BehaVerify: Verifying Temporal Logic Specifications for Behavior Trees

Bernard Serbinowski[1][0000−0002−9259−1586] and
Taylor T. Johnson[1][0000−0001−8021−9923]

Vanderbilt University, Nashville TN 37235, USA
{bernard.serbinowski,taylor.johnson}@vanderbilt.edu

Abstract. Behavior Trees, which originated in video games as a method for controlling NPCs but have since gained traction within the robotics community, are a framework for describing the execution of a task. BehaVerify is a tool that creates a nuXmv model from a py_tree. For composite nodes, which are standardized, this process is automatic and requires no additional user input. A wide variety of leaf nodes are automatically supported and require no additional user input, but customized leaf nodes will require additional user input to be correctly modeled. BehaVerify can provide a template to make this easier. BehaVerify is able to create a nuXmv model with over 100 nodes and nuXmv was able to verify various non-trivial LTL properties on this model, both directly and via counterexample. The model in question features parallel nodes, selector, and sequence nodes. A comparison with models based on BTCompiler indicates that the models created by BehaVerify perform better.

Keywords: Behavior Tree · Model Verification

1 Introduction

Behavior Trees are a framework for describing the execution of a task that originated in computer games as a method of controlling Non-Player Characters (NPCs), but have since expanded into the domain of robotics [14] [26]. Behavior Trees are split into composite nodes that control the flow through the tree and leaf nodes which execute actions. Behavior Trees have a variety of strengths: they facilitate code re-use (nodes and sub-trees can easily be attached), their modular nature makes reasoning about them easier, and changing one region of a tree doesn't affect how other regions function [1]. However, at present, tools to verify the correctness of a Behavior Tree are scarce. Therefore, we present BehaVerify, a tool for converting a py_tree into a .smv file which can be verified using nuXmv [6].

Contributions. We present BehaVerify, a tool that enables verification with Linear Temporal Logic (LTL) model checking that improves upon BTCompiler, the only previously existing tool for such a task, in terms of run time and in ease of use with respect to Blackboard variables. Specifically, we present an automatic

method to perform the translation and encoding of behavior trees to nuXmv models, a description of this method in a publicly available software tool, a characterization of the verification performance of these different encodings and how they compare to the models created by BTCompiler, and apply the tool to verify key LTL specifications of a challenging robotics case study for an underwater robot used as a controller in an ongoing DARPA project. However, we first define what Behavior Trees are.

1.1 Background

A Behavior Tree (BT) is a rooted tree. Each node has a single parent, save for the root which has no parent. A BT does nothing until it receives a tick event, at which point the tick event propagates throughout the tree. Composite nodes serve to control the flow of execution, determining which children receive tick events. By contrast, Leaf nodes are either actions, such as Accelerate, or guard checks, such as GoingToSlow. Leaf nodes do not have children. Finally, decorator nodes are used to customize the output of their children without actually modifying the children themselves, allowing for greater re-usability. Usually, a Decorator node will have one child.

There are three types of composite nodes: Sequence, Selector, and Parallel. Sequence nodes execute a sequence of children. A Sequence node returns a value if a child returns Failure or Running or there are no more children to run. Sequences return Failure if any child returns Failure, Running if any child returns Running, and Success if every child returned Success. Selector nodes, also known as Fallback nodes [1] [24], execute children in order of priority. A Selector node returns a value if a child returns Success or Running or there are no more children to run. Selectors return Success if any child returns Success, Running if any child returns Running, and Failure if every child returned Failure.

Parallel nodes execute all their children regardless of what values are returned. At least three different definitions exist for parallel nodes. The first definition, found in [24], states that parallel nodes return Failure if any child returns Failure, Success if a satisfactory subset of children return Success, and Running otherwise. The second definition, found in [11], [10], and [18] is similar, but states that parallel nodes return Success only if all children return Success. The third definition, found in [2], [25], [20], [14], [12], and [13], states that parallel nodes return Success if at least m children return Success, Failure if $n-m+1$ children return Failure, and Running otherwise. Here n is the number of children the parallel node has and m is a node parameter. BehaVerify, the tool created alongside this paper, was designed for py_trees and therefore utilizes the definition presented in [24].

In addition to these differences, Composite nodes can be further characterized into Nodes with Memory and Nodes without Memory. The above definitions describe Nodes without Memory. Nodes with Memory allow the composite nodes to remember what they previously returned and continue accordingly. Thus a Sequence with Memory will not start from its first child if it previously returned Running and will instead skip over each child that returned Success. Similarly, a

Selector with Memory will skip over each child that returned Failure. A Parallel node with Memory will only rerun children that returned Running.

However, memory is also not standardized. In [24], Nodes with Memory 'forget' if one of their ancestors returns Success or Failure. So, for instance, if a Sequence with Memory returns Running, but its Parallel node parent returns Success, the Sequence with Memory will not behave as though it returned Running. However, in section 1.3.2 of [14], the authors state "Control flow nodes with memory always remember whether a child has returned Success or Failure, avoiding the re-execution of the child until the whole Sequence or Fallback finishes in either Success or Failure", and notably makes no mention of Parallel nodes with Memory. Finally, note that py_trees supports Selector with and without Memory, Sequences with and without Memory, and both types of Parallel nodes. However, the Parallel nodes with Memory and without Memory are instead called Synchronized Parallel and Unsynchronized Parallel, respectively.

