BehaVerify: A Formal Verification Tool for Behavior Trees

Serena S. Serbinowska Ben Wooding Diego Manzanas Lopez Luke Nam
Anne M. Tumlin Nicholas K. Potteiger Hongchao Zhang Taylor T. Johnson
Institute for Convergent Software Integrated Systems, Vanderbilt University
Nashville, TN 37235, USA
{serena.serbinowska, ben.wooding, diego.manzanas.lopez, luke.nam,
anne.m.tumlin, nicholas.potteiger, hongchao.zhang, taylor.johnson}@vanderbilt.edu

This paper presents BehaVerify, a verification toolbox for *behavior trees* (BTs) and Neuro-Symbolic BTs. BTs are a graphical language for planning and sequencing behaviors — similar to extended state machines — and are increasingly used in safety-critical domains. BehaVerify enables users to specify a BT using a *Domain Specific Language* (DSL) with formal syntax and semantics. This language further includes *Linear Temporal Logic* and *Computation Tree Logic* specifications. BehaVerify uses an efficient encoding into nuXmv, called *fastforwarding*, to quickly verify if a specification holds over the BT. Additionally, BehaVerify can generate implementations of the BT in Python and Haskell. Through Python, BehaVerify has been integrated with PyTrees and Robot Operating System (ROS) and has been deployed to several autonomous robotic systems, including for verifying runtime monitors. While prior works have developed the underlying theory, this manuscript focuses on the toolbox usage, functionality and implementation. BehaVerify is open source:

https://github.com/verivital/behaverify

1 Introduction

Behavior Trees (*BT*s) are high-level controllers that have gained popularity and are increasingly used for safety-critical tasks. Accordingly, it is imperative to verify their correctness. In medicine, *BT*s are used for sensitive machine insertion tasks [36] and to aid with hand rehabilitation [37]. In security, [24, 22] use *BT*s with learning-enabled components for autonomous computer network defense. Sensitive drone tasks rely on *BT*s for drones that inspect structures [23] and in mining a BT controls robot that assembles explosive charges and places them in the rock face [14]. Example use cases for *BT*s are provided in detail in the surveys [31, 19]. We distinguish our work from the formalism 'Behavior Trees' that refers to a graphical modeling language used for requirement handling [13]. Intuitively, a *BT* is a rooted tree re-evaluated from the root on every 'tick'. Interior nodes are control-flow operators. A *selector* tries its children in order until one succeeds, while a *sequence* runs its children in order until one fails. In contrast, leaf nodes check a condition or perform an action. Every node returns *Success*, *Failure* or *Running*, and that status propagates back up the tree to determine what runs next. The result offers the modularity and readability of a state machine without the quadratic growth in explicit transitions, which is partly why *BT*s have displaced hand-written state machines across robotics.

In this work, we present BehaVerify, which builds upon the theoretical foundations [26, 27, 28, 29]. To maintain brevity and avoid repetition of theoretical details, this work focuses on the usage, functionality and implementation of BehaVerify.

Related Verification Tools. Several existing tools and frameworks verify *BT*s. ArcadeBT [15] is a verification tool for *BT*s that uses linear *Constrained Horn Clauses* (CHCs). BT2BIP [34], or xml2bip,

Submitted to:
FMAS 2026

© S.S. Serbinowska et al.
This work is licensed under the
Creative Commons Attribution License.

transforms *BT*s into the *Behavior-Interaction-Priority* (BIP) framework and then treats them as an extended finite state machine to enable verification using nuXmv [7]. BT2Fiacre [18] translates *BT*s into Fiacre [3] which enables verification. BTCompiler [6] utilizes OCaml and ROCQ (formerly COQ) [33]. MoVe4BT [21] uses the Timed CSPbt modeling and allows for LTL specifications and true parallel execution of nodes. We recreate examples from BT2BIP and BT2Fiacre, and compare BehaVerify against MoVe4BT and BT2Fiacre, while [26] sufficiently compares BTCompiler. A framework for formal verification of *Linear Temporal Logic* (LTL) specifications for *BT*s is included in [5]; unlike BehaVerify, it only support Booleans.

While we are primarily focused on design time/offline verification, there exist other tools for *BT*s. A formal property specification language was introduced in [9]; it can be used to build runtime monitors for *BT*. *BT* synthesis tools [25, 17] create *BT*s to meet specifications.

This Work versus Prior BehaVerify Papers. BehaVerify has never itself been the subject of a tool paper. Our earlier publications [26, 29, 28, 27] are theory papers, albeit the appearance of the tool's name in the title of [26] invites confusion on this point. They contribute the semantics of Stateful Behavior Trees (*SBT*s), their neuro-symbolic extension, and the runtime-monitor construction. Each paper was accompanied by a separate research prototype, and none evaluated BehaVerify as a tool. We summarize and cite that theory here rather than re-develop it. What this paper contributes is the tool itself: one distribution-ready toolbox that implements all three lines of that theory behind a single Domain Specific Language (DSL) and command-line interface (CLI), the engineering that makes it usable and reusable by others, and an evaluation the theory papers did not perform.

Contributions. BehaVerify[1] formalizes *BT*s and provides a framework to verify them with nuXmv, as well as run them in Python and Haskell. The following three capabilities implement previously published semantics. What is new is their *consolidation*: all three now live in one toolbox, behind a single DSL and CLI, rather than in three separate prototypes.

- **DSL for SBTs** (*SBT*s): BehaVerify can verify the *SBT*s formalized in [29], implemented as a DSL for *BT*s and their semantics, including a restricted grammar for writing updates to the state variables (the 'blackboard');
- **Neuro-Symbolic Behavior Trees** (*NSBT*s): BehaVerify handles *BT*s that make use of neural networks within leaf nodes, as introduced in [27], with extensions of the DSL and semantics as *SBT*s;
- **Verified Runtime Monitors**: BehaVerify has the capability to generate and verify runtime monitors, leveraging the method developed in [28].

The contributions that are new to this paper, and that did not exist at the time of the prior papers, are the following.

- **First distribution-ready release:** BehaVerify can be installed via PyPI[2] (`pip install behaverify`) or Docker and run directly from the command line with multiple modes, as described in the Documentation webpages[3]; a tutorial with examples is also available[4]. A graphical user interface (GUI)

[1] `https://github.com/verivital/behaverify`
[2] `https://pypi.org/project/behaverify/`
[3] `https://verivital.github.io/behaverify/`
[4] `https://github.com/verivital/behaverify/tree/main/tutorial_examples`

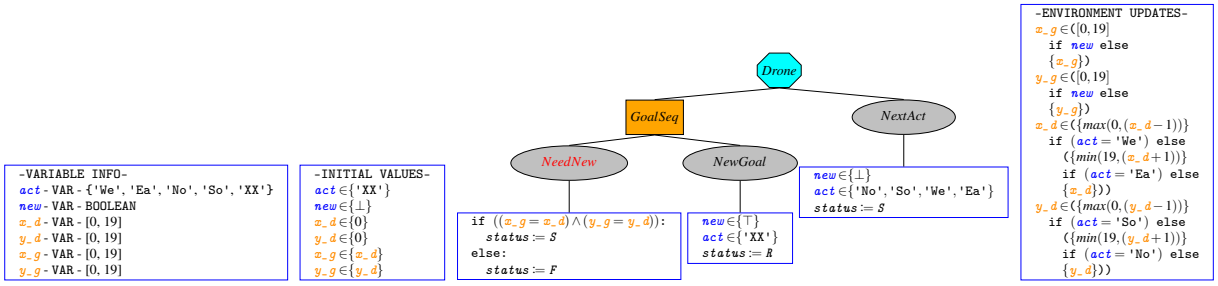


Figure 1: *A simple drone controller BT*. Node *NeedNew* checks if the drone's location (x_d, y_d) is equal to the location of the goal (x_g, y_g). If it is, a new goal is needed. Otherwise, an action is nondeterministically chosen. Between ticks, the environment updates occur; the location of the drone is updated based on the chosen direction and a new goal is generated if needed. An example execution can be found in Table 1. `XX` is a placeholder meaning *no action was chosen* on that time step. "Sampling" from an interval denotes a nondeterministic choice over that interval, which the model checker explores exhaustively rather than treating as a random draw.

