Verification of Behavior Trees with Contingency Monitors

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

Nicholas Potteiger
0009-0005-0406-0355
Vanderbilt University
Nashville TN, USA
`nicholas.potteiger@vanderbilt.edu`

Anne M. Tumlin
0009-0000-1635-8793
Vanderbilt University
Nashville TN, USA
`anne.m.tumlin@vanderbilt.edu`

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

Behavior Trees (*BT*s) are high level controllers that have found use in a wide range of robotics tasks. As they grow in popularity and usage, it is crucial to ensure that the appropriate tools and methods are available for ensuring they work as intended. To that end, we created a new methodology by which to create Runtime Monitors for *BT*s. These monitors can be used by the *BT* to correct when undesirable behavior is detected and are capable of handling LTL specifications. We demonstrate that in terms of runtime, the generated monitors are on par with monitors generated by existing tools and highlight certain features that make our method more desirable in various situations. We note that our method allows for our monitors to be swapped out with alternate monitors with fairly minimal user effort. Finally, our method ties in with our existing tool, BehaVerify, allowing for the verification of *BT*s with monitors.

1 Introduction

A Behavior Tree (*BT*) is a high-level tree-structured controller with leaf nodes that interact with the environment and interior nodes that control which branches of the tree are executed. The tree-structure means that *BT*s are often more intuitive than equivalent finite state machines, but are also powerful tools capable of being used in many environments. Furthermore, the inherently recursive nature of tree structures allows for adaptability, modularity, and reuse.

*BT*s originated in video games and were used for Non Playable Characters (NPCs). NPCs are, in essence, virtual agents in a digital environment. As time progressed, NPCs needed to respond to more complex environments. The video game industry responded to this by creating *BT*s: designer friendly controllers for complex systems. In light of this, it is unsurprising that the controllers subsequently made the jump to areas such as robotics and drone control. Bipedal locomotion for robots [18], vision measurement systems of road users [26], and swarms of agents have all utilized *BT*s. A recent survey [21] provides even more examples of *BT*s in action.

It is clear that *BT*s are continuing to grow in popularity and usage. As they expand into new domains, especially real-world safety-critical domains, it is imperative to be able to provide guarantees about their correctness. Two methods for providing such guarantees are runtime monitoring and design time verification. Runtime monitoring can be used to alert the *BT* if there is danger of a violation occurring, allowing the tree to self-correct, while design time verification can be used to ensure the model is correct.

At present, tools already exist for the creation of runtime monitors, such as NASA's Copilot [24] and NuRV [7], though they are not designed for *BT*s specifically. However, it is important that the tools not only exist, but be compatible with the *BT*, and that the *BT* reacts correctly to these tools. After all, if the monitor correctly indicates a dangerous situation is occurring but the *BT* ignores this warning, then the danger has not been averted.

The primary contributions of this work are the following:

1. We provide a formal definition for *BT*s with Monitors (*BTM*).
2. We expand the Domain Specific Language (DSL) of BehaVerify [27], allowing it to describe *BTM*s. BehaVerify was originally created for design time verification on *BT*s.
3. We present software for converting Linear Temporal Logic (LTL) specifications written in the DSL into input for the existing tool LTL2BA [16].
   (a) We then translate the output of LTL2BA back into the DSL. This enables BehaVerify to generate nuXmv [6] models, allowing us to use Design Time Verification to confirm that the *BTM* works as intended.
   (b) Additionally, we can translate the output of LTL2BA directly into a C or Python monitor, for use with code generated by BehaVerify.
   (c) Furthermore, we compare the generated C monitors to monitors generated by Copilot [24] and demonstrate our monitors are on par in terms of performance and offer certain improvements in terms of correctness.

2 Related Work

There is a broad body of work utilizing behavior trees for planning purposes in robotic systems [5, 9–11, 22, 31, 35, 36], illustrating their broad usage in safety-critical systems. There are several practical implementations of *BT*s (such as PyTrees [32] and its Robotic Operating System (ROS) extension PyTreesRos, BehaviorTree.cpp [1], and Unreal Engine [13]). Each of these feature a Blackboard (shared memory between nodes). For a variety of practical reasons, our tool targets the implementation of *BT*s presented in PyTrees, though we hope to target BehaviorTree.cpp in the future as well.

There are several existing works that develop and apply formal verification for *BT*s. There are several tools for model verification of *BT*s: [4], BehaVerify [27], BTCompiler, ArcadeBT [19], and MoVe4BT [23]. Prior to this work, however, none of these tools supported Runtime Verification of *BT*s. [8] does runtime verification for a fragment of Timed Propositional Temporal Logic (TPTL) for *BT*s, but failed to configure our examples to work with it.