Decorator nodes are generally used to augment the output of a child. For instance, a RunningIsFailure decorator will cause an output of Running to be interpreted as Failure. As there are many decorators, we omit attempting to fully list or describe them here.

Furthermore, we note that in many of the above works, Selector nodes are represented using ?, Sequence nodes are represented using $\rightarrow$, and Parallel nodes are represented using $\rightrightarrows$. However, we will utilize the notation given in py_trees, as seen in Figure 1.

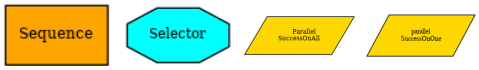

Fig. 1. Composite Nodes in py_trees.

1.2 The Blackboard

In certain situations, such as when multiple nodes need to use the result of a computation, it can be useful to read and write information in a centralized location. This sort of shared memory is frequently called a Blackboard [24] [5] [16] [15]. Unfortunately, there are also drawbacks to using Blackboards. As [23] points out, Blackboards can make BTs difficult to understand and reduce subtree reuse. Ultimately, however, the fact remains that in many cases there are substantial benefits to using a Blackboard, and various implementations, such as py_trees seek to alleviate some of the aspects by creating visualization tools for blackboards [24]. Accordingly, BehaVerify supports Blackboard variables.

2 Related Work

First, we clarify that the term "Behavior Tree" sometimes refer to different concepts. Behavior Trees exist as a formal graphical modeling language, as part of

Behavior Engineering and are used for requirement handling [19]. These are not the BTs we are talking about.

2.1 Strengths and Uses of BTs

In [20], the author shows how general Hybrid Dynamical Systems can be written as BTs and how this can be beneficial. Furthermore, the paper provides justifications for why BTs are useful to UAV guidance and control systems. [4] compares BTs to a variety of other Action Selection Mechanisms (ASM) and proves that unrestricted BTs have the same expressive capabilities as unrestricted Finite State Machines. [1] presents a framework for verifying the correctness of BTs without compromising on the main strengths of Behavior Trees, which they identify as modularity, flexibility, and re-usability.

[17] considers the various implementations of BTs, such as BehaviorTree.cpp and py_trees, and examines a variety of repositories that utilize BTs. In [25] the authors propose an algorithm to translate an I/O automaton into a BT that connects high level planning and low level control. The authors of [9] demonstrate how it is possible to synthesize a BT that is guaranteed to be complete a task specified by LTL. This does require restricting LTL to a fragment of LTL, so there are limits to what BTs can be synthesized in this way. [8] describes a tool-chain for designing, executing, and monitoring robots that uses BTs for controlling high level behaviors of the robots while [7] formalizes the context within which BTs are executed.

2.2 Expanded BTs

The capabilities of BTs have been expanded in several papers. In [3], the authors consider how it is possible to expand BTs by introducing K-BTs which replace Success and Failure with K different outputs. [10], [11], and [12] introduce Concurrent BTs and expand on them by introducing various nodes designed to better enable synchronization in BTs that deal with concurrency. Meanwhile [18] extends BTs to Conditional BTs, which enforce certain pre and post conditions on various nodes within the tree and introduces a tool which can confirm that the entire tree is capable of being executed based on the pre and post conditions given. [21] extends BTs to Belief BTs which are better suited to dealing with non-deterministic outcomes of actions.

2.3 Verification of BTs

Some of the above works deal with the verification of BTs. [1], for instance, presents an algorithm for the verification of BTs. [9], on the other hand, presents a method by which to synthesize a BT that is guaranteed to be correct, thereby by-passing the need for verification, but the specifications are limited to a fragment of LTL. The only existing tool we were able to find that allows the user to

create and verify LTL specifications for BTs is BTCompiler [1]. Unfortunately, we were not able to install the tool, and as such our knowledge of it is somewhat limited. Most of what we understand comes from analyzing the various examples in the smv folder in the github repository.

From what we understand, BTCompiler uses the following assumptions and definitions. All composite nodes are assumed to have exactly 2 children. Parallel nodes do not have memory. Parallel nodes utilize the third definition presented in the background section. Sequence and Selector nodes with and without memory are supported. Unlike the implementation in py_trees, nodes with memory do not 'forget' if an ancestor terminates. Please note that the requirement that composite nodes have only 2 children does not impact expressiveness. By self-composing nodes, it is possible to effectively create a node with any number of children greater than 2. For a proof, see section 5.1 of [14]. Thus the only downside is potential model complexity and readability.

We will compare the models created by BTCompiler and BehaVerify.

3 Overview of Approach

BehaVerify begins by recursively walking a py_tree and recording relevant information. This information includes what the type of each node is, recording any important parameters (like the Success policy for a parallel node), and the structure of the tree. Once this process has finished, BehaVerify begins to create the .smv file. Most of this process is straightforward. For instance, for each node type, BehaVerify creates a module (basically a class) in the .smv file. These modules are static and don't change between runs. For each node, BehaVerify creates an instance of a module with the necessary parameters, like what children the node has.