visualizes trees and counterexample traces. Continuous Integration and Continuous Delivery/Deployment (CI/CD) via GitHub Actions, with unit and end-to-end tests, enables frequent and reliable code changes and updates.[5]

- **Scalability, and the first controlled ablation of fastforwarding:** a naive-encoding mode isolates the effect of the encoding itself (Table 4). The comparison in [29] was against MoVe4BT, a different tool with different semantics, so it cannot separate the effect of the encoding from the effect of the tool. Running both encodings inside BehaVerify separates the two, adds state-space measurements, and quantifies the scalability advantage of BehaVerify over the prior theoretical implementations.
- **New comparisons:** the BT2Fiacre drone recreation (Table 5), including a previously unreported battery-priority violation, and recreations of BT2BIP's Mars Rover and Train Control examples. Neither tool appears in any prior BehaVerify paper. Of the tool comparisons, only the MoVe4BT summary is carried over. The MoVe4BT comparison scales to 2 055 nodes, while the BT2Fiacre comparison involves a tree of more than 30 nodes operating in a three-dimensional environment.

2 Behavior Tree Overview

Figure 1 and Table 1 provide a *BT* sample execution of a 'Drunken Drone'. A *BT* is a rooted tree that executes when it receives an exterior signal: a 'tick.' We use discrete time steps to describe stepping through the tree. At each time step during a tick, exactly one node in the tree is active. If a node is active at a given time step, the node that will be active at the next time step is either a parent or a child of said node (with exception of the root node).

Tree traversal is generally depth-first. When a node returns, its status becomes one of *success*, *failure*, or *running* (*S*, *F*, or *R*). Until a node returns, its status is *invalid* (*I*). The children of a node are ordered; visually we use left to right, *i.e.* the leftmost child is the first child. We consider nodes in three broad categories: *leaf*, *composite*, and *decorator*.

[5] A key contribution of BehaVerify is its reusability, and a reproducibility package accompanies every result reported here.

Time	1	2	3	4	5	6	7	8	9	10	11	12	13	14	15	16
Tick	1	1	1	1	1	1	1	1	2	2	2	2	2	2	2	2
Active	D	GS	NN	GS	NG	GS	D	Env	D	GS	NN	GS	D	NA	D	Env
Return	*I*	*I*	*S*	*I*	*R*	*R*	*R*	*I*	*I*	*I*	*F*	*F*	*I*	*S*	*S*	*I*
act	XX	XX	XX	XX	XX	XX	XX	XX	XX	XX	XX	XX	XX	No	No	No
new	$\bot$	$\bot$	$\bot$	$\bot$	$\top$	$\top$	$\top$	$\top$	$\top$	$\top$	$\top$	$\top$	$\top$	$\bot$	$\bot$	$\bot$
x_d	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0
y_d	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	1
x_g	0	0	0	0	0	0	0	2	2	2	2	2	2	2	2	2
y_g	0	0	0	0	0	0	0	3	3	3	3	3	3	3	3	3
FF	s_0							s_1								s_2

Table 1: *A table illustrating a hypothetical execution of the BT shown in* Figure 1. **Time** refers to each time step. **Tick** refers to the number of times the tree is called. **Active** refers to a single 'active' node at each time step. Node abbreviations: D=Drone (root selector), GS=GoalSeq (sequence), NN=NeedNew (check), NG=NewGoal (action), NA=NextAct (action), Env=environment update (occurs between ticks). Nodes return one of *S*, *F*, or *R* when they finish; *I* indicates the node has not yet returned this tick. **Return** shows the current active node status (not a cumulative status). **FF** marks the only states kept by fastforwarding; each tick is one transition, $s_0 \to s_1 \to s_2$.

Variables. Variables can be split into two categories: blackboard and environment. The *blackboard* is the shared memory of a tree and can be updated between time steps, *e.g.* act. The *environment* represents variables not under control of the *BT* and they can be updated between ticks, *e.g.* x_g. It is up to the user to decide if a variable should be a blackboard or environment variable.

Runtime Assumption. As mentioned, there are three return statuses: *S*, *F*, and *R*. We assume nodes return instantaneously. If a node waits on a long-running calculation, it instantly returns *R* to not block the tree. Moreover, we assume that the environment does not change during the tick, only between ticks.

Leaf Nodes. Leaf nodes are split into two categories: *checks* and *actions*. *Checks* return *S* if a condition is true and *F* otherwise. The conditions are predicates depending on blackboard and environment variables. *Actions* can read and set variables and are able to return *S*, *F*, or *R*. Although it is possible to reduce an entire *BT* to a single action, key benefits of *BT*s would be lost: *modularity* (ability to reuse subtrees across contexts), *visual clarity* (hierarchical structure providing an intuitive representation of control flow), and *maintainability* (nodes can be modified without affecting the tree significantly).

Composite Nodes. The three composite node types are *selector*, *sequence*, and *parallel*. *Selector* nodes activate each of their children in order until a child returns *S* or *R*, then the selector immediately returns that same value. If all children return *F*, the selector returns *F*. Thus a selector can return any of *S*, *F* or *R*. *Sequence* nodes work dually, performing identically to Selector nodes but with *F* and *S* swapped. A sequence can likewise return any of the three. *Parallel* nodes allow their children to be executed simultaneously. In practice, our tool uses PyTrees [32], which does not have truly parallel execution. Each child is activated in order (no shuffling) and a status is returned based on a user-chosen policy, *e.g.* '*success on all*'. While selector and sequence nodes can terminate early, skipping children, parallel nodes cannot.

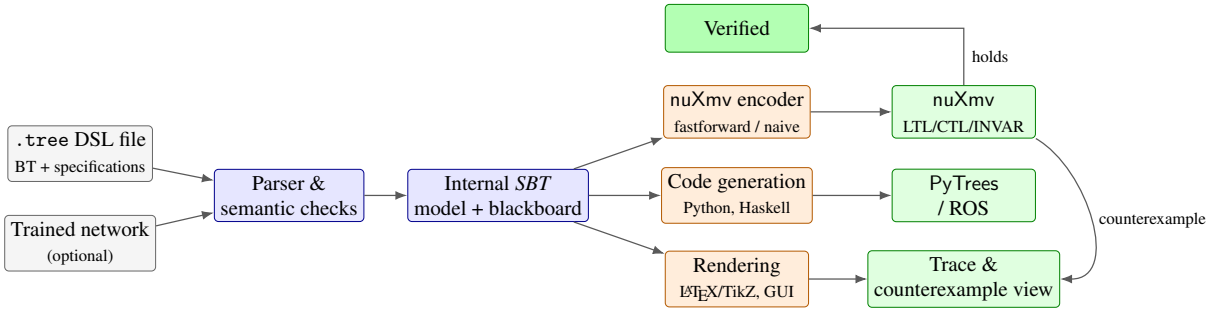


Figure 2: Architecture of BehaVerify. One DSL front end builds a single internal model that every backend reads.

Decorator Nodes. Decorator nodes have **exactly** one child. Decorator nodes augment the behavior of the child, *e.g.* negate child, repeat child 'n' times, *etc*. They also facilitate node reuse.

Behavior Tree Diagrams. A key strength of *BT*s is their visual aspect. As such, it is important to be able to visualize *BT*s. BehaVerify allows the user to visualize *BT*s through `dsl_to_latex.py`, which accepts as input a `.tree` file and produces a LaTeX diagram; this was used to generate Figure 1. Each node is represented using a different shape and color. Selector nodes are octagonal, sequence nodes are rectangles, parallel nodes are parallelograms, and decorator nodes are trapezoids. We represent leaf nodes using ovals; checks have a red label and actions have a black label. Beneath each node is a box containing the code for that leaf node. This code is executed from top to bottom.

