st),a).$$

Then $([0,1,\ldots,n-1],BTSM(r,s_{SBT},st_0),\Sigma_{SBT})$ is an equivalent *FSM*, where $st_0\in ST$ such that $\forall v\in V, st_0(v)=I$. By translating the *SBT* to a *FSM*, we allow for verification with tools such as nuXmv [5].

Turing Complete In our translation, we assumed that S_{SBT} was finite. If the blackboard can store one or more infinite variables (e.g. true integers), then S_{SBT} is not finite and *SBT*s are Turing Complete. To see this, consider the following:

- While the 'tape' of a Turing Machine (TM) is infinite, the alphabet of symbols that can appear in each cell is finite. Assume that there are n such symbols. Then the 'tape' can clearly be represented as an integer in base n where each cell is represented by a digit.
- There are finitely many states that the Turing Machine can be in. Thus, these can be enumerated and stored using a finite integer.
- The location of the 'tape head' can be stored using an unbounded integer.

Thus by storing two unbounded integers and a bounded integer in the blackboard, we can fully capture the state and tape of a TM. All that remains is to reproduce the transition function of the TM. Because the transition function is a function from the set of finite states and finite alphabet to the set of finite states, finite alphabet, and a tape motion (Left, Right), it can easily be captured using a BT. For each possible input to the transition function that has a defined output, create a 3-node subtree as seen in Figure 2. Finally, add a selector root node with the 3-node subtrees as children.

Turing Incomplete A TM that only has access to a finite tape is not Turing Complete. Similarly, if we restrict the blackboard to storing finitely many finite variables, the *SBT* ceases to be Turing Complete, as seen by the fact that a translation to a *FSM* exists.

5 DSL and Implementation Details

In this section, we provide details regarding the BehaVerify DSL used to specify *SBT*s. The grammar presented in Grammar 1 differs from the actual DSL (see [1]) for the following reasons:

1. **Syntactic sugar.** For example, the actual DSL allows for fixed size arrays. In practice, this is equivalent to utilizing a large number of variables, but is more convenient (especially when combined with some basic loop functions).
2. **Visual divisions.** The actual DSL uses far more pronounced visual dividers between sections (e.g., a case result is written as 'case { Code } result { Code, Code, ... }'). This ensures that a specification written using the implementation is readable. However, it also injects a great deal of 'text' into the grammar of the DSL, making it more difficult to parse here.
3. **Other features.** The actual DSL is still actively being developed for new features, many of which are not relevant to this paper (e.g., hyperproperties). Such features were omitted.
4. **Format.** The actual DSL is written for use with textX [11] rather than with Backus-Naur Form.

⟨*SBT*⟩ ::= ⟨*Enums*⟩ ';' ⟨*Consts*⟩ ';' ⟨*BLVars*⟩ ';' ⟨*ENVVars*⟩ ';' ⟨*Env*⟩ ';' ⟨*Chks*⟩ ';' ⟨*Acts*⟩ ';' '{' ⟨*Node*⟩ '}' ';' ⟨*Specs*⟩ ';'

⟨*Status*⟩ ::= 'success' | 'running' | 'failure'

⟨*Code*⟩ ::= ⟨*Int*⟩ | ⟨*Boolean*⟩ | ⟨*String*⟩ | ⟨*ID*⟩ | '(' ⟨*Function*⟩ ')'
#ID is used to reference variables, constants, etc

⟨*CodeList*⟩ ::= ⟨*Code*⟩ | ⟨*Code*⟩ ',' ⟨*CodeList*⟩

⟨*Function*⟩ ::= 'add' ',' ⟨*Code*⟩ ',' ⟨*CodeList*⟩ | 'not' ',' ⟨*Code*⟩ | ...

⟨*Enums*⟩ ::= ε | '{' ⟨*String*⟩ '}' ⟨*Enums*⟩ #ε is the empty string

⟨*Const*⟩ ::= ⟨*ID*⟩ ':=' ⟨*Int*⟩ | ⟨*ID*⟩ ':=' ⟨*Boolean*⟩ | ⟨*ID*⟩ ':=' ⟨*String*⟩

⟨*Consts*⟩ ::= ε | '{' ⟨*Const*⟩ '}' ⟨*Consts*⟩ #ε is the empty string

⟨*Domain*⟩ ::= 'BOOLEAN' | '[' ⟨*Code*⟩ ',' ⟨*Code*⟩ ']' | '{' ⟨*CodeList*⟩ '}'
#Code is used to allow constants and expressions

⟨*CaseResult*⟩ ::= ⟨*Code*⟩ '?' ⟨*CodeList*⟩ #If Case (left), then Result (right).
#Choose nondeterministically if multiple Results

⟨*Assign*⟩ ::= '{' ⟨*CaseResult*⟩ '}' ⟨*Assign*⟩ | '{' ⟨*CodeList*⟩ '}'
#Try each CaseResult until one works. Default to CodeList if all fail

⟨*RCaseResult*⟩ ::= ⟨*Code*⟩ '?' ⟨*Status*⟩
#Same as CaseResult but for status. Deterministic.