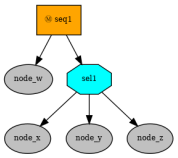

Fig. 2. A simple BT

However, not everything is simple or static. The primary sources of complexity are Nodes with Memory. A lazy approach to implementing Nodes with Memory is to have each node store an integer marking which child returned Running. Such an encoding can greatly increase the number of states in the model. Consider Figure 2. Seq1 has two children, while sel1 has three. The lazy encoding would therefore produce six states to record which children returned

[1] https://github.com/CARVE-ROBMOSYS/BTCompiler

Running. However, consider that if we know that node_y returned Running, then sel1 will also return Running. Thus we only need four states.

Next, BehaVerify begins to handle the blackboard. BehaVerify has several ways of doing this. The first method is to have the user provide an input file which is simply included in the .smv file. Assuming no such file is provided, BehaVerify can generate the blackboard. If the user requests, the generated blackboard can be saved. This allows the user to modify the generated blackboard file and use it as an input file on subsequent runs. In addition, BehaVerify also allows the user to specify a file containing LTL specifications which are then included in the .smv file.

At this point, the .smv file is complete, and can be used with nuXmv [6], either for simulation or verification.

4 Encodings

BehaVerify uses two primary encodings: Leaf and Total. The general ideas behind these encodings are presented here. Note that the actual models BehaVerify creates for use with nuXmv differ from what is presented here, but the general motivations are the same. Also note that from this point forward, we write Success as S, Failure as F, Running as R, and Invalid as I. For both encodings, it is useful to consider how a BT operates. A BT remains inactive until it receives a tick. Once a tick is received, it begins to propagate throughout the tree causing various nodes to execute. The path of the tick signal through the Tree is similar to a Depth First Search, though it will sometimes skip over branches of the tree. A basic version of the Leaf encoding explicitly follows the tick signal as it moves throughout the tree, tracing the exact path the tick signal takes through the tree. The Leaf encoding presented here includes some optimizations to improve performance, but the general idea is the same. The Total encoding doesn't follow the path of the signal. The state of the tree in the Total encoding is instead represented by a chain of dependencies and by considering the path of the tick signal through the tree, the chain can be resolved. Additional details follow.

4.1 Leaf

Fig. 3. A selector node with many children.

As was mentioned, an intuitive encoding for BTs follows the path of the tick throughout the tree. At each time step t, one node is the Active Node ($ActNode(t)$), its status is computed, and then another node becomes Active.

Note that in this encoding each time step t does NOT correspond to a tick. A tick instead occurs between any time steps t and $t+1$ such that $\mathit{ActNode}(t) = -1$. Now consider Figure 3. In this simple encoding, we would start at wideSel, then move to child1, then back to wideSel, then to child2, back to wideSel, etc., until one of the children returned S or R, or we ran out of children. Thus this encoding spends many steps going through wideSel. The Leaf encoding realizes that the actual points of interest are the leaf nodes themselves. If child1 returns S or R, then the tree returns a status. If child1 returns F, then we need to check child2. Thus we can eliminate many unnecessary steps in the traversal of the tree by jumping from leaf to leaf. Formally, this encoding is as follows:

$$\mathit{ActNode}(t+1) \coloneqq \begin{cases} \text{if } t+1 \leq 0, \text{ then } -1 \\ \text{else if } \mathit{ActNode}(t) = -1, \text{ then } \mathit{NextNode}(\mathit{root}, t, -1) \\ \text{else } \mathit{NextNode}(\mathit{ActNode}(t), t, \mathit{ActNode}(t)) \end{cases}$$

So at each time step t, $\mathit{ActNode}(t)$ either indicates a Node that is active or returns -1, which symbolizes the tree returning a value. In $\mathit{NextNode}(n, t, \mathit{prev})$, n is either -1 or a node, t is an integer indicating the time-step, and prev is either -1 or a node and indicates which node asked for the Next Node. This is used to determine which node should be active next.

$$\mathit{NextNode}(n, t, \mathit{prev}) \coloneqq$$

$$\begin{cases} \text{if } n = -1, \text{ then } -1 \\ \text{else if } \mathit{status}(n,t) \neq I, \text{ then } \mathit{NextNode}(\mathit{parent}(n), t, n) \\ \text{else if } \mathit{IsLeaf}(n), \text{ then } n \\ \text{else if } \mathit{IsSel}(n) \wedge \mathit{prev} = \mathit{parent}(n), \\ \quad \text{then } \mathit{NextNode}(\mathit{Unskipped}(\mathit{FChl}(n), t), t, n) \\ \text{else if } \mathit{IsSel}(n), \text{ then } \mathit{NextNode}(\mathit{rNeigh}(\mathit{prev}), t, n) \\ \text{else if } \mathit{IsSeq}(n) \wedge \mathit{prev} = \mathit{parent}(n), \\ \quad \text{then } \mathit{NextNode}(\mathit{Unskipped}(\mathit{FChl}(n), t), t, n) \\ \text{else if } \mathit{IsSeq}(n), \text{ then } \mathit{NextNode}(\mathit{rNeigh}(\mathit{prev}), t, n) \\ \text{else if } \mathit{IsPar}(n) \wedge \mathit{prev} = \mathit{parent}(n), \\ \quad \text{then } \mathit{NextNode}(\mathit{Unskipped}(\mathit{FChl}(n), t), t, n) \\ \text{else if } \mathit{IsPar}(n), \text{ then } \mathit{NextNode}(\mathit{Unskipped}(\mathit{prev}, t), t, n) \\ \text{else if } \mathit{IsDec}(n) \wedge \mathit{SkipChl}(n, t), \text{ then } n \\ \text{else } \mathit{NextNode}(\mathit{FChl}(n), t, n) \end{cases}$$