3 BehaVerify Tool Description

This section describes BehaVerify, its inputs from the DSL, output targets, and overall usage. Figure 2 shows how the tool is organized. The user writes the *BT*, its environment and its specifications in one `.tree` file. A front end parses this file, checks types and variable scopes, and builds one internal model; neural-network leaves enter through the same front end. Every backend reads this model. The nuXmv encoder writes an `.smv` model, using either the fastforwarding or the naive encoding. The code generators write Python and Haskell implementations, which run the *BT* on robotic systems through PyTrees and ROS. The renderer draws the tree and any counterexample trace. For each specification, nuXmv [7] either confirms that it holds or returns a counterexample, which the trace viewer displays. BehaVerify is cross-platform, has been tested on Linux, macOS and Windows, installs with `pip`, and also runs through Docker for reproducibility; the documentation gives detailed instructions.

3.1 Domain Specific Language (DSL) Input

The input for BehaVerify is provided using a DSL: this includes the *BT*, the environment, and any specifications to be verified. We used textX [12] to create a grammar for *BT*s[6]. Some specific features of BehaVerify relating to the DSL are as follows: **Functions:** Functions are written in a Lisp style. For example, $1+2$ would be `(add, 1, 2)`. **Nondeterminism:** Blackboard variables can be updated

[6]The textX file that defines the grammar can be found at: `https://github.com/verivital/behaverify/tree/main/src/behaverify/data/metamodel`.

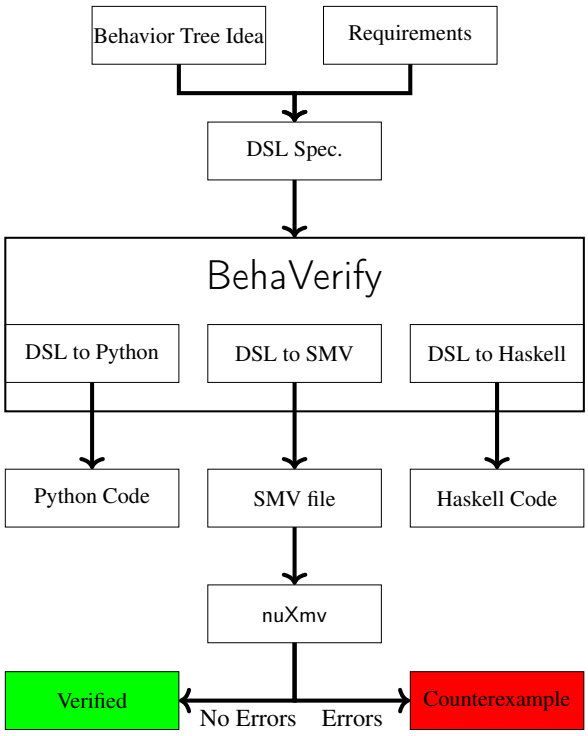


Figure 3: The user writes down a *BT* along with specifications using the DSL. Depending on the desired output, a different mode of BehaVerify is used. It is possible to produce Python and Haskell implementations of the *BT*. These implementations allow for execution of the *BT*, for example, in robotics systems via PyTrees and robot operating system (ROS). A `.smv` model can be generated, which can then be used with nuXmv [7] to verify the *BT* satisfies specifications. Should it fail, a counterexample will be produced.

nondeterministically between time steps, or each tick for environment variables. If more than one value is in a result, a value will be chosen nondeterministically. **Loops:** Loops are unraveled by BehaVerify. The example loops are used to assign values nondeterministically. **Types:** Variables are Booleans, enumerations or bounded integer ranges such as `[0, 19]`, which become finite-domain variables in nuXmv. The DSL also accepts unbounded `INT` and `REAL` variables, which become nuXmv `integer` and `real` variables. The verification modes of BehaVerify run nuXmv's BDD-based engine, which needs finite domains; a model with unbounded variables must instead be checked with nuXmv's SMT-based commands, which support invariants and LTL but not CTL. Every model in this paper uses bounded domains, so every reported result is for a finite-state system. **Arrays:** BehaVerify supports array variables, though no arrays are featured in the example. **Arguments:** Nodes can take arguments. To use an argument, modify the node to read `arguments{arg := INT}`. In the tree, to pass 3 to a node labeled `Node`, use `Node{3}`. **Specifications:** Specifications can be written using Linear Temporal Logic (LTL), Computation Tree Logic (CTL), and Invariant specifications.

To show how the DSL is actually used, Listing 1 gives the complete encoding of the drone *BT* of Figure 1, including its CTL specifications. The specifications read '`x_g` at 0' because of fastforwarding: 'at *n*' selects the stage of the variable within the tick, as described in Section 3.2.

```
configuration {} enumerations {'We', 'Ea', 'No', 'So', 'XX'}
constants {min_val := 0, max_val := 19}
variables {
    variable { bl act VAR {'We', 'Ea', 'No', 'So', 'XX'} assign{result{'XX'}}}
    variable { bl new VAR BOOLEAN assign{result{False}}}
    variable { env x_d VAR [min_val, max_val] assign{result{min_val}}}
    variable { env y_d VAR [min_val, max_val] assign{result{min_val}}}
    variable { env x_g VAR [min_val, max_val] assign{result{x_d}}}
    variable { env y_g VAR [min_val, max_val] assign{result{y_d}}}
```

```
} environment_update {
    variable_statement { x_g #{if new, assign value nondeterministically, else leave
        unchanged}#
        assign { case {new} result {(loop, loop_var, [min_val, max_val] such_that
            True, loop_var)}
            result {x_g}}}
    variable_statement {y_g
        assign {case{new} result {(loop, loop_var, [min_val, max_val] such_that True,
             loop_var)}
            result {y_g}}}
    variable_statement {x_d
        assign {
            case {(eq, act, 'We')} result { (max, min_val, (sub, x_d, 1))}
            case {(eq, act, 'Ea')} result { (min, max_val, (add, x_d, 1))}
            result {x_d}}}
    variable_statement {y_d
        assign {
            case {(eq, act, 'So')} result { (max, min_val, (sub, y_d, 1))}
            case {(eq, act, 'No')} result { (min, max_val, (add, y_d, 1))}
            result {y_d}}}
} checks {}
environment_checks {
    environment_check { NeedNew
        arguments {} read_variables { x_d, y_d, act}
        condition {(and, (eq, x_g, x_d), (eq, y_g, y_d))}}
} actions {
    action { NewGoal
        arguments{} local_variables {}read_variables {} write_variables {new, act}
        initial_values {}
        update {
            variable_statement { new assign{result{True}}}
            variable_statement { act assign{result{'XX'}}}
            return_statement {result {running}}
        }
    } action {NextAct
        arguments{} local_variables {}read_variables {} write_variables {new, act}
        initial_values {}
        update {
            variable_statement {new assign{result{False}}}
            variable_statement {act assign {result{'No', 'So', 'We', 'Ea'}}}
            return_statement {result { success}}
        }
    }
} sub_trees {}
tree {
    composite {Drone selector children {
            composite { GoalSeq sequence children { NeedNew {} NewGoal {}}}
            NextAct {}
        }
    }
}
tick_prerequisite {(True)}
```

```
specifications {
    CTLSPEC {(always_globally, (always_finally, (and, (eq, x_d at -1, x_g at 0), (eq,
        y_d at 1, y_g at 0))))}
    CTLSPEC {(always_globally, (exists_finally, (and, (eq, x_d at -1, x_g at 0), (eq,
        y_d at 1, y_g at 0))))}
}
```

Listing 1: The complete BehaVerify DSL encoding of the drone example in Figure 1.

3.2 Fastforwarding

A naive encoding of *BT*s into model checking languages would model each node activation as a separate transition. This approach is problematic: a tree with 100 nodes would require 100+ transitions per tick, leading to extremely long traces and poor verification performance. As demonstrated in [29], BehaVerify uses **Fastforwarding** to compress an entire tick into a single transition. Rather than modeling the step-by-step traversal, we compute the final state after a complete tick in one transition. This dramatically reduces trace lengths and is a primary reason BehaVerify outperforms other tools in verification time. Table 1 illustrates the difference. BehaVerify's naive encoding (`-use_encoding naive`) takes one step per node visit plus one step between ticks, which is one state per column, so the two ticks shown take 15 transitions. Fastforwarding keeps only the initial state and the state at the end of each tick, s_0, s_1 and s_2 in the row **FF**, so each tick is a single transition and a counterexample trace has one step per tick.