⟨*RAssign*⟩ ::= '{' ⟨*RCaseResult*⟩ ',' ⟨*RAssign*⟩ '}' | '{' ⟨*Status*⟩ '}'

[1] https://github.com/verivital/behaverify/blob/main/metamodel/behaverify.tx

⟨*Statement*⟩ ::= ⟨*ID*⟩ ‘,’ ⟨*Assign*⟩ #Update variable ID using Assign

⟨*Statements*⟩ ::= ε | ‘{’ ⟨*Statement*⟩ ‘}’ ⟨*Statements*⟩ #ε is the empty string

⟨*Var*⟩ ::= ⟨*ID*⟩ ‘,’ ⟨*Domain*⟩ ‘,’ ⟨*Assign*⟩

⟨*BLVars*⟩ ::= ε | ‘{’ ⟨*Var*⟩ ‘}’ ⟨*BLVars*⟩ #ε is the empty string

⟨*ENVVars*⟩ ::= ⟨*BLVars*⟩

⟨*Envs*⟩ ::= ⟨*Statements*⟩

⟨*Chk*⟩ ::= ⟨*ID*⟩ ‘,’ ⟨*Code*⟩ #Code must resolve to a Boolean
#ID is the NodeType. Allows for reusage.

⟨*Chks*⟩ ::= ε | ‘{’ ⟨*Chk*⟩ ‘}’ ⟨*Chks*⟩ #ε is the empty string

⟨*Act*⟩ ::= ⟨*ID*⟩ ‘,’ ⟨*Statements*⟩ ⟨*RAssign*⟩ ⟨*Statements*⟩
#ID is the NodeType. Allows for reusage.

⟨*Acts*⟩ ::= ε | ‘{’ ⟨*Act*⟩ ‘}’ ⟨*Acts*⟩ #ε is the empty string

⟨*NodeBody*⟩ ::= ‘`sequence`’ ‘{’ ⟨*Children*⟩ ‘}’ | ‘`selector`’ ‘{’ ⟨*Children*⟩ ‘}’
| ‘`parallel one`’ ‘{’ ⟨*Children*⟩ ‘}’ | ‘`parallel all`’ ‘{’ ⟨*Children*⟩ ‘}’
| ‘`inverter`’ ‘{’ ⟨*Node*⟩ ‘}’ | ‘`X_is_Y`’ ⟨*Status*⟩ ⟨*Status*⟩ ‘{’ ⟨*Node*⟩ ‘}’
| ⟨*ID*⟩ ‘:=’ ⟨*ID*⟩ #Name of leaf node (left) and NodeType (right)

⟨*Node*⟩ ::= ⟨*ID*⟩ ‘,’ ⟨*NodeBody*⟩

⟨*Children*⟩ ::= ‘{’ ⟨*Node*⟩ ‘}’ | ‘{’ ⟨*Node*⟩ ‘}’ ⟨*Children*⟩

⟨*Spec*⟩ ::= ‘`LTL`’ ‘{’ ⟨*Code*⟩ ‘}’ | ‘`CTL`’ ‘{’ ⟨*Code*⟩ ‘}’ | ‘`Invar`’ ‘{’ ⟨*Code*⟩ ‘}’

⟨*Specs*⟩ ::= ε | ⟨*Spec*⟩ ⟨*Specs*⟩ #ε is the empty string

Grammar 1: Representation of BehaVerify DSL with slight changes. We avoid defining basic types such as Int or ID (a letter followed by letters or digits).

Note that both Grammar 1 and the actual DSL allow for nonsensical statements (e.g. (add, 1, 'dog')). It is possible to create a grammar to exclude such cases, but the practical implementation proved both cumbersome to maintain and slow in practice. Instead, we made a semantic checker for basic type checking along with other cases not covered by the grammar structure (e.g. ensuring that identifiers are unique). Listing 1 is an example of how Grammar 1 would be used to create the *SBT* in Figure 4.

Listing 1: A basic example of a *SBT* specified using Grammar 1

```
;;//no enumerations or constants
{x, [(neg, 1), 5], {0, 1}};
//^Bl var x, nondeterministically initialized to 0 or 1
{y, {0, 1}, {(eq, x, 0) ? 0} {1}};
//^Env var y, initialized to 0 if x is 0, otherwise 1
{y, {(eq, y, 0) ? 1} {0}};//between ticks, swap y value
{cN, (geq, (add, x, y), 3)};
```

```
//^declare check cN, checks if x+y is more than 3
{aN1, {x, {(add, x, 1)}}, {success}}
//^declare action aN1, adds 1 to x (; only on last action)
{aN2, {x, {(sub, y, 1)}}, {success}};
//^declare action aN2, set x to y-1
{a, sequence {{b := aN1}{c := cN}{d := aN2}}};
//^tree structure
;//no specifications
```

Specifications The user may write specifications for the *SBT* using Linear Temporal Logic (LTL), Computational Tree Logic (CTL), or using Invariants over first order logic with standard connectives (and, or, etc.). In the case of LTL and CTL, temporal functions may be used that are otherwise unavailable. We provide a brief overview on LTL, as we use it in Section 7 to specify the desired outcomes and confirm that they occur (or provide a violation).

LTL operates on traces (sequences of states). Let $tr = [s_0, s_1, s_2, ...]$ be a trace for a *FSM*. When considering such a trace, time t refers to s_t. In general, we are interested in whether an LTL formula is true for the entire trace; this is the same as asking if the LTL formula is true at time 0. Below we provide a minimal Grammar 2 and then an overview of the presented functions and some syntactic sugar.

$\langle LTL \rangle ::= \langle a \rangle$ #First Order Logic Formula
| $\neg \langle LTL \rangle \mid \langle LTL \rangle \vee \langle LTL \rangle$ #Boolean operators; in practice we allow more operators
| $\bigcirc(\langle LTL \rangle) \mid (\langle LTL \rangle) \mathscr{U} (\langle LTL \rangle)$ #Temporal operators next and until

Grammar 2: Minimal LTL Grammar.