The authors in [25] utilized the runtime verification framework NASA Copilot [24] to ensure that a flying aircraft maintains an airspeed above a threshold using natural language and Past Linear Temporal Logic (PLTL). Similarly, stream runtime verification (SRV) monitors were generated using HLola [17] in [34] to seamlessly integrate with a UAV hybrid navigation controller for post decision making and online remediation actions. Furthermore, the authors in [33] develop an architecture that allows for construction of runtime monitors that can be integrated into an Urban Air Mobility (UAM) System. They demonstrate that runtime monitors are built using NASA OGMA [25] and NASA FRET [25] for battery monitoring of a UAV simulated in Microsoft AirSim [30]. Runtime monitoring instrumentation frameworks for ROS specifically also have been developed [14]. Our approach generally differs from these works because we are interested in creating monitors for *BT*s specifically. However, our approach also clearly has overlap with several of these methods; we too seek to enable remediation actions for the models being monitored. Furthermore, while we create our own monitors, our general framework is compatible with alternative monitors. In light of this, we compare our monitors to those created by Copilot in Section 5.

Another related aspect is that of the Simplex architecture [2, 28, 29]. The Simplex architecture describes control switching logic swaps between multiple controllers depending on the current state of the system. Our work differs from this approach in two main methods: first our approach is focused explicitly on BTs, and second our approach does not utilize multiple controllers. Instead, we utilize the structure of BTs to integrate the 'switch' into the BT itself.

The work that is most closely related to ours is [8]. This work is about BTs equipped with runtime monitors based on Timed Propositional Temporal Logic (TPTL) and provides a formal definition of the setup. However, unlike our work, the monitors are not meant to be interacted with; the BT has no way of reacting to a violation. In terms of generating monitors by transforming temporal logic specifications to automata, our approach is similar to that of ltl2mon and LamaConv [3, 12, 15], but differs in that we consider BTs. We further differentiate our work through the design time verification aspect. Our method allows us to prove that a BT equipped with a monitor and its contingency response for detected violations is guaranteed to satisfy a specification.

3 Preliminaries

This section provides a formal definition for Linear Temporal Logic and Buchi Automata. This is followed by an intuitive overview of BTs and a formal definition.

3.1 Linear Temporal Logic

A Linear Temporal Logic (LTL) formula is evaluated on a trace. A trace is a sequence of states $Tr \triangleq [n_0, n_1, \ldots]$. Here n_0 is the state at time 0, n_1 is the state at time 1, etc. The grammar of an LTL formula is presented in Grammar 1.

$$\begin{array}{lcll}
\langle\text{LTL}\rangle & ::= & \langle\text{a}\rangle & \text{\#First Order Logic Formula} \\
 & | & \neg\langle\text{LTL}\rangle \,|\, \langle\text{LTL}\rangle \vee \langle\text{LTL}\rangle & \text{\#Minimal Boolean operators} \\
 & | & \bigcirc(\langle\text{LTL}\rangle) \,|\, (\langle\text{LTL}\rangle)\mathcal{U}(\langle\text{LTL}\rangle) & \text{\#Temporal operators next and until}
\end{array}$$

Grammar 1: Minimal LTL Grammar.

We assume the reader is familiar with Boolean logic, so we will not describe them here. $\bigcirc(\phi)$ is true at time t if ϕ is true at time $t+1$. $\phi_1 \mathcal{U} \phi_2$ is true at time t if $\exists t''$ such that $t \leq t''$ and ϕ_2 is true at t'' and $\forall t'$ such that $t \leq t' < t''$, ϕ_1 is true at t'.

In addition to the grammar presented in Grammar 1, we also utilize $\square(\phi)$ (globally) and $\Diamond(\phi)$ (finally). These do not increase the expressiveness of LTL, but make writing formulas easier. $\square(\phi)$ is true at time t if $\forall t'$ such that $t \leq t'$, ϕ is true at time t'. $\Diamond(\phi)$ is true at time t if $\exists t'$ such that $t \leq t'$, ϕ is true at time t'.

Finally, we say that ϕ is true for the entire trace if it is true at time 0. Notationally, we will write $Tr \vDash \phi$ to mean that ϕ is true for the entire trace Tr, $Tr_{[i,j]} \vDash \phi$ if we are looking at the segment of the trace $[n_i, n_{i+1}, \ldots n_j]$, and utilize $\nvDash$ in the same way but to mean not true.

3.2 Buchi Automata

A Buchi Automaton (BA) is a tuple $(Q, \Sigma, \Delta, q_0, F)$.

1. Q is a finite set representing the states BA can be in.
2. Σ is a finite set representing the possible inputs.

3. Δ is a function from $Q \times \Sigma \mapsto 2^Q$. Here 2^Q is the power set of Q. This function describes the nondeterministic transitions available from a given state-input combination.
4. q_0 is an element of Q. This is the initial state.
5. F is a finite set such that $F \subseteq Q$. This is the set of accepting states.