Relation to Existing Encodings. Fastforwarding applies a well-known idea rather than a new one. A tick is a loop-free computation: each node runs at most once, in an order fixed by the tree. Summarizing such a computation as one transition, with a fresh version of each variable after every assignment and intermediate values given by definitions instead of state variables, is the static single assignment (SSA) form [10]. SSA-style symbolic encodings are standard in software verification, for example large-block encoding in software model checking [4] and bounded model checking of C programs [8]. The stages described below are the SSA versions of the blackboard and environment variables. What is specific to *BT*s is how the guards of these versions arise and how they are exposed: return statuses decide which nodes run in a tick, nondeterministic choices inside a tick remain branching of the single transition, and stages stay visible to specifications through `at`, so a user can still state properties of values inside a tick.

Stages. A variable can change value during a tick, so we need a way to refer to it at different points within a single tick; we call these *stages*. *Stage* 0 is the value at the start of the tick, *Stage* 1 the value after the first possible change, and so on, with *Stage* -1 denoting the last stage. Here 'the first possible change' means the first node able to change that variable under a depth-first traversal; if that node is skipped, the stage falls back to the value of the previous stage.

A natural concern is *state space explosion*: if a variable has 10 possible values and we keep three stages, that appears to be 1000 possible values. There is a simple way around it. If a variable updates deterministically then each stage is a function of the previous one, so the stage can be stored as a `DEFINE` and costs no state at all. If, for instance, x is updated by incrementing it, we need no new variable for the new stage — we simply insert $x+1$ wherever it is needed. In some cases a stage can be removed outright. Specifications do not need to be rewritten when using fastforwarding: the LTL next operator (X) still refers to the next tick, **not** the next node activation [29].

Why Fastforwarding Is Sound. A compression of this kind can be mistaken for an abstraction that would require a preservation proof, so it is worth being precise. Fastforwarding is *not* an abstraction of a finer model. It directly constructs the tick-level transition relation that *defines SBT* semantics in [29], whose states are exactly the blackboard and environment valuations at tick boundaries. Deterministic intra-tick updates fold into `DEFINE`s, as above, and each nondeterministic intra-tick choice is exposed as a choice of that same transition. The resulting Kripke structure is identical to the tick-level semantics up to auxiliary variables that no specification refers to — the two are bisimilar — and bisimulation preserves full CTL*. Branching is preserved because each tick's nondeterminism becomes branching at a single transition, and intra-tick values remain expressible through stages. Empirically, both encodings returned identical verdicts on every instance where the naive encoding completed (Table 4).

State-Space Growth. In the fastforwarded model, the activity and return status of every node is a `DEFINE` computed from the stages. A state therefore holds only the blackboard and environment values at the start of a tick, plus one variable for each nondeterministic choice inside a tick; there is no program counter and no per-node status. The size of the state space depends on the variables and their domains, not on the number of nodes. In the binary-tree benchmark of Table 4, the blackboard is a counter $vv \in [0, L]$, where $L = 2^N$ is the number of leaves in the scaled subtree, and a Boolean *rv* with three stages, so the model has $8(L+1)$ states (at N=10, $\log_2(8 \cdot 1025) = 13.0$). Here the state count grows linearly with the number of nodes, $7 + 2^{N+1}$, only because this benchmark ties the range of *vv* to the number of leaves. The naive encoding instead adds a program counter over all nodes and a three-valued status variable for every leaf. Its total state space therefore grows exponentially with the number of nodes, and its reachable states grow roughly quadratically in this benchmark.

3.3 From Stateful to Neuro-Symbolic Behavior Trees

An *NSBT* is an *SBT* in which some leaf nodes consult a neural network. The step from one to the other is deliberately small: the DSL gains declarations for neural networks and for the variables they read and write, while everything else — tick semantics, blackboard, environment, specification language, fastforwarding — is unchanged. This is what lets *NSBT*s be verified by exactly the same pipeline, with no separate front end. A network declaration names an ONNX file, the input variables and the output classes, and selects how the network enters the nuXmv model. In `table` mode, BehaVerify runs the network with ONNX Runtime on every combination of input values, which is possible because network inputs must have bounded integer domains, and writes the predicted class as a `case` expression. The model stays finite-state and exact for the given network. BehaVerify also offers symbolic modes that write each `Gemm` and `ReLU` layer as piecewise-linear definitions [1], over reals or over fixed-width bit vectors; the experiments in this paper use `table` mode.

3.4 Output Targets

BehaVerify can output Python code, Haskell code, or a `.smv` model for verification with nuXmv. In the case of Python and Haskell code, BehaVerify creates the necessary tree structure and appropriately sets up variables. A simulation interface is also formed; this will run the code and print out relevant information at the end of each tick alongside an environment that evolves according to the rules in the input file provided by the user. In the Python implementation, the environment file can be treated as an interface through which the user connects the *BT* to the true environment it is being used in. For instance, a more practical version of the drone in Figure 1 might include an extra leaf node 'SendAct' that uses a

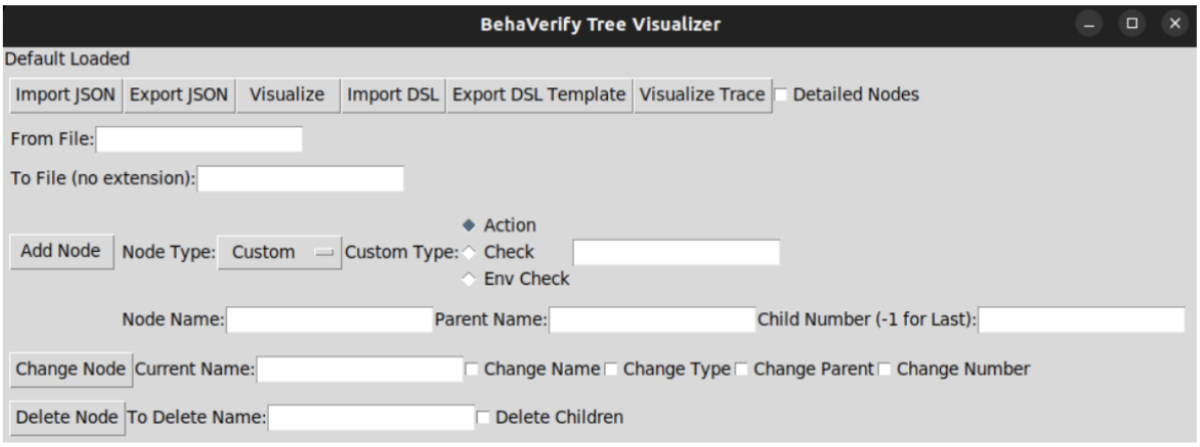


Figure 4: Graphical User Interface for BehaVerify

write environment statement in its update to communicate with the environment. The generated Python leaf node would include a function call, to a function in the environment, that the user could overwrite to implement sending an action in practice (*e.g.*, through ROS2 [20]). As we use PyTrees for our Python implementation, we have deployed numerous *BT*s with specifications verified from our DSL in nuXmv to corresponding implementations in several robotic systems, including underwater vehicles and quadcopter drones [26, 27], via the ROS PyTree library[7].