- $\bigcirc(\phi)$ (next) is true at time t if ϕ is true at time $t+1$.
- $(\phi_1) \mathscr{U} (\phi_2)$ (until) is true at time t if $\exists t''$ such that $t \leq t''$ and ϕ_2 is true at t'' and $\forall t'$ such that $t \leq t' < t''$, ϕ_1 is true at t'.
- $(\phi_1) \mathscr{M} (\phi_2)$ (strong release) is true at time t if $\exists t''$ such that $t \leq t''$ and ϕ_2 is true at t'' and $\forall t'$ such that $t \leq t' \leq t''$, ϕ_1 is true at t'.
- $\square(\phi)$ (globally) is true at time t if $\forall t'$ such that $t \leq t'$, ϕ is true at time t'.
- $\Diamond(\phi)$ (finally) is true at time t if $\exists t'$ such that $t \leq t'$, ϕ is true at time t'.

6 Fastforwarding Execution

Recall the example execution in Figure 1. The execution presented was intuitive and clear, but also highlighted a clear drawback: it took 14 time steps to complete 2 ticks in a 5 node tree. This is not ideal for verification. The total encoding, presented in our previous paper [25], was created to address this issue. The experiments conducted in [25] clearly demonstrated the performance concerns associated with stepping through nodes one at a time and demonstrated that the total encoding is an effective method by which to mitigate this. However, that encoding required the user to edit the resulting model by hand: an arduous task requiring not only expertise in nuXmv but in how BehaVerify encoded the model. We have now addressed this issue (the user need only provide a specification file; everything else is handled automatically) and will explain our solution below.

Fastforwarding and a review of the total encoding The goal is to handle the entire tick in one step, rather than stepping through each node one at a time. To that end, the total encoding represented the status of each node as a function of other nodes. This is shown in Figure 3. Both the encoding and the process of automatically creating the total encoding were part of our prior work. However, our prior work required the user to manually handle the creation of functions such as $user_B$ in nuXmv, along with creating appropriate variable updates in nuXmv. These limitations were the result of:

1. A lack of a DSL. Our prior work handled existing Py Tree objects to create the tree. Unfortunately, this was not conducive to specifying how leaf nodes behave.
2. The complexity of variables in a total encoding. Suppose the variable x is 0 at the start of the tick and 1 at the end; what value of x does the function $user_C$ use?

This is where our new work comes into play. The user uses our DSL to specify leaf nodes, defining what status they return and how they change variables, and BehaVerify takes that information and automatically creates an improved total encoding allowing for the fastforwarding of execution, complete with variable updates and functions for leaf nodes, no additional input from the user required. BehaVerify resolves the issue with variables through the use of variable stages. Each variable has at least one stage. For each possible update to a variable, an additional stage is created representing the variable after the update. Thus, the number of stages a variable has is equal to the number of possible updates to that variable during a single tick plus one. This can be seen in Figure 4. Each stage describes the value of a variable during a portion of the tick; BehaVerify tracks which stage the variable is in and references the appropriate stage in other functions, thus resolving the issue posed above.

Specification Writing We found that fastforwarding often simplifies specification writing. Consider Figure 4. Suppose we want to write that during each tick, (d) returns S (note that this specification is false; during the first tick (d) is I). With fastforwarding, this can be written as an invariant condition, namely $status(d) = S$. If we are not using fastforwarding, we must write this using LTL

$$\square(status(a) = I \implies ((status(a) = I) \mathscr{U} (status(d) = S))).$$

This specification is far more complicated, because we now have to define the duration of a tick. In this case, we accomplish this task by realizing that since (a) is the root, the tick ends when (a) returns. As such, this specification says that it is always the case that if (a) is I, then (a) will stay I until (d) returns S.

Note that sequential properties can also be written with fastforwarding, though may require a little more forethought. For instance, suppose we want to specify that (b) eventually returns S and until that

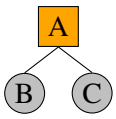


Figure 3:
(L) 3 node tree.
(R) total encoding.
Note: $x?y:z$ means if x, then y, else z.

- act is a function that describes if a node is active or not during a given tick. $status$ is a function that describes the status of a node during a given tick.
- $act(A) \triangleq \top$, $act(B) \triangleq act(A)$, $act(C) \triangleq act(A) \wedge (status(B) = S)$
- $status(A) \triangleq \begin{cases} I \text{ if } \neg act(A) \\ F \text{ if } status(B) = F \vee status(C) = F \\ R \text{ if } status(B) = R \vee status(C) = R \\ S \text{ if } status(B) = S \wedge status(C) = S \end{cases}$
- Let $user_B$ be a function specified by the user that depends on variables and outputs a status. Then $status(B) \triangleq act(B)?user_B(vars) : I$.
- Let $user_C$ be a function specified by the user that depends on variables and outpus a status. Then $status(C) \triangleq act(C)?user_C(vars) : I$.