Then, *BA* accepts a given sequence of inputs $[a_0,a_1,...]$ if and only if there exists a sequence $[q_0,q_1,...]$ such that

1. $\forall j \in \mathbb{Z}$ s.t. $j \geq 0, q_{j+1} \in \Delta(q_j,a_j)$
2. $\forall j \in \mathbb{Z}, \exists k \in \mathbb{Z}$ s.t. $j < k \wedge a_k \in F$

Thus we accept if we begin in the initial state, take valid transitions, and enter accepting states an infinite number of times. We accept if any such trace is possible; thus a single trace *can* be used to prove that the input is accepted but it *cannot* be used to prove that the input is *not* accepted.

3.3 Behavior Tree Overview

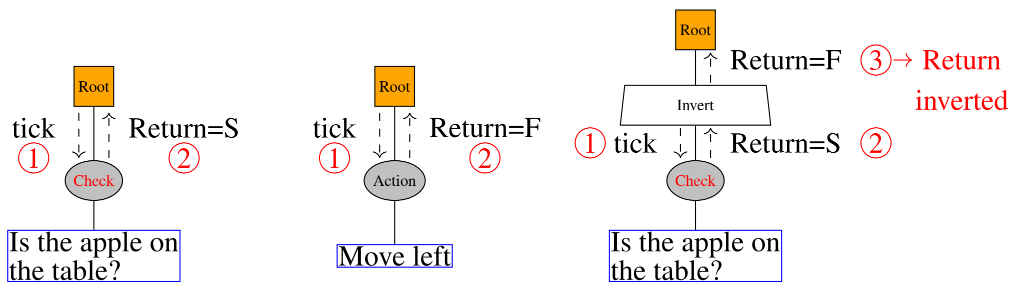
*BT*s are rooted trees with parent-child relationships. Each node has one parent, except the root which has no parent. When executing, a *BT* starts from the root and follows a Depth First Traversal. Nodes can change this traversal order and leave certain branches unexplored based on what their children return. This process is started by an external activation signal called a *tick*. In practice, trees are often structured recursively and parents propagate this signal to their children, but for this paper a *tick* will only be used to refer to the root receiving the external signal. Note that *BT*s are inactive until they receive a tick. In the interest of conciseness, we will omit these periods of inactivity from various diagrams. We assume that a tick will only arrive while the tree in inactive.

Each node is always in one of four states: Success (*S*), Failure (*F*), Running (*R*), or Invalid (*I*). We will use *active* and *executing* to describe where we are in the execution of the tree. When a new tick arrives, each node is set to *I* and the root becomes both active and executing. Until the root finishes executing, exactly one node will be active at all times but more than one node can be executing. A node that is executing is similar to a function that has been called but has not yet turned. A node that is active is similar to a function that is currently being stepped through. When a node finishes executing it returns *S*, *R*, or *F*.

In addition to ticks, we use *timestep* or *t* to track each time the active node changes. Both tick and timestep will be enumerated sequentially starting from 1. Refer to Figure 2 for an example.

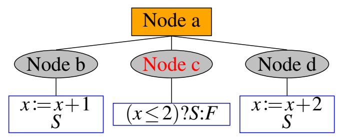
Nodes can be grouped into three categories: leaf, decorator, and composite.

Leaf Nodes Leaf nodes do not have children. It is common to categorize leaf nodes as checks/guards (*e.g.* at boundary?) and actions (*e.g.* go forward). Checks evaluate a boolean condition and return *S* if true and *F* otherwise. Consider Subfigure (a) of Figure 1; if there is an apple on the table, the check will return *S*. If there is not, *F* will be returned. Either way, the status will be returned to the root which will proceed accordingly. It is important to note that checks *only* check a condition and return the appropriate status; they do not set variables or take any sort of action.. By contrast, actions can execute actions, for lack of a better word. For instance, in Subfigure (b) of Figure 1, the action executes the action of moving left. It is important to note that actions are also not restricted in what status they return. While Subfigure (b) of Figure 1 shows *F* being returned, it would be valid to create a version of this action that always return *S*, or it could return *R* because it hasn't finished, or it could return *F* based on some sort of conditional logic.



(a) Check Node (**Leaf**) (b) Action Node (**Leaf**) (c) Inverter Node (**Decorator**)

Figure 1: Example Leaf and Decorator Nodes.



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

Figure 2: A *BT* consisting of a sequence node (a) with two actions (b, d) and a check (c). We use the ternary operator $i?j:k$ to mean if i then j else k. Tick indicates the number of times the tree has been ticked. At each timestep t, the variable x is updated based on the active node. If a node is finished, then it returns one of S, F, or R.