The two code backends serve different purposes. Python targets *deployment*: it builds on PyTrees and ROS, and has been run on real underwater and quadcopter systems. Haskell provides an *executable reference semantics* — pure, typed code mirroring the formal tick function directly — which we use for differential testing against both the Python backend and the nuXmv model, at compiled speed. The generated Haskell is deliberately self-contained, depending only on `base`, `random`, `array` and `containers`, which avoids inheriting the semantic gaps of third-party *BT* libraries (compare the PyTrees versus `BehaviorTree.cpp` discussion in Section 4.2, or Haskell's smarties). C, C++ and Rust backends are planned.

nuXmv can be used to verify *Linear Temporal Logic* (LTL), *Computation Tree Logic* (CTL), and Invariant specifications provided by the user. In our example, a CTL specification can state that the drone will *always eventually* reach the goal; this claim is false and nuXmv provides a counterexample. To understand why, consider that the drone moves nondeterministically, selecting North, South, East or West each tick. A counterexample trace might show the drone oscillating North-South indefinitely, never reaching the goal. Another specification can state that the drone is *always capable of eventually* reaching the goal; nuXmv confirms this claim is true. The key difference is "always eventually" (AF) versus "always exists a path to eventually" ($AGEF$). AF requires *all* paths to reach the goal, which fails under nondeterministic movement. $AGEF$ requires only that from any state *some* path exists that eventually reaches the goal, and since the drone can always choose the correct direction, such a path always exists.

3.5 Visualization and Graphical User Interface

BehaVerify includes a graphical user interface (GUI), launched with `behaverify gui` and shown in Figure 4. It is written in Python with Tkinter and draws trees with Graphviz. A user can import a tree

[7] `https://index.ros.org/p/py_trees/`

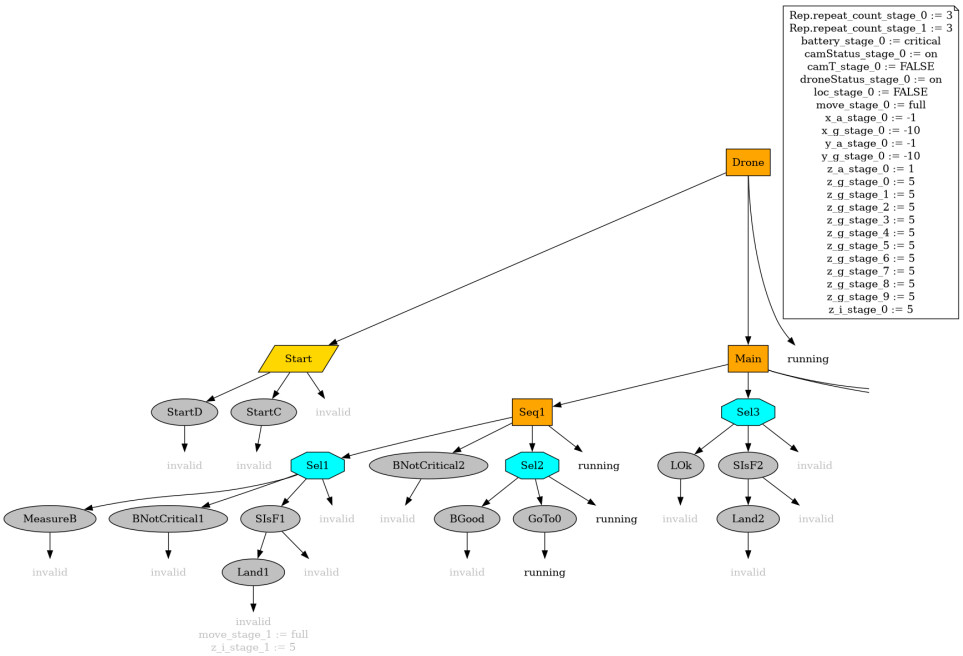


Figure 5: One tick of the `drone3` counterexample of Section 4.2, drawn by `behaverify trace` (cropped). The battery is `critical`, but `Seq1` resumes its running child `Sel2`, so the battery checks under `Sel1` are skipped and `Land1` stays `invalid`.

from a `.tree` file or from JSON, add, change and delete nodes, and render the tree, optionally labeling every leaf with its type. The edited tree can be exported as JSON or as a `.tree` skeleton that holds the tree structure and stub leaves; the user then adds variables, leaf behavior and specifications in the DSL. It also displays counterexample traces, but does not yet edit leaf behavior or run verification, which remain DSL and command-line steps.

Tree Visualization. BehaVerify's LaTeX renderer is separate from the GUI: `behaverify latex` turns a `.tree` file into a TikZ diagram that also shows the code of every leaf, the initial values, the variable domains and the environment updates. Figure 1 and the other tree diagrams in this paper were drawn this way.

Trace Visualization. nuXmv produces counterexamples when a specification is falsified. This result is valuable standalone, but also enables a user to correct any mistakes present in the model. This requires a familiarity with both nuXmv and the BehaVerify encoding in nuXmv — for particularly large examples the process of reading a trace may be tedious. To alleviate this, we developed the ability to visualize nuXmv traces: `behaverify trace`, also available in the GUI, turns a counterexample into one Graphviz image per tick (Figure 5). Each node is labeled with the status it returned in that tick, colored by status; each leaf lists the staged values it wrote; and a side box lists the values at the start of the tick. Because fastforwarding makes each tick a single step, a counterexample of k ticks becomes k images, whatever the size of the tree. For grid-world models, `behaverify grid` also draws the trajectory of the agent as an image and an animation.

3.6 Runtime Monitoring

BehaVerify also has support for *Behavior Trees with Monitor* (*BTM*) [28]. *BTM* is a variant *BT* model that allows for the creation of monitors using LTL specifications that were embedded in leaf nodes. This allows for detection of liveness violations, and can be used to monitor whether a *BT* makes progress at runtime. In the interest of space, we do not describe this further here, but interested readers can see details in [28][8].

3.7 Testing and Distribution

BehaVerify is released on PyPI (version 1.0.0) by a GitHub Actions workflow that builds and publishes each tagged release. A second workflow runs on every push and pull request. It runs the 406 `pytest` tests on Python 3.10, 3.11 and 3.12; these cover the command-line interface and every mode, parsing example models, and generation and execution of Python code for example trees. It also runs each mode of the command-line tool on example models, and it downloads nuXmv and checks that verification completes on three example models. A third workflow builds the documentation and publishes it to GitHub Pages.

4 Comparisons and Results

We compare BehaVerify against MoVe4BT [21], BT2Fiacre [18], and BT2BIP [34], and contrast BehaVerify's fastforwarding encoding to a naive nuXmv encoding. Trees, `.smv` models, raw timings (including reachable/total state-space counts across all N), and counterexample traces are in our reproducibility package. All experiments used a 6-core, 12-thread i7-10750H Intel CPU with a 16GB RAM.

On the design of this evaluation. An all-tools-on-all-benchmarks matrix is not achievable ; no two *BT* tools share an input format or node semantics: a PyTrees sequence is a `BehaviorTree.cpp` *reactive* sequence, and `BehaviorTree.cpp`'s sequence-with-memory has no PyTrees counterpart at all. Building the other toolchains also took substantial effort, and some we could not run despite dockerized, documented attempts and issues opened upstream. We therefore compare on each tool's own published examples — the setting most favorable to that tool. BehaVerify installs through `pip` or Docker and reproduces its results end to end, which was not our experience elsewhere.

Reproducibility. The comparison setups are public. Our reproducibility package [30] contains one Docker image that builds BehaVerify, nuXmv and the BT2Fiacre toolchain (Fiacre, Hippo and TINA) at pinned versions, and regenerates Tables 4 and 5 and the BT2BIP verdicts. The recreated trees are in the BehaVerify repository. MoVe4BT has only a graphical interface, so we provide its generated inputs, a containerized desktop in which it runs, and its recorded logs. The released BT2BIP toolchain did not run in our containers, so only the BehaVerify side of that comparison is reproduced.