$\mathit{parent}(\mathit{Root}) = -1$ and otherwise $\mathit{parent}(n)$ returns the parent of n.
$\mathit{SkipChl}(n, t)$ returns True if at time t decorator n does not run it's child. $\mathit{IsLeaf}(n)$, $\mathit{IsSel}(n)$, $\mathit{IsSeq}(n)$, $\mathit{IsPar}(n)$, and $\mathit{IsDec}(n)$ are all predicates that return True if the node n is of the described type and False otherwise (all return

False if $n = -1$). $FChl(n)$ returns the first child of n, and $rNeigh(n)$ indicates the right neighbor of n.

$$Unskipped(n,t) \coloneqq \begin{cases} \text{if } Skipped(n,t), \text{ then } Unskipped(rNeigh(n),t) \\ \text{else } n \end{cases}$$

$Unskipped(n,t)$ returns the first right Neighbor of n that is not Skipped (Nodes with Memory can cause their children to be skipped in some cases). If there is no right neighbor, then $rNeigh(n) = -1$.

$$Skipped(n,t) \coloneqq \begin{cases} \text{if } t \leq 0, \text{ then } \bot \\ \text{else if } \exists a \in Anc(n) \text{ s.t. } status(a, t-1) \in \{S, F\}, \\ \quad \text{then } \bot \\ \text{else if } IsParSynch(parent(n)) \wedge status(n, t-1) = S, \\ \quad \text{then } \top \\ \text{else if } IsSeqWM(parent(n)) \wedge \\ \quad \exists x \geq 1 \text{ s.t. } status(rNeigh(n)^x, t-1) = R, \text{ then } \top \\ \text{else if } IsSelWM(parent(n)) \wedge \\ \quad \exists x \geq 1 \text{ s.t. } status(rNeigh(n)^x, t-1) = R, \text{ then } \top \\ \text{else } Skipped(n, t-1) \end{cases}$$

Here $rNeigh(n)^x \coloneqq rNeigh(rNeigh(n)^{x-1})$, with $rNeigh(n)^1 \coloneqq rNeigh(n)$. In other words, $rNeigh(n)^x$ is the x^{th} right neighbor. $Anc(n)$ is the set of nodes that are ancestors to n. This set does not include n or -1. $IsSeqWM(n)$ and $IsSelWM(n)$ check if n is a Sequence/Selector node with memory, respectively.

$$status(n,t) \coloneqq$$

$$\begin{cases} \text{if } IsLeaf(n) \wedge ActNode(t) = n, \text{ then } LeafStatus(n,t) \\ \text{else if } IsSel(n) \wedge (\exists c \in Chl(n) \text{ s.t. } status(c,t) \in \{S, R\}), \text{ then } status(c,t) \\ \text{else if } IsSel(n) \wedge status(LChl(n), t) = F, \text{ then } F \\ \text{else if } IsSeq(n) \wedge (\exists c \in Chl(n) \text{ s.t. } status(c,t) \in \{F, R\}), \text{ then } status(c,t) \\ \text{else if } IsSeq(n) \wedge status(LChl(n), t) = S, \text{ then } S \\ \text{else if } IsPar(n) \wedge \\ \quad (\exists c \in Chl(n) \text{ s.t. } (status(c,t) \neq I) \wedge Unskipped(c,t) = -1), \\ \quad \text{then } ParStatus(n,t) \\ \text{else if } IsDec(n) \wedge (ActNode(t) = n \vee status(FChl(n), t) \neq I), \\ \quad \text{then } DecStatus(n,t) \\ \text{else } I \end{cases}$$

$status(n,t)$ describes the status of node n at time step t. $Chl(n)$ is the set of children of n. If both $IsDec(n)$ and $ActNode(t) = n$, then n is a decorator that

skipped its child.

$$ParStatus(n,t) \coloneqq \begin{cases} \text{if } IsFailure(n,t), \text{ then } F \\ \text{else if } NumSucc(n,t) \geq SuccThresh(n), \text{ then } S \\ \text{else } R \end{cases}$$

$$IsFailure(n,t) \coloneqq \begin{cases} \text{if } \exists a \in Anc(n) \cup \{n\} \text{ s.t. } status(a, t-1) \in \{S,F\}, \\ \quad \text{then } \bot \\ \text{else } IsFailure(n,t-1) \vee \\ \quad \exists c \in Chl(n) \text{ s.t. } status(c,t) = F \end{cases}$$

$$NumSucc(n,t) \coloneqq \begin{cases} \text{if } \exists a \in Anc(n) \cup \{n\} \text{ s.t. } status(a, t-1) \in \{S,F\}, \\ \quad \text{then } 0 \\ \text{else if } \exists c \in Chl(n) \text{ s.t. } status(n,t) = S, \\ \quad \text{then } NumSucc(n,t-1) + 1 \\ \text{else } NumSucc(n,t) \end{cases}$$