Tick	1	1	1	1	1	2	2	2	2	2	2	2	2
t	0	1	2	3	4	5	6	7	8	9	10	11	12
Active	a	b	a	c	a	a	b	a	c	a	d	a	a
Returns	-	*S*	-	*F*	*F*	-	*S*	-	*S*	-	*S*	*S*	-
x	0	1	1	1	1	1	2	2	2	2	1	1	1
y	0	0	0	0	0	1	1	1	1	1	1	1	0

Tick	1	2
a	*F*	*S*
b	*S*	*S*
c	*F*	*S*
d	*I*	*S*
x_0	0	1
x_1	1	2
x_2	1	1
y_0	0	1
y_1	1	0

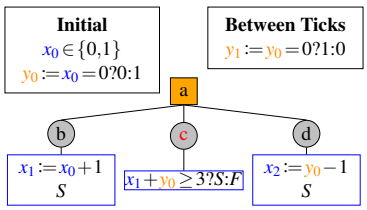


Figure 4: Here x is a blackboard variable and y is an environment variable (an input). The input changes between ticks, so y_1 does not appear in the tree. The upper table shows what stepping through the tree one node at a time would look like. The lower table shows what fastforwarding looks like. The tree itself includes stage subscripts that were added for readability; the user would normally create a tree without knowledge of the stages, as BehaVerify handles stage creation. Note that when fastforwarding, the value of x_2 is set to that of x_1 if (d) does not execute. Furthermore, note that the value of x at the end of the tick (x_2) is the value of x at the start of the next tick (x_0).

happens, (c) does not returns *S*. Since (b) occurs before (c) in the tree, this specification can be written as $status(c) \neq S \mathscr{U} status(b) = S$. The reverse, namely that (c) eventually returns *S* and until that happens (b) does not return *S*, is slightly trickier. One might be tempted to write $status(b) \neq S \mathscr{U} status(c) = S$, but this accepts the case where (b) and (c) both return *S* for the first time on the same tick. To account for this, one should use $\mathscr{M}$ instead of $\mathscr{U}$. In general, this example illustrates the point well; it is entirely possible to describe sequential events when writing specifications for fastforwarded execution, but one must be mindful of the structure of the tree. Finally, it is important to note that some sequential specifications are trivially true, and their verification is not a major objective for BehaVerify. For instance, if (b) and (c) are both active during a tick, then because (b) is before (c) in a depth first traversal of the tree, node (b) was active before node (c). BehaVerify does generate a list of nodes in order of depth first traversal; if one wishes to confirm such specifications, they may consult this order.

Optimizations An immediate concern raised by introducing stages is how this effects model size. After all, if we went from storing a single variable with 10 states to storing 3 variables with 10 states each, then the model is now spending 1000 states on this single variable. Fortunately, through optimizations, we can generally avoid this issue. Specifically, if a variable update is deterministic, then the stage is a function of the previous stage and doesn't increase the model size. Furthermore, even if an update is nondeterministic, we can sometimes avoid an increase in model size. Consider a very simple model with one variable, x, which is updated nondeterministically twice per tick. Then, without optimizations, we would have three stages x_0, x_1, and x_2, and each stage would increase the size of the model. Here x_0 is the value of x at the start of the tick, x_1 the value after the first update, and x_2 the value after the second update. Note that the next value of x_0 is the current value of x_2. This is true for all variables; the next value of stage 0 is equal to the current value of the last stage. Furthermore, each other stage depends only on current values. In this example, since nothing depends on the value of x_2 other than the next value of x_0, we can safely remove the last stage and simply make it so the next value of x_0 is the value that x_2 would have been assigned. It is important to note that if a user writes a specification that checks the value of x at the end of a tick, this optimization *would* change the result; we detect such cases and automatically disable the optimization for the variable. Furthermore, if another variable or node depends on the value of x_2, this optimization would

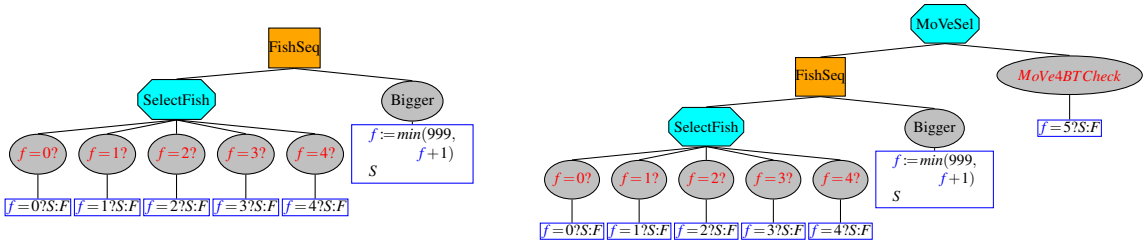


Figure 5: Left is original, right is reworked for MoVe4BT. Number of nodes is counted using right tree (10 in this case). f is the size of the 'biggest fish' so far. Each check returns S if the f is equal to its number and F otherwise. If any check returns S, the left half of the tree returns S and f is incremented.

change the result and would therefore be disabled. In such cases, we can instead try to combine the 0 stage with the 1 stage using a similar process, with similar caveats. We can attempt this with the first and last stage of each variable.

7 Verification Results for Stateful Behavior Trees

Here we present formal verification results and include comparisons to MoVe4BT [23] that demonstrate BehaVerify scales significantly better in the size of the tree and overall state space. Additionally, we present an interesting example demonstrating that BehaVerify is capable of finding complex counterexamples. All results were generated on a Dell Inc. OptiPlex 7040 with 64 GiB of Memory with an Intel i7–6700 CPU @ 3.40GHz with 8 cores. The code used and instructions for reproducing the results are available [2].

We note here that BehaVerify takes a specification file written using the DSL and produces a nuXmv model. The timing results for BehaVerify are based *solely* on the time it takes nuXmv to run on the generated model. We do not include the time it takes for BehaVerify to generate the nuXmv model. This is because we wanted to compare our encoding to that of the competing tool, MoVe4BT, and we felt this was best demonstrated through a comparison of the model checking aspect. However, we also note that the compile times do not meaningfully change the outcome of the results; compiling the simple robot experiment takes fractions of a second even with a 30 by 30 grid and the same is true for the bigger fish experiment with 1000 nodes.