Limitations. Our comparisons inherit the semantic mismatch described above, so a recreated tree is the closest faithful equivalent rather than an identical model. BehaVerify follows PyTrees semantics and so does not offer true parallel node execution, which BT2BIP and MoVe4BT do; this is a deliberate alignment with the library we deploy against rather than an implementation gap. BehaVerify also has

[8] `https://github.com/verivital/behaverify/tree/main/REPRODUCIBILITY`

Capability	BehaVerify	MoVe4BT	BT2Fiacre	BT2BIP
LTL specifications	✓	✓	✓	✓
CTL specifications	✓	–	✓‡	–
Invariant checking	✓	–	✓	✓
Variables in specifications	✓	–	✓	✓
Non-Boolean variables	✓	✓	✓	✓
Neural-network leaves	✓	–	–	–
Runtime monitors	✓	–	✓§	–
True parallel nodes	–	✓	✓	✓
Time-bounded specifications	†	✓	✓	–
Code generation	✓	–	✓§	–
`pip`/Docker install	✓	–	–¶	–

Table 2: Feature comparison across *BT* verification tools. †: BehaVerify has no per-node duration primitive, but time-bounded properties are recoverable through a tick-counter blackboard variable. ‡: supported by the underlying Fiacre/TINA formalism. §: through the Hippo engine, which compiles a Fiacre model to executable C and supports run-time verification [16, 11]. ¶: the BT2Fiacre toolchain is distributed as source and binaries; its TINA model checker is binary-only. "✓" and "–" denote supported and not supported, respectively.

no native notion of per-node duration. Finally, the BT2BIP examples are small enough that timing comparisons against them carry little information.

Neuro-Symbolic Behavior Trees. BehaVerify supports *NSBT*s, trees with neural-network leaf nodes, following the theory of [27]. A neural network verifier checks open-loop input–output properties of an isolated network [1]. *NSBT* verification instead checks temporal properties of the closed loop formed by the tree, the network and a nondeterministic environment. To our knowledge no other *BT* tool supports neural-network leaves at all.

We report results for a drone that seeks to avoid obstacles on a 10×10 grid with 22 obstacles while navigating to randomly assigned goals. This is the *BT* of Figure 1, modified so that a neural network replaces the A* search algorithm. We train two fully-connected networks, each with one layers of 1000 neurons and ReLU activations. The input is the current position and the goal position (four integers) and the output is one of five actions: the four directions or no move. One network reached 1.0 accuracy on the training data, the other 0.9995. Both networks are used in `table` mode. BehaVerify evaluates each network on all 10^4 combinations of its four inputs and stores the chosen action as a 10 000-row `case` expression, so the model has no real-valued variables and the hidden layer never reaches the model checker. nuXmv then checks the specifications with its standard BDD-based algorithms (`check_invar` and `check_ctlspec`). No abstraction is involved, so the verdicts are exact for these networks. The cost grows with the size of the input domain, which determines the number of table rows, rather than with the size of the network; larger or continuous input domains need the symbolic modes instead. For each network we attempt to verify an invariant and a CTL specification:

$$\begin{aligned} \text{INVAR:} \quad & (x_d, y_d) \notin \mathit{Obs} \\ \text{CTL:} \quad & AG(AF((x_d, y_d) = (x_g, y_g)) \vee (x_g, y_g) \in \mathit{Obs}) \end{aligned}$$

where *Obs* is the set of obstacles. In both cases, the open-loop, obstacle-avoidance invariant was verified.

Task	1.0000 Acc	0.9995 Acc
Translation Time	0.55	0.54
Invariant Time	0.87	0.95
CTL Time	9.24	33.15

Table 3: Timing results for the neural network example implemented as an *NSBT*. All times in seconds, truncated to two decimal places because that is the precision nuXmv reports. **Translation Time** is the time to translate a `.tree` file to a `.smv` file.

		CTL (s)		LTL (s)		Reachable / Total ($\log_2$)	
N	nodes	**FF**	**Naive**	**FF**	**Naive**	**FF**	**Naive**
1	11	0.01	0.01	0.01	0.03	3.17 / 4.58	6.94 / 17.38
2	15	0.01	0.02	0.02	0.08	4.39 / 5.32	8.16 / 21.67
3	23	0.01	0.06	0.06	0.18	5.49 / 6.17	9.49 / 29.42
4	39	0.01	0.14	0.13	0.56	6.54 / 7.09	10.95 / 43.73
5	71	0.02	0.62	0.38	3.54	7.56 / 8.04	12.57 / 70.88
6	135	0.06	4.62	1.31	35.49	8.57 / 9.02	14.32 / 123.48
7	263	0.11	53.03	4.27	TIMEOUT	9.58 / 10.01	16.17 / 226.86
8	519	0.24	TIMEOUT	13.62	TIMEOUT	10.58 / 11.01	TIMEOUT
9	1031	0.57	TIMEOUT	45.75	TIMEOUT	11.58 / 12.00	TIMEOUT
10	2055	2.24	TIMEOUT	153.65	TIMEOUT	12.58 / 13.00	TIMEOUT

Table 4: Fastforwarding-vs-naive ablation on a binary-tree benchmark scaled by depth N (5 min per-instance timeout, times in seconds, state-space columns as $\log_2$). The timing results presented here are the averages calculated over three separate runs. **FF** is BehaVerify's fastforwarding encoding. **Naive** is the same tool with `-use_encoding naive`. Under **FF**, reachable and total states grow linearly with the number of nodes; under Naive, reachable states grow roughly quadratically and the total state space exponentially. For example, at N=7 the naive total state space is already 2^{226} versus FF's 2^{10}. Naive timed out at $N \geq 8$ for both verification and reachability computation.

The closed-loop CTL specification, however, could only be verified for the perfect-accuracy network. For the 0.9995-accuracy network, the counter-example trace repeatedly emitted a non-progressing action over a multi-tick trajectory, so the drone stopped and never reached the target. This closed-loop behavior cannot simply be expressed as an open-loop query about the network alone.

4.1 Fastforwarding vs. Naive Encoding

The scalability advantage over MoVe4BT (50–20 000 nodes, approx. two orders of magnitude) and the simple-robot blackboard-scaling result (grids up to 30×30) were established in [29]. MoVe4BT additionally lacks invariant checking, CTL and variables-in-specifications; its only edge is per-node duration for time-bounded LTL, recoverable in BehaVerify through a tick-counter blackboard variable. Here we add what was *not* ablated in [29]: fastforwarding versus a naive nuXmv encoding within BehaVerify itself. The ablation in Table 4, on a binary-tree benchmark scaling to 2 055 nodes, confirms that fastforwarding is what enables linear growth in reachable states, against roughly quadratic growth in reachable states for the naive encoding.

Figure 6 plots the same benchmark against MoVe4BT as well, placing the encoding ablation of