4.2 Total

Unlike the Leaf encoding, in the Total encoding a tick occurs at each time step t and we compute the entire state of the tree in one time step. Consider Figure 3. By definition, the status of wideSel is S if a child returns S, R if a child returns R, and F if all children return F (a status of I is impossible for the root as the root will always run). The Total encoding uses this sort of definition directly for each node. Thus the status of each child is based on if the child runs and the custom code of the leaf node. As a result, in this case child3 will only run if child2 runs and returns F, and child2 will only run if child1 runs and returns F. This is all directly encoded, though it is done formulaically. The state of the tree is determined by resolving the dependency chain. Formally the encoding is defined as follows:

$$IsActive(n,t) \coloneqq$$

$$\begin{cases} \text{if } IsRoot(n), \text{ then } \top \\ \text{else if } \neg IsActive(parent(n),t) \vee Skipped(n,t), \text{ then } \bot \\ \text{else if } n = FChl(parent(n)), \text{ then } \top \\ \text{else if } ResFrom(n,t), \text{ then } \top \\ \text{else if } IsSel(parent(n)), \text{ then } status(lNeigh(n),t) = F \\ \text{else if } IsSeq(parent(n)), \text{ then } status(lNeigh(n),t) = S \\ \text{else if } IsPar(parent(n)), \text{ then } \top \\ \text{else } \bot \end{cases}$$

$IsActive(n,t)$ is True if at time t node n executed. In this encoding multiple nodes can be active at the same time. Notation is reused from the Leaf encoding where applicable. For instance, $IsSel(n)$ is defined as before. $lNeigh(n)$ functions the same way as $rNeigh(n)$, except with the Left Neighbor.

$$Skipped(n,t) \coloneqq \begin{cases} \text{if } t \leq 0, \text{ then } \bot \\ \text{else if } \exists a \in Anc(n) \text{ s.t. } status(a, t-1) \in \{S, F\}, \\ \quad \text{then } \bot \\ \text{else if } IsParSynch(parent(n)) \wedge status(n, t-1) = S, \\ \quad \text{then } \top \\ \text{else if } IsSeqWM(parent(n)) \wedge \\ \quad \exists x \geq 1 \text{ s.t. } status(rNeigh(n)^x, t-1) = R, \text{ then } \top \\ \text{else if } IsSelWM(parent(n)) \wedge \\ \quad \exists x \geq 1 \text{ s.t. } status(rNeigh(n)^x, t-1) = R, \text{ then } \top \\ \text{else } Skipped(n, t-1) \end{cases}$$

$Skipped(n,t)$ is used to determine if a node with memory caused node n to be skipped at time t.

$$ResFrom(n,t) \coloneqq IsSeq(parent(n)) \wedge \exists x \geq 1 \text{ s.t. } status(rNeigh(n)^x, t-1) = R$$

Intuitively, $ResFrom(n,t)$ tells us if at time t we are supposed to resume from node n or not (only affects certain nodes with memory). As before $status(n,t)$ is used to describe the status of a node n at time t.

$$status(n,t) \coloneqq \begin{cases} \text{if } \neg IsActive(n,t), \text{ then } I \\ \text{else if } IsSel(n), \text{ then } SelStatus(n,t) \\ \text{else if } IsSeq(n), \text{ then } SeqStatus(n,t) \\ \text{else if } IsPar(n), \text{ then } ParStatus(n,t) \\ \text{else if } IsDec(n), \text{ then } DecStatus(n,t) \\ \text{else } LeafStatus(n,t) \end{cases}$$

$$SelStatus(n,t) \coloneqq \begin{cases} \text{if } \exists c \in Chl(n) \text{ s.t. } status(c,t) \in \{S, R\}, \\ \quad \text{then } status(c,t) \\ \text{else } F \end{cases}$$

$$SeqStatus(n,t) \coloneqq \begin{cases} \text{if } \exists c \in Chl(n) \text{ s.t. } status(c,t) \in \{F, R\}, \\ \quad \text{then } status(c,t) \\ \text{else } S \end{cases}$$

$$ParStatus(n,t) \coloneqq \begin{cases} \text{if } \exists c \in Chl(n) \text{ s.t. } status(c,t) = F, \text{ then } F \\ \text{else if } NumSucc(n,t) \geq SuccThresh(n), \text{ then } S \\ \text{else } R \end{cases}$$

$$NumSucc(n,t) \coloneqq \left|\{c : c \in Chl(n) \wedge (status(c,t) = S \vee Skipped(c,t))\}\right|$$

$SuccThresh(n)$ represents the number of nodes that need to return Success for the parallel policy to return S. For the two default policies, Success On One and Success On All, the values would be 1 and $\left|Chl(n)\right|$ respectively. Therefore, if a node is a Parallel node and isn't I, then if any of the children returned F the node returns F. Otherwise, it compares against the $SuccThresh(n)$. $NumSucc(n,t)$ is the number of children of n that returned S at time t. Since Leaf Nodes can be customized, it is impossible to fully characterize their behavior, and there are too many Decorator nodes to concisely list here. As such, we have $DecStatus(n,t) \in \{S,F,R\}$ and $LeafStatus(n,t) \in \{S,F,R\}$.

4.3 BTCompiler

The encoding for the BTCompiler, as best we understand it, has been included in [22]. Unfortunately, we were unable to install the tool. However, based on various examples in the BTCompiler repository, we concluded that the file 'bt_classic.smv' [2] contains the relevant encoding. The encoding presented in [22] is meant to approximate this, in the same way that the Leaf and Total encodings approximate the actual encodings used by BehaVerify.