Scaling Tree Experiment: Bigger Fish The bigger fish experiment (see Figure 5) scales the tree while the blackboard and environment are unchanged. f is an integer between 0 and 1000 inclusive and is initially 0. The upper limit was increased for the tests on 10000 and 20000 nodes. We intended to verify $\Diamond(\Box(f=n))$ (variable attains and maintains a value), but MoVe4BT does not support LTL specifications over variables; you are restricted to checking the statuses of nodes. We modified the tree and created a node (named *MoVe4BTCheck*) to check the value of the variable (see Figure 5 for details). We then tried to verify $\Diamond(\Box(MoVe4BTCheck=S))$. In BehaVerify, this specification states that eventually, during each tick the check returns S. However, because MoVe4BT does not utilize fastforwarding (see Section 6 for details), MoVe4BT interprets this to mean that eventually the check is always active, which is false. We instead had to settle for verifying $\Diamond(MoVe4BTCheck=S)$. This means that instead of verifying that the variable attains the value and maintains it, MoVe4BT only verifies that it attains it. It is crucial to

[2] https://github.com/verivital/behaverify/tree/main/REPRODUCIBILITY/2024_FMAS_SBT

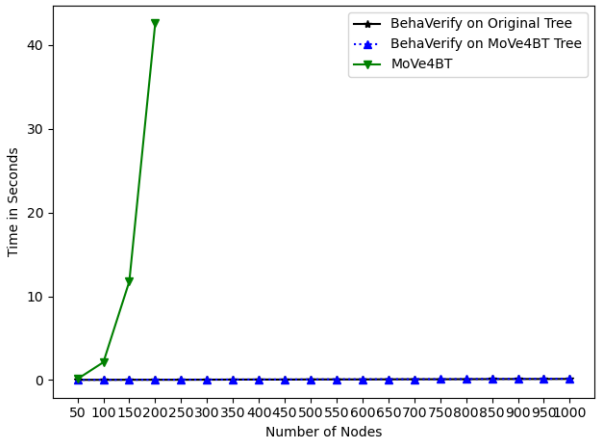


Figure 6: Time to verify $\Diamond(\Box(f=n))$ (original) and $\Diamond(f=n)$ (changed) where n is the number of nodes minus 5. Starting at 250 nodes total, MoVe4BT ran for over a minute, producing a blank screen with no results; we interpret this as a timeout. We ran BehaVerify with 10000 and 20000 nodes, taking 8.20 and 32.32 seconds. At 200 and 20000 nodes, BehaVerify reported 196 and 19996 reachable states. BehaVerify reports similar runtimes for both trees.

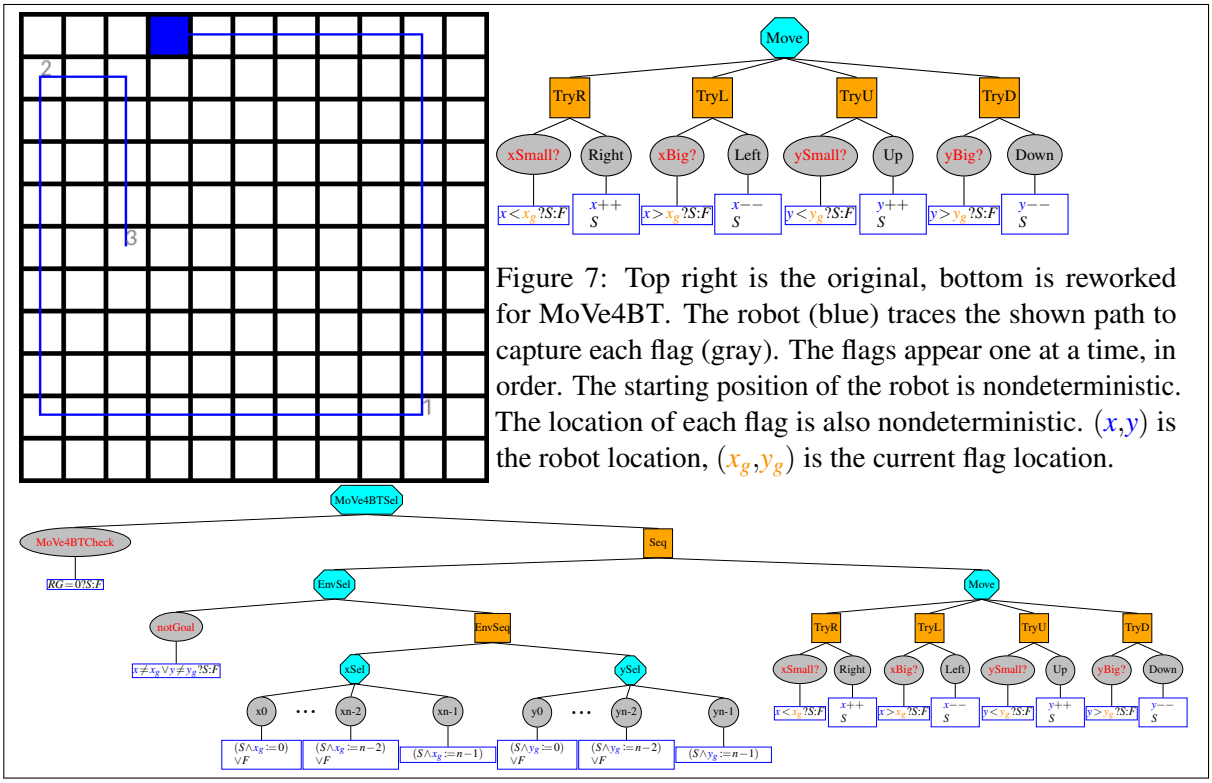


Figure 7: Top right is the original, bottom is reworked for MoVe4BT. The robot (blue) traces the shown path to capture each flag (gray). The flags appear one at a time, in order. The starting position of the robot is nondeterministic. The location of each flag is also nondeterministic. (x,y) is the robot location, (x_g,y_g) is the current flag location.