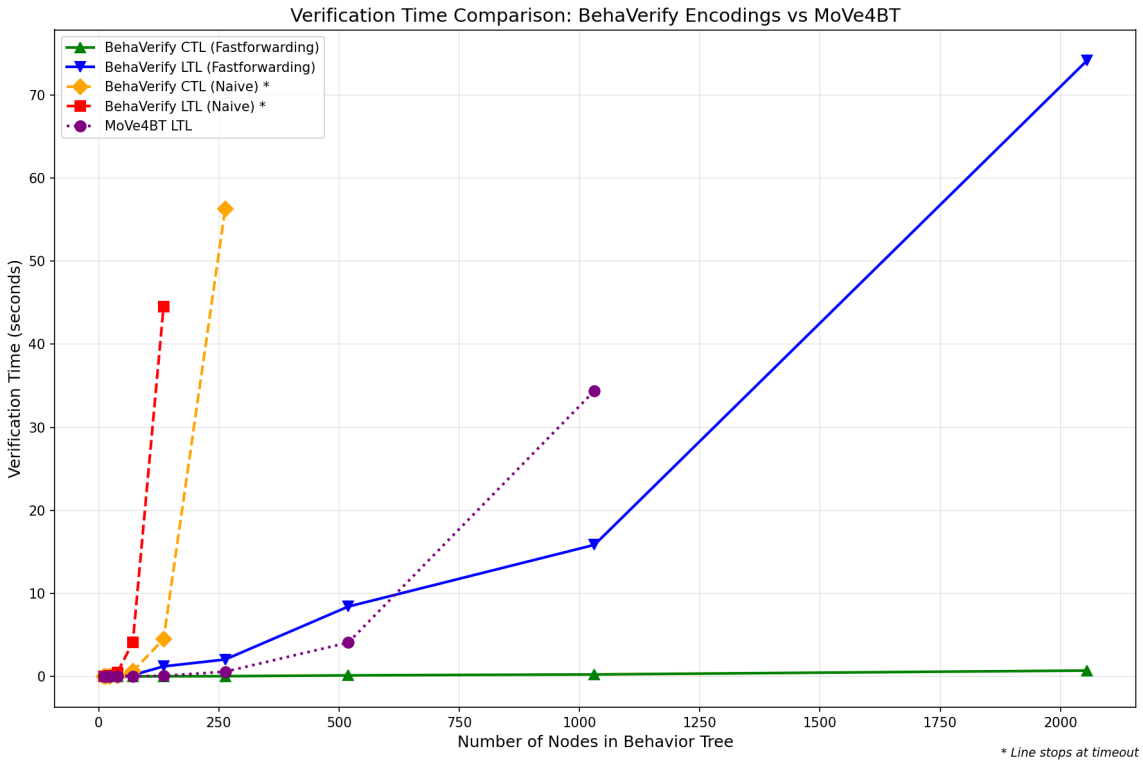


Figure 6: Timing for the binary-tree experiment, comparing BehaVerify's naive encoding, BehaVerify's fastforwarding encoding, and MoVe4BT, with a 5-minute timeout. The node count grows as $7+2^{N+1}$, from 11 to 2 055. At 2 055 nodes MoVe4BT reported `QProcess: Destroyed while process ("mono") is still running`. Times exclude loading time for MoVe4BT and translation time for BehaVerify; the latter is under one second except at the largest size, where it is three seconds.

Table 4 in context: the gap between BehaVerify's two encodings is of the same order as the gap between BehaVerify and an entirely different tool.

4.2 BT2Fiacre: Drone

BehaVerify follows PyTrees semantics whereas `BehaviorTree.cpp` [2] is used by BT2Fiacre (e.g., a PyTrees sequence is a `BehaviorTree.cpp` reactive sequence; `BehaviorTree.cpp`'s sequence-with-memory has no direct PyTrees counterpart), so an exact recreation is not possible. We recreate the UAV example of [18] as `drone3` (closest possible) and `droneNew` (BehaVerify rewrite that moves state from memory-bearing nodes onto the blackboard). Both trees, the integer-grid encoding ($x, y \in [-30, 30]$, height in $[0, 80]$), and the height-bound invariant are in the repository.[8] Table 5 shows BehaVerify is several orders of magnitude faster than BT2Fiacre, which times out when trying to verify the height invariant. `droneNew` further shrinks the reachable state space from $2^{22}/2^{51}$ to $2^{16}/2^{40}$.

BehaVerify additionally falsifies a battery-priority invariant ($Main_{act} \implies ((battery{=}\text{`critical'}) \implies Land_{1,act})$) on `drone3`: because the tree relies on memory-bearing nodes, a *low*-battery navigation event can mask a subsequent *critical* event, and the drone fails to land. This is not an expressiveness gap on BT2Fiacre's part: the flaw is in the tree and is expressible in their framework, so Fiacre should in principle

	BehaVerify		BT2Fiacre	
Task	**drone3**	**droneNew**	**drone3**	**drone4 (droneNew)**
Prep.	0.73	1.14	0.21	0.24
Check Height	86.87	0.48	TIMEOUT	TIMEOUT
Reach. states ($\log_2$)	22.83	16.53	TIMEOUT	TIMEOUT
Total states ($\log_2$)	51.52	39.97	TIMEOUT	TIMEOUT

Table 5: Drone (BT2Fiacre comparison). Times in seconds, averaged over three separate runs. Timeout set to 5 minutes. "Prep." is `.tree`→`.smv` translation for BehaVerify; tree-loading for BT2Fiacre. "Check Height" verifies the height stays within bound. BT2Fiacre's `drone4` is a reactive reformulation analogous to BehaVerify's `droneNew`. `droneNew` restructures `drone3` into a reactive style, shrinking reachable states and total states by approximately 2^6 and 2^{12} respectively.

falsify it too, resources permitting (it timed out on the simpler height invariant). The point is rather that cheap specifications plus fast verification meant more properties actually got checked, which is how the bug surfaced at all. BT2Fiacre did not flag it because their original LTL specification did not require checking the battery. Figure 5 shows the tick in which the property fails; the full counterexample trace is in the repository.

4.3 BT2BIP

We recreated the Mars Rover and Train Control examples of [34] (originally from [5]) in BehaVerify. Both tools agree the Mars Rover spec $storm \implies (Hibernate_{st} = S)$ is false (low power preempts hibernation) and the Train Control spec $(s \implies v{=}0) \wedge (d \implies v{=}0)$ is false (s is never reset). BT2BIP timed out on LTL specifications on the larger trees while BehaVerify completed verification.

5 Conclusions and Future Work

We presented BehaVerify, a tool for formal verification of Behavior Trees. We highlighted the tool's maturity through the contributions of Neuro-Symbolic Behavior Trees and Stateful Behavior Tree support, and the ability to formalize behavior trees in the Domain Specific Language (DSL) through verification and implementation. BehaVerify is scalable as shown via a 2 055 node case study comparison, illustrating BehaVerify's superior scalability to existing methods due to the efficient fastforwarding encoding into nuXmv, versus naive encodings and those used in some other behavior tree verification tools. Future work includes plans to expand the output options for BehaVerify with other frameworks, such as `BehaviorTree.cpp`. The DSL is verbose for end users, so we plan to let the GUI edit leaf behavior and specifications and launch verification, to offer specification patterns or a higher-level syntax for temporal properties, and to run BehaVerify as a language server so that an editor such as VS Code can check a tree while it is being designed. We also plan to compare BehaVerify with BT2Fiacre on a simulated robotic application. We also plan to continue developing and improving the tool overall, as we are using it in several ongoing projects for autonomous systems and robotics applications. We will also explore physical instantiations of the self-correcting neuro-symbolic framework derived in [35].

AI Usage

Generative AI was used to assist with polishing the manuscript; including checking that results presented match the underlying code and validating the accuracy of the features in Table 2 (each also manually checked).

6 Acknowledgments

The material presented in this paper is based upon work supported by the National Science Foundation (NSF) through grant number 2220401, the Defense Advanced Research Projects Agency (DARPA) under contract number FA8750-23-C-0518, and the U.S. Department of Energy, Office of Science, Office of Advanced Scientific Computing Research, under Award Number DE-SC0025528. Any opinions, findings, and conclusions or recommendations expressed in this paper are those of the authors and do not necessarily reflect the views of DARPA, DOE, or NSF.

References

[1] Aws Albarghouthi (2021): *Introduction to neural network verification*. *Foundations and Trends® in Programming Languages* 7(1–2), pp. 1–157, doi:10.1561/2500000051.

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

[3] Bernard Berthomieu, Jean-Paul Bodeveix, Patrick Farail, M Filali, Hubert Garavel, Pierre Gaufillet, Frederic Lang & François Vernadat (2008): *Fiacre: an Intermediate Language for Model Verification in the Topcased Environment*. In: *4th European Congress ERTS Embedded Real Time Software (ERTS 2008)*, SEE : Société de l'électricité, de l'électronique et des technologies de l'information et de la communication and 3AF : Association Aéronautique et Astronautique de France, Toulouse, France, pp. 1–8. Available at https://inria.hal.science/inria-00262442.

[4] Dirk Beyer, Alessandro Cimatti, Alberto Griggio, M Erkan Keremoglu & Roberto Sebastiani (2009): *Software model checking via large-block encoding*. In: *2009 Formal Methods in Computer-Aided Design*, IEEE, pp. 25–32, doi:10.1109/FMCAD.2009.5351147.

[5] 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.

[6] CARVE-ROBMOSYS (2019): *BTCompiler*. Available at https://github.com/CARVE-ROBMOSYS/BTCompiler. Github repository.

[7] 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, doi:10.1007/978-3-319-08867-9_22. Available at http://dx.doi.org/10.1007/978-3-319-08867-9_22.