5 Results

We include the results of two main experiments: Checklist and BlueROV. Checklist is a parameterized example that takes as input an integer n and produces a BT that contains n checks which must either succeed or a fallback triggers. For each check we include two LTL specs, one to be proved and one to be disproved. Leaf_v2, Total_v2, Total_v3, and BTC models were used in this experiment, where Leaf_v2 is based on the Leaf encoding, Total_v2 and Total_v3 are based on the Total encoding, and BTC is based on the BTCompiler encoding. The other example is BlueROV, the controller in an ongoing DARPA project. As this example requires blackboard variables which BTCompiler does not support, it is not included, so only the 3 BehaVerify encodings are considered. We include timing results for verifying the LTL spec as well as memory usage. Timing values are based on nuXmv's 'time' command. Maximum Resident Size values are based on nuXmv's usage command, which uses getrusage(2) [6]. Maximum Resident Size is the maximum amount of RAM that is actually used by a process. All tests were run on a computer using Ubuntu 22.04 with 32 gb of ram and an i7-8700K Intel processor. Both the tool and instructions on how to recreate these tests are available [3]. The tests only consider the time to verify LTL specifications in nuXmv. Time spent building the model in nuXmv is not included as it never exceeded .2 seconds. The time spent converting the BTs to models is not included as it is also fairly negligible, but can be found in [22].

[2] https://github.com/CARVE-ROBMOSYS/BTCompiler/blob/master/smv/bt_classic.smv

[3] https://github.com/verivital/behaverify

5.1 Checklist and Parallel-Checklist

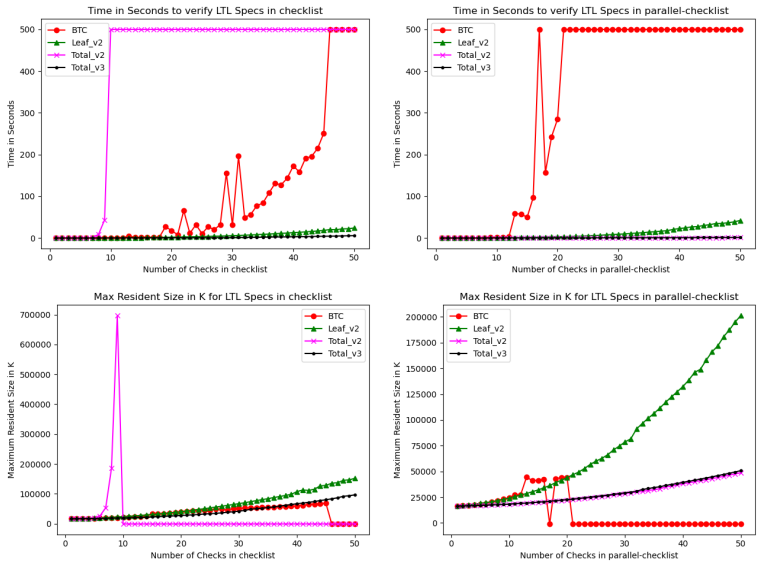

Fig. 4. Timing and memory results for verifying LTL specifications in nuXmv for Checklist and Parallel-Checklist. Timeout is set to 5 minutes. If a timeout occurred, a value of 350 is used for timing and -1000 for memory. After 3 timeouts, the remaining tests for the version are skipped. BTC is based on BTCompiler, Leaf_v2 is a model based on the Leaf encoding, and Total_v2 and Total_v3 are models based on the Total encoding.

The checklist examples consist of a series of checks that run in order by nested sequence nodes. Each check consists of a selector node, a safety check leaf node that can return S or F, and a backup node that can only return S. Thus if the safety check fails, the selector will run the backup which will return S. This process continues until each check has been run. See [22] for visual examples. Parallel-checklist replaces the sequence nodes with parallel nodes. Each check has two LTL specifications, one True and one False. The True/False specifications require that if a safety check fails, then a backup is triggered/not triggered. Due to differences in encodings, the specifications are slightly different for each version. We include one example here. The remainder can be found in [22].

For Total_v2 and Total_v3:

$G(safety\_checkX.status = F \implies backupX.status = S)$;

$G(safety\_checkX.status = F \implies !(backupX.status = S))$;

Checklist Results Discussion Having re-run the checklist and parallel checklist experiments three times for BTCompiler only, we have found that the spikes are present each time. These results can be found in [22]. The results are extremely similar, so we find it unlikely that this is a fluke. Furthermore, we note that there is a spike at 19 in both the checklist and parallel-checklist experiments. Since nuXmv is using a BDD model to verify the LTL Specifications, we assume that there is some sort of awkward break point with the number of variables that causes the efficiency to greatly suffer at certain points.

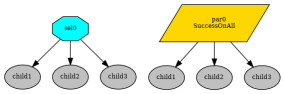

Fig. 5. Two examples with 3 children.

Note that Total_v2 works much better on Parallel-Checklist than on Checklist. This is because of the logic chain created by Selector and Sequence nodes. Consider the Selector Example in Figure 5. The status of child3 depends on if child3 is active, which depends on the status of child2, which depends on if child2 is active, which depends on the status of child1, which depends on if child1 is active, which depends on if sel0 is active. The chain quickly becomes unmanageable (see [22] for visual examples of the BTs). This is not the case with Parallel-Checklist. Consider the Parallel Example in Figure 5. The status of child3 depends on if child3 is active, which depends only on par0 and what child3 returned last time. Thus the dependency chain is much shorter and thus Total_v2 performs better on Parallel-Checklist. Total_v3 avoid this by 'guiding' nuXmv through this dependency chain by introducing intermediate variables.