stress that BehaVerify is capable of verifying the original condition; in fact, we include timing results for both the original and new condition. Figure 6 shows BehaVerify scales well with tree complexity, while MoVe4BT does not. The specification is true. If the model is changed so it is false (by removing a leaf in the chain), both tools produce a counterexample.

Scaling Blackboard Experiment: Simple Robot The simple robot experiment (see Figure 7) scales the blackboard while the tree is constant. A robot on a n by n board tries to reach a goal. Once reached, a new goal is generated. We verify that eventually 3 goals are reached. The experiment scales by increasing n from 2 to 30 in increments of 4. We compare to MoVe4BT for this experiment. MoVe4BT has no

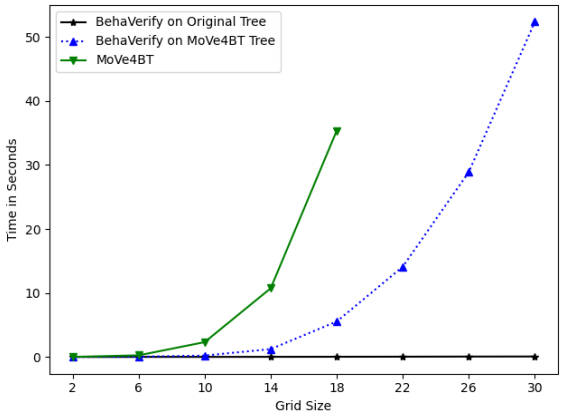


Figure 8: Time in seconds to verify that finally, there are no remaining goals: $\Diamond(RG = 0)$. Our tool ran on both the original tree and the changed tree. Starting at a 22 by 22 grid, MoVe4BT ran for over a minute and then produced a blank screen; we interpreted this as a timeout. At 18 by 18, BehaVerify reported about 2^{19} reachable states. At 30 by 30, BehaVerify reported about 2^{22} reachable states.

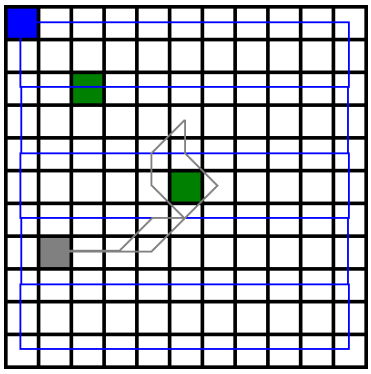


Figure 9: The target (gray) is able to avoid the drone (blue) by circling a tree (green) that obstructs vision but not movement. The target's initial position and movement are nondeterministic; the drone's are not. Note that this was a counterexample generated by nuXmv from the generated model.

concept of an environment so the environment has to become part of the *BT*. Furthermore, MoVe4BT cannot assign values to variables nondeterministically; instead MoVe4BT nodes can nondeterministically choose one of *S*, *R*, *F*, and a single accompanying 'program' with updates. Thus to achieve the effect of nondeterminism, we had to use a series of nodes. Each node could nondeterministically choose to change the value or not. Thus the tree is more complex and also grows in size with the blackboard. Finally, we had to introduce an extra node (called *MoVe4BTCheck*) so that MoVe4BT could verify the condition at all. Refer to Figure 7 for a visual comparison of the trees. See Figure 8 for timing results.

Moving Target Experiment For this experiment, a drone is searching for a mobile target on a grid. The drone has limited vision, which is further obscured by trees (the trees do not prevent movement). We conduct two experiments for this setup: one where the target can move every 5 turns and one where the target can move every 10 turns. When the target can move every 10 turns, we verify that eventually the drone finds the target. When the target can move every 5 turns, we generate a counterexample (see Figure 9). In both cases, the specification being verified is that we eventually 'see' the target. Given the complexity of encoding the simple robot environment for MoVe4BT and the fact that this experiment is more complex and nondeterministic, we did not attempt to recreate it in MoVe4BT.

Reasoning About Results We believe fastforwarding (see Section 6) is the main reason BehaVerify outperforms MoVe4BT. Based on the traces that MoVe4BT produces, we believe that MoVe4BT jumps from leaf to leaf during execution. This provides a performance boost over stepping through every single node in the tree, but it is not enough. Consider the bigger fish experiment. With 20000 nodes,

there are 19997 leaf nodes. During the first tick, 2 leaf nodes will be active. During the second tick, there will be 3. Finally, there will be 19996 active leafs. Thus, MoVe4BT would have to step through $2+3+\cdots+19996=199{,}930{,}005$ leaf nodes resulting in a very long trace. By comparison, the BehaVerify trace would have less than 20000 states. Next, we consider the simple robot experiment, where changing the tree had a huge impact. This is because MoVe4BT does not support nondeterministic variable assignments; as such we had to create a series of nondeterministic nodes. Thus instead of having one nondeterministic update for x and one for y, we had $n-1$ for each. This caused the number of total states to jump from 2^{23} to 2^{80} while the number of reachable states was relatively unchanged. The changed tree is close to a worst case scenario for BehaVerify; many variable updates, all of them nondeterministic. Even in this worst case scenario, BehaVerify significantly outperformed MoVe4BT.