[8] Edmund Clarke, Daniel Kroening & Flavio Lerda (2004): *A tool for checking ANSI-C programs*. In: *International Conference on Tools and Algorithms for the Construction and Analysis of Systems*, Springer, pp. 168–176, doi:10.1007/978-3-540-24730-2_15.

[9] 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.

[10] Ron Cytron, Jeanne Ferrante, Barry K Rosen, Mark N Wegman & F Kenneth Zadeck (1991): *Efficiently computing static single assignment form and the control dependence graph*. *ACM Transactions on Programming Languages and Systems (TOPLAS)* 13(4), pp. 451–490, doi:10.1145/115372.115320.

[11] Silvano Dal Zilio, Pierre-Emmanuel Hladik, Félix Ingrand & Anthony Mallet (2023): *A formal toolchain for offline and run-time verification of robotic systems*. *Robotics and Autonomous Systems* 159, p. 104301, doi:10.1016/j.robot.2022.104301.

[12] 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.

[13] Lars Grunske, Kirsten Winter & Nisansala Yatapanage (2008): *Defining the abstract syntax of visual languages with advanced graph grammars—A case study based on behavior trees*. *Journal of Visual Languages and Computing* 19(3), pp. 343–379, doi:10.1016/j.jvlc.2007.11.003.

[14] Mattias Hallen, Matteo Iovino, Shiva Sander-Tavallaey & Christian Smith (2024): *Behavior Trees in Industrial Applications: A Case Study in Underground Explosive Charging*. In: *2024 IEEE 20th International Conference on Automation Science and Engineering (CASE)*, pp. 156–162, doi:10.1109/CASE59546.2024.10711822.

[15] 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.

[16] Pierre-Emmanuel Hladik, Félix Ingrand, Silvano Dal Zilio & Reyyan Tekin (2021): *Hippo: A formal-model execution engine to control and verify critical real-time systems*. *Journal of Systems and Software* 181, p. 111033, doi:10.1016/j.jss.2021.111033.

[17] Weijiang Hong, Zhenbang Chen, Minglong Li, Yuhan Li, Peishan Huang & Ji Wang (2024): *Formal Verification Based Synthesis for Behavior Trees*. In Holger Hermanns, Jun Sun & Lei Bu, editors: *Dependable Software Engineering. Theories, Tools, and Applications*, Springer Nature Singapore, Singapore, pp. 72–91, doi:10.1007/978-981-99-8664-4_5.

[18] Felix Ingrand (2025): *A formal implementation of Behavior Trees to act in robotics*, doi:10.48550/arXiv.2502.11904. arXiv:2502.11904.

[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] Steven Macenski, Tully Foote, Brian Gerkey, Chris Lalancette & William Woodall (2022): *Robot Operating System 2: Design, architecture, and uses in the wild*. *Science Robotics* 7(66), p. eabm6074, doi:10.1126/scirobotics.abm6074. Available at https://www.science.org/doi/abs/10.1126/scirobotics.abm6074.

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

[22] Nicholas Potteiger, Ankita Samaddar, Hunter Bergstrom & Xenofon Koutsoukos (2024): *Designing Robust Cyber-Defense Agents with Evolving Behavior Trees*. In: *2024 International Conference on Assured Autonomy (ICAA)*, pp. 1–10, doi:10.1109/ICAA64256.2024.00011.

[23] Bernardo Martinez Rocamora, Paulo V. G. Simplicio & Guilherme A. S. Pereira (2024): *A Behavior Tree Approach for Battery-Aware Inspection of Large Structures Using Drones*. In: *2024 International Conference on Unmanned Aircraft Systems (ICUAS)*, pp. 234–240, doi:10.1109/ICUAS60882.2024.10557083.

[24] Ankita Samaddar, Nicholas Potteiger & Xenofon Koutsoukos (2025): *Out-of-Distribution Detection for Neurosymbolic Autonomous Cyber Agents*. In: *2025 IEEE 4th International Conference on AI in Cybersecurity (ICAIC)*, pp. 1–9, doi:10.1109/ICAIC63015.2025.10849024.

[25] Emily Scheide, Graeme Best & Geoffrey A. Hollinger (2025): *Synthesizing compact behavior trees for probabilistic robotics domains*. *Autonomous Robots* 49(1), p. 3, doi:10.1007/s10514-024-10187-z. Available at https://doi.org/10.1007/s10514-024-10187-z.

[26] 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.

[27] Serena S. Serbinowska, Diego Manzanas Lopez, Dung Thuy Nguyen & Taylor T. Johnson (2025): *Neuro-Symbolic Behavior Trees (NSBTs) and Their Verification*. In: *Proceedings of the International Conference on Neuro-symbolic Systems*, *Proceedings of Machine Learning Research* 288, PMLR, pp. 409–423. Available at https://proceedings.mlr.press/v288/serbinowska25a.html.

[28] Serena S. Serbinowska, Nicholas Potteiger, Anne M. Tumlin & Taylor T. Johnson (2024): *Verification of Behavior Trees with Contingency Monitors*. In Matt Luckcuck & Mengwei Xu, editors: Proceedings Sixth International Workshop on *Formal Methods for Autonomous Systems*, Manchester, UK, 11th and 12th of November 2024, *Electronic Proceedings in Theoretical Computer Science* 411, Open Publishing Association, pp. 56–72, doi:10.4204/EPTCS.411.4.

[29] Serena S. Serbinowska, Preston Robinette, Gabor Karsai & Taylor T. Johnson (2024): *Formalizing Stateful Behavior Trees*. In Matt Luckcuck & Mengwei Xu, editors: Proceedings Sixth International Workshop on *Formal Methods for Autonomous Systems*, Manchester, UK, 11th and 12th of November 2024, *Electronic Proceedings in Theoretical Computer Science* 411, Open Publishing Association, pp. 201–218, doi:10.4204/EPTCS.411.14.

[30] Serena Serafina Serbinowska, Ben Wooding, Diego Manzanas Lopez, Luke Nam, Anne Tumlin, Nicholas K. Potteiger, Hongchao Zhang & Taylor Johnson (2026): *BehaVerify: A Formal Verification Tool for Behavior Trees (software, version v1.0)*. Zenodo, doi:10.5281/zenodo.20280547.

[31] Mingyu Shin & Soyi Jung (2024): *A Survey of Behavior Tree-Based Task Planning Algorithms for Autonomous Robotic Systems*. In: *2024 15th International Conference on Information and Communication Technology Convergence (ICTC)*, pp. 2039–2041, doi:10.1109/ICTC62082.2024.10827191.

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

[33] The Coq Development Team (2024): *The Coq Proof Assistant, Reference Manual*. Available at https://coq.inria.fr/doc/V8.20.0/refman/.

[34] Qiang Wang, Huadong Dai, Yongxin Zhao, Min Zhang & Simon Bliudze (2024): *Enabling Behaviour Tree Verification via a Translation to BIP*. In Diego Marmsoler & Meng Sun, editors: *Formal Aspects of Component Software*, Springer Nature Switzerland, Cham, pp. 3–20, doi:10.1007/978-3-031-71261-6_1.

[35] Ben Wooding, Kiersten Brennan, Anne M Tumlin, Hongchao Zhang & Taylor T Johnson (2026): *A Self-correcting Neuro-symbolic AI Reasoning Framework*. In: *International Symposium on AI Verification*, Springer, pp. 191–211, doi:10.1007/978-3-032-32357-6_9.

[36] Yansong Wu, Fan Wu, Lingyun Chen, Kejia Chen, Samuel Schneider, Lars Johannsmeier, Zhenshan Bing, Fares J. Abu-Dakka, Alois Knoll & Sami Haddadin (2024): *1 kHz Behavior Tree for Self-adaptable Tactile Insertion*. In: *2024 IEEE International Conference on Robotics and Automation (ICRA)*, pp. 16002–16008, doi:10.1109/ICRA57147.2024.10610835.

[37] Igor Zubrycki (2024): *Behavior Tree Based Controller for a Soft Rehabilitation Device Stretchbox*. In: *2024 13th International Workshop on Robot Motion and Control (RoMoCo)*, pp. 168–173, doi:10.1109/RoMoCo60539.2024.10604409.