Finally, note that the timing results in Figure 4 clearly demonstrate that the Total_v3 encoding outperforms the rest.

5.2 BlueROV

We considered three versions of BlueROV: warnings only, small, and full. The differences between these versions is what range of values each blackboard variable is allowed to use. See [22] for an image of the BT. We consider 5 sets of 2 LTL specifications. The timeout for each set of specifications was 10 minutes. For each warning, the first LTL specification requires that if the warning is set to True, then the appropriate Surface Task is triggered. This specification is False in all cases except battery low warning. The second LTL specification requires that if in a given tick a warning is set, then during that tick a surface task will trigger. This is true for all warnings except the home reached warning.

Table 1. blueROV, Time in Seconds to Compute LTL

Model	LTL Spec	Leaf_v2	Total_v2	Total_v3
warnings only	low battery	0.39	4.16	0.12
warnings only	emergency stop	0.48	4.21	0.14
warnings only	home reached	0.66	-	1.70
warnings only	obstacle	0.54	7.79	0.17
warnings only	sensor degradation	0.49	4.11	0.13
small	low battery	23.43	5.06	0.33
small	emergency stop	30.47	6.40	1.02
small	home reached	31.48	-	2.58
small	obstacle	39.34	9.87	0.39
small	sensor degradation	31.73	5.23	0.34
full	low battery	79.08	5.54	0.60
full	emergency stop	156.20	6.49	1.81
full	home reached	107.05	-	3.59
full	obstacle	323.00	10.06	1.10
full	sensor degradation	106.16	6.57	1.46

For the Leaf_v2 encoding, these look as follows for battery:

$$G(next(battery\_low\_warning) = 1 \wedge active\_node = battery2bb \implies (active\_node > -1 U (active\_node = surface)));$$
$$G(next(battery\_low\_warning) = 1 \wedge active\_node = battery2bb \implies (active\_node > -1 U (active\_node \in \{surface, surface1, surface2, surface3, surface4\})));$$

For the Total encodings, these look as follows for battery:

$$G(next(battery\_low\_warning) = 1) \wedge battery2bb.active) \implies (surface.active));$$
$$G(next(battery\_low\_warning) = 1) \wedge battery2bb.active) \implies (surface.active | surface1.active | surface2.active | surface3.active | surface4.active));$$

BlueROV Results Discussion The BlueROV models differ from each other only in the number of values that each blackboard variable can take. Thus based on the results in Table 1, we can see that the Leaf_v2 encoding has the worst scaling of the three with respect to blackboard variable size. Total_v3 improves upon both Total_v2 and Leaf_v2. BTCompiler does not support blackboard variables.

6 Conclusions and Future Work

We introduced BehaVerify, a tool for turning a py_tree into a .smv file for use with nuXmv. We consider several possible encodings for this task and compared them to the encoding that BTCompiler uses. The results indicate that the encoding used by Total_v3 is the best choice.

Future work includes general polish and improvements and expanding support for the various built-in nodes in py_trees. In addition to this, we plan to re-work certain elements of BehaVerify. For instance, currently, in order for BehaVerify to detect blackboard variables in a py_tree using custom leaf nodes, the user must create a field that BehaVerify looks for within the custom node. This could certainly be handled better in the future. In terms of encodings, we plan to focus on Total_v3. An improvement that has been considered, but not yet implemented, would be to restrict the incoming values to the leaf nodes to reduce state space. Specifically, in cases where a leaf node does not run, there is no need to consider the incoming status. Currently, this could be accomplished by tying the incoming value to the active value. However, this would likely cause worse performance for the same reason that Total_v2 performs worse than Total_v3. Therefore, the intended solution would be to, in some sense, enumerate all possible input values, which would hopefully shift some of the burden off of nuXmv and onto BehaVerify.

Acknowledgments

The material presented in this paper is based upon work supported the Defense Advanced Research Projects Agency (DARPA) through contract number FA8750-18-C-0089, the Air Force Office of Scientific Research (AFOSR) award FA9550-22-1-0019, and the National Science Foundation (NSF) through grant number 2028001. Any opinions, findings, and conclusions or recommendations expressed in this publication are those of the authors and do not necessarily reflect the views of DARPA, AFOSR, or NSF.

References

1. Biggar, O., Zamani, M.: A framework for formal verification of behavior trees with linear temporal logic. IEEE Robotics and Automation Letters **5**(2), 2341–2348 (2020). https://doi.org/10.1109/LRA.2020.2970634
2. Biggar, O., Zamani, M., Shames, I.: On modularity in reactive control architectures, with an application to formal verification (2020). https://doi.org/10.48550/ARXIV.2008.12515, https://arxiv.org/abs/2008.12515
3. Biggar, O., Zamani, M., Shames, I.: A principled analysis of behavior trees and their generalisations (2020). https://doi.org/10.48550/ARXIV.2008.11906, https://arxiv.org/abs/2008.11906
4. Biggar, O., Zamani, M., Shames, I.: An expressiveness hierarchy of behavior trees and related architectures (2021)