8 Conclusions and Future Work

We introduced and formally defined *SBT*s and demonstrated they are Turing Complete under certain assumptions. We presented a DSL for specifying *SBT*s implemented in BehaVerify. Our experiments demonstrate BehaVerify can complete a verification task on a tree with 20000 nodes in the time that MoVe4BT, a different verification tool, verifies a property on a tree with 200 nodes, demonstrating two orders of magnitude scalability improvement. Potential future work includes improving our encoding of array variables, developing a graphical user interface for the creation of *SBT*s, visualization of counterexamples provided by nuXmv, general performance improvements, and expanding support for unbounded variable types in nuXmv for bounded model checking (BMC).

Acknowledgements

The material presented in this paper is based upon work supported by the National Science Foundation (NSF) through grant numbers 2220426 and 2220401, the Defense Advanced Research Projects Agency (DARPA) under contract number FA8750-23-C-0518, and the Air Force Office of Scientific Research (AFOSR) under contract numbers FA9550-22-1-0019 and FA9550-23-1-0135. This paper was also supported in part by a fellowship award under contract FA9550-21-F-0003 through the National Defense Science and Engineering Graduate (NDSEG) Fellowship Program, sponsored by the Air Force Research Laboratory (AFRL), the Office of Naval Research (ONR), and the Army Research Office (ARO). Any opinions, findings, and conclusions or recommendations expressed in this paper are those of the authors and do not necessarily reflect the views of AFOSR, DARPA, or NSF.

References

[1] Auryn Robotics: *Tutorial 02: Blackboard and Ports*. Available at https://www.behaviortree.dev/docs/tutorial-basics/tutorial_02_basic_ports.

[2] Oliver Biggar & Mohammad Zamani (2020): *A Framework for Formal Verification of Behavior Trees With Linear Temporal Logic*. *IEEE Robotics and Automation Letters* 5(2), pp. 2341–2348, doi:10.1109/LRA.2020.2970634.

[3] Oliver Biggar, Mohammad Zamani & Iman Shames (2020): *A principled analysis of Behavior Trees and their generalisations*. *CoRR* abs/2008.11906, doi:10.48550/arXiv.2008.11906. arXiv:2008.11906.

[4] Oliver Biggar, Mohammad Zamani & Iman Shames (2021): *An Expressiveness Hierarchy of Behavior Trees and Related Architectures*. *IEEE Robotics and Automation Letters* 6(3), pp. 5397–5404, doi:10.1109/lra.2021.3074337.

[5] Roberto Cavada, Alessandro Cimatti, Michele Dorigatti, Alberto Griggio, Alessandro Mariotti, Andrea Micheli, Sergio Mover, Marco Roveri & Stefano Tonetta (2014): *The nuXmv Symbolic Model Checker*. In: *CAV*, pp. 334–342. Available at http://dx.doi.org/10.1007/978-3-319-08867-9_22.

[6] Michele Colledanchise, Giuseppe Cicala, Daniele E. Domenichelli, Lorenzo Natale & Armando Tacchella (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)*, IEEE Press, pp. 9841–9848, doi:10.1109/IROS51168.2021.9636129.

[7] Michele Colledanchise & Petter Ögren (2014): *How Behavior Trees modularize robustness and safety in hybrid systems*. In: *2014 IEEE/RSJ International Conference on Intelligent Robots and Systems*, pp. 1482–1488, doi:10.1109/IROS.2014.6942752.

[8] Michele Colledanchise & Petter Ögren (2016): *How Behavior Trees generalize the Teleo-Reactive paradigm and And-Or-Trees*. In: *2016 IEEE/RSJ International Conference on Intelligent Robots and Systems (IROS)*, pp. 424–429, doi:10.1109/IROS.2016.7759089.

[9] Michele Colledanchise & Petter Ögren (2017): *How Behavior Trees Modularize Hybrid Control Systems and Generalize Sequential Behavior Compositions, the Subsumption Architecture, and Decision Trees*. *IEEE Transactions on Robotics* 33(2), pp. 372–389, doi:10.1109/TRO.2016.2633567.

[10] Alessio De Luca, Luca Muratore & Nikos G. Tsagarakis (2023): *Autonomous Navigation With Online Replanning and Recovery Behaviors for Wheeled-Legged Robots Using Behavior Trees*. *IEEE Robotics and Automation Letters* 8(10), pp. 6803–6810, doi:10.1109/LRA.2023.3313052.

[11] I. Dejanović, R. Vaderna, G. Milosavljević & Z. Vuković (2017): *TextX: A Python tool for Domain-Specific Languages implementation*. *Knowledge-Based Systems* 115, pp. 1–4, doi:10.1016/j.knosys.2016.10.023. Available at https://www.sciencedirect.com/science/article/pii/S0950705116304178.

[12] Alexandre Duret-Lutz, Etienne Renault, Maximilien Colange, Florian Renkin, Alexandre Gbaguidi Aisse, Philipp Schlehuber-Caissier, Thomas Medioni, Antoine Martin, Jérôme Dubois, Clément Gillard & Henrich Lauko (2022): *From Spot 2.0 to Spot 2.10: What's New?* In: *Proceedings of the 34th International Conference on Computer Aided Verification (CAV'22)*, *Lecture Notes in Computer Science* 13372, Springer, pp. 174–187, doi:10.1007/978-3-031-13188-2_9.