5. Broder, D.: Blackboard documentation (Apr 2014), https://forums.unrealengine.com/t/blackboard-documentation/1795
6. Cavada, R., Cimatti, A., Dorigatti, M., Griggio, A., Mariotti, A., Micheli, A., Mover, S., Roveri, M., Tonetta, S.: The nuxmv symbolic model checker. In: Biere, A., Bloem, R. (eds.) CAV. Lecture Notes in Computer Science, vol. 8559, pp. 334–342. Springer (2014)
7. Colledanchise, M., Cicala, G., Domenichelli, D.E., Natale, L., Tacchella, A.: 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 (sep 2021). https://doi.org/10.1109/iros51168.2021.9636129, https://doi.org/10.1109%2Firos51168.2021.9636129
8. Colledanchise, M., Cicala, G., Domenichelli, D.E., Natale, L., Tacchella, A.: A toolchain to design, execute, and monitor robots behaviors. CoRR **abs/2106.15211** (2021), https://arxiv.org/abs/2106.15211
9. Colledanchise, M., Murray, R.M., Ögren, P.: Synthesis of correct-by-construction behavior trees. In: 2017 IEEE/RSJ International Conference on Intelligent Robots and Systems (IROS). pp. 6039–6046 (2017). https://doi.org/10.1109/IROS.2017.8206502
10. Colledanchise, M., Natale, L.: Improving the parallel execution of behavior trees. In: 2018 IEEE/RSJ International Conference on Intelligent Robots and Systems (IROS). IEEE (oct 2018). https://doi.org/10.1109/iros.2018.8593504, https://doi.org/10.1109%2Firos.2018.8593504
11. Colledanchise, M., Natale, L.: Analysis and exploitation of synchronized parallel executions in behavior trees. In: 2019 IEEE/RSJ International Conference on Intelligent Robots and Systems (IROS). IEEE (nov 2019). https://doi.org/10.1109/iros40897.2019.8967812, https://doi.org/10.1109%2Firos40897.2019.8967812
12. Colledanchise, M., Natale, L.: Handling concurrency in behavior trees. CoRR **abs/2110.11813** (2021), https://arxiv.org/abs/2110.11813
13. Colledanchise, M., Natale, L.: On the implementation of behavior trees in robotics. IEEE Robotics and Automation Letters **6**(3), 5929–5936 (jul 2021). https://doi.org/10.1109/lra.2021.3087442, https://doi.org/10.1109%2Flra.2021.3087442
14. Colledanchise, M., Ögren, P.: Behavior Trees in Robotics and AI. CRC Press (jul 2018). https://doi.org/10.1201/9780429489105, https://doi.org/10.1201%2F9780429489105
15. Crytek: Behavior tree blackboard (Apr 2022), https://docs.cryengine.com/display/CEPROG/Behavior+Tree+Blackboard
16. EpicGames: Behavior tree overview (Aug 2021), https://docs.unrealengine.com/4.27/en-US/InteractiveExperiences/ArtificialIntelligence/BehaviorTrees/BehaviorTreesOverview/
17. Ghzouli, R., Berger, T., Johnsen, E.B., Dragule, S., Wąsowski, A.: Behavior trees in action: a study of robotics applications. In: Proceedings of the 13th ACM SIGPLAN International Conference on Software Language Engineering. ACM (nov 2020). https://doi.org/10.1145/3426425.3426942, https://doi.org/10.1145%2F3426425.3426942
18. Giunchiglia, E., Colledanchise, M., Natale, L., Tacchella, A.: Conditional behavior trees: Definition, executability, and applications. In: 2019 IEEE International Conference on Systems, Man and Cybernetics (SMC). pp. 1899–1906 (2019). https://doi.org/10.1109/SMC.2019.8914358

19. Grunske, L., Winter, K., Yatapanage, N.: Defining the abstract syntax of visual languages with advanced graph grammars—a case study based on behavior trees. Journal of Visual Languages & Computing **19** (06 2008). https://doi.org/10.1016/j.jvlc.2007.11.003
20. Ogren, P.: Increasing modularity of uav control systems using computer game behavior trees (08 2012). https://doi.org/10.2514/6.2012-4458
21. Safronov, E., Colledanchise, M., Natale, L.: Task planning with belief behavior trees. CoRR **abs/2008.09393** (2020), https://arxiv.org/abs/2008.09393
22. Serbinowski, B., Johnson, T.: Behaverify: Verifying temporal logic specifications for behavior trees (2022), https://arxiv.org/abs/2208.4436782
23. Shoulson, A., Garcia, F., Jones, M., Mead, R., Badler, N.: N.i.: Parameterizing behavior trees. vol. 7060, pp. 144–155 (11 2011). https://doi.org/10.1007/978-3-642-25090-3_13
24. Stonier, D.: py-trees 2.1.6 module api (May 2021), https://py-trees.readthedocs.io/en/devel/modules.html
25. Tumova, J., Marzinotto, A., Dimarogonas, D.V., Kragic, D.: Maximally satisfying ltl action planning. In: 2014 IEEE/RSJ International Conference on Intelligent Robots and Systems. pp. 1503–1510 (2014). https://doi.org/10.1109/IROS.2014.6942755
26. Ögren, P.: Convergence analysis of hybrid control systems in the form of backward chained behavior trees. IEEE Robotics and Automation Letters **5**(4), 6073–6080 (2020). https://doi.org/10.1109/LRA.2020.3010747