[13] EpicGames (2021): *Behavior tree overview*. Available at https://docs.unrealengine.com/4.27/en-US/InteractiveExperiences/ArtificialIntelligence/BehaviorTrees/BehaviorTreesOverview/.

[14] Sergio García, Patrizio Pelliccione, Claudio Menghi, Thorsten Berger & Tomas Bures (2020): *PROMISE: High-Level Mission Specification for Multiple Robots*. In: *Proceedings of the ACM/IEEE 42nd International Conference on Software Engineering: Companion Proceedings*, ICSE '20, Association for Computing Machinery, New York, NY, USA, pp. 5–8, doi:10.1145/3377812.3382143.

[15] Razan Ghzouli, Thorsten Berger, Einar Broch Johnsen, Andrzej Wasowski & Swaib Dragule (2023): *Behavior Trees and State Machines in Robotics Applications*. *IEEE Transactions on Software Engineering*, pp. 1–24, doi:10.1109/TSE.2023.3269081.

[16] Zhaoyuan Gu, Nathan Boyd & Ye Zhao (2022): *Reactive Locomotion Decision-Making and Robust Motion Planning for Real-Time Perturbation Recovery*. In: *2022 International Conference on Robotics and Automation (ICRA)*, pp. 1896–1902, doi:10.1109/ICRA46639.2022.9812068.

[17] Thomas Henn, Marcus Völker, Stefan Kowalewski, Minh Trinh, Oliver Petrovic & Christian Brecher (2022): *Verification of Behavior Trees using Linear Constrained Horn Clauses*. In Jan Friso Groote & Marieke Huisman, editors: *Formal Methods for Industrial Critical Systems*, Springer International Publishing, Cham, pp. 211–225, doi:10.1007/978-3-031-15008-1_14.

[18] Qian Huang, Xianming Ma, Kun Liu, Xinyi Ma & Weijian Pang (2022): *Autonomous Reconnaissance Action of Swarm Unmanned System Driven by Behavior Tree*. In: *2022 IEEE International Conference on Unmanned Systems (ICUS)*, pp. 1540–1544, doi:10.1109/ICUS55513.2022.9986758.

[19] Matteo Iovino, Edvards Scukins, Jonathan Styrud, Petter Ögren & Christian Smith (2022): *A survey of Behavior Trees in robotics and AI*. *Robotics and Autonomous Systems* 154, p. 104096, doi:10.1016/j.robot.2022.104096.

[20] Seungwoo Jeong, Taekwon Ga, Inhwan Jeong & Jongeun Choi (2022): *Behavior Tree-Based Task Planning for Multiple Mobile Robots using a Data Distribution Service*. In: *2022 IEEE/ASME International Conference on Advanced Intelligent Mechatronics (AIM)*, pp. 1791–1798, doi:10.1109/AIM52237.2022.9863364.

[21] Alejandro Marzinotto, Michele Colledanchise, Christian Smith & Petter Ögren (2014): *Towards a unified behavior trees framework for robot control*. In: *2014 IEEE International Conference on Robotics and Automation (ICRA)*, pp. 5420–5427, doi:10.1109/ICRA.2014.6907656.

[22] Leonardo de Moura & Nikolaj Bjørner (2008): *Z3: An Efficient SMT Solver*. In C. R. Ramakrishnan & Jakob Rehof, editors: *Tools and Algorithms for the Construction and Analysis of Systems*, Springer Berlin Heidelberg, Berlin, Heidelberg, pp. 337–340, doi:10.1007/978-3-540-78800-3_24.

[23] Huang Peishan, Hong Weijiang, Chen Zhenbang & Wang Ji: *MoVe4BT: Modeling & Verification For BT*. Available at https://move4bt.github.io/. Accessed: 2023-12-14.

[24] Fangbo Qin, De Xu, Blake Hannaford & Tiantian Hao (2023): *Object-Agnostic Vision Measurement Framework Based on One-Shot Learning and Behavior Tree*. *IEEE Transactions on Cybernetics* 53(8), pp. 5202–5215, doi:10.1109/TCYB.2022.3181054.

[25] Serena S. Serbinowska & Taylor T. Johnson (2022): *BehaVerify: Verifying Temporal Logic Specifications For Behavior Trees*. In: *Software Engineering and Formal Methods: 20th International Conference, SEFM 2022, Berlin, Germany, September 26-30, 2022, Proceedings*, Springer-Verlag, Berlin, Heidelberg, pp. 307–323, doi:10.1007/978-3-031-17108-6_19.

[26] Christopher I. Sprague & Petter Ögren (2022): *Continuous-Time Behavior Trees as Discontinuous Dynamical Systems*. *IEEE Control Systems Letters* 6, pp. 1891–1896, doi:10.1109/LCSYS.2021.3134453.

[27] Daniel Stonier: *PyTrees Module API*. Available at https://py-trees.readthedocs.io/en/devel/modules.html. Accessed: 2023-12-14.

[28] Petter Ögren (2020): *Convergence Analysis of Hybrid Control Systems in the Form of Backward Chained Behavior Trees*. *IEEE Robotics and Automation Letters* 5(4), pp. 6073–6080, doi:10.1109/LRA.2020.3010747.

[29] Petter Ögren & Christopher I. Sprague (2022): *Behavior Trees in Robot Control Systems*. *Annual Review of Control, Robotics, and Autonomous Systems* 5(Volume 5, 2022), pp. 81–107, doi:10.1146/annurev-control-042920-095314.