Decorator Nodes Decorator nodes 'decorate' their children, allowing for easy adjustments to be made. A decorator node will always have exactly one child, but that child can have children of its own. An inverter (decorator that swaps S and F) is in Subfigure (c) of Figure 1.

Composite Nodes Composite nodes control the traversal of the tree. Changing a composite node will change the conditions under which branches of the tree are activated. The three primary types of composite nodes are selector, sequence, and parallel nodes. The children of composite nodes are ordered and are activated according to the order, which we will treat as being left-to-right, both visually and in our language.

1. Recall our notation of Success (S), Running (R), and Failure (F).
2. **Selector Nodes:** Selector/Fallback nodes activate their children from left-to-right until one of them returns S or R, at which point the selector itself returns S or R. If a child node returns F, the selector activates the next child. Selectors can be used to prioritize behaviors, ordering them from most preferable at the left to least preferable at the right. Another way to think of these nodes as providing a series of fallbacks in case the intended action fails.
3. **Sequence Nodes:** Sequence nodes activate their children in a left-to-right order, requiring each child to return S before moving on to the next. The sequence returns S only if all its children return S. It returns F or R as soon as one of its children does. This is ideal for defining a series of actions that must be performed in a specific order to achieve a goal. Figure 2 shows an example sequence node with two actions and one check. The figure clearly demonstrates the order in which the children become active. Additionally, it showcases how composite nodes control the flow of

logic in a *BT*; during tick 1, all three children are active at some point. However, during tick 2 node (d) never becomes active because node (c) returned F.

4. **Parallel Nodes:** For the purpose of this paper, we do not consider truly parallel nodes (i.e., nodes that activate all their children simultaneously). Instead, our parallel nodes activate nodes one at a time in a left-to-right order. However, unlike selector and sequence nodes, a parallel node will *always* tick all of its children. Once the last child returns, the parallel node uses a policy that considers what all of the children returned. The two most common policies are Success on All, which requires that all children return S for S to be returned, and Success on One, which requires only one child to return S for S to be returned. This is consistent with [32]. It is possible to define more complex policies, but this is beyond the scope of this paper.

3.4 Formal Definition of Behavior Trees

A Behavior Tree (*BT*) is a tuple $(S,V,\Sigma_T,\Delta_T,s_0,v_0)$.

1. S is a finite set that describes the state of the tree itself (which node is active, what nodes have returned so far, etc).
2. V is a finite set that describes the state of the Blackboard variables (the persistent memory of the tree).
3. Σ_T is a finite set of all possible 'inputs' to the tree (e.g., environmental factors).
4. Δ_T is a function $S \times V \times \Sigma_T \mapsto S \times V$. This function takes the current state of the tree, the variables, and environmental factors and produces a new tree state and new variable values.
5. s_0 is an element of S that describes the initial state of the tree.
6. v_0 is an element of V that describes the initial state of the Blackboard variables.

This definition does not allow for nondeterminism, though it could be simulated through a 'seed' variable. Furthermore, this definition is slightly too permissive; additional restrictions would need to be placed on Δ_T to enforce the tree structure. We omit these details as we believe that they would complicate the issue without providing additional insight.

For a given sequence of inputs $[a_0,a_1,...]$, the corresponding *BT* trace is a sequence $[(s_0,v_0),(s_1,v_1),...]$ such that

$$\forall j \in \mathbb{Z} \text{ s.t. } j \geq 0, \Delta_T(s_j,v_j,a_j) = (s_{j+1},v_{j+1})$$

4 Problem Statement and Methodology

While we have defined *BT*s, we have not defined Behavior Trees with Monitors (*BTM*s). In this section we will first define *BTM*, then formally state the problem we are addressing, and finally present our method for addressing the issue.

4.1 Formal Definition of Behavior Trees with Monitors

A *BTM* is a tuple $(S,V,M,\Sigma_T,\Delta_M,s_0,v_0,m_0)$.

1. S, V, Σ_T, s_0, and v_0 are unchanged from *BT*.
2. M is a finite set describing the state of the monitor.

3. Δ_M is a function $S \times V \times \Sigma_T \times M \mapsto S \times V \times M$. This function takes the current state of the tree, the variables, environmental factors, and the state of the monitor and produces a new tree state, new variables state, and a new monitor state.

4. m_0 is an element of M and describes the initial state of the monitor.

For a given sequence of inputs $[a_0, a_1, ...]$, the corresponding BTM trace is a sequence $[(s_0, v_0, m_0), (s_1, v_1, m_1), ...]$ such that

$$\forall j \in \mathbb{Z} \text{ s.t. } j \geq 0, \Delta_T(s_j, v_j, a_j, m_j) = (s_{j+1}, v_{j+1}, m_{j+1})$$

4.2 Problem Statement

Our problem statement follows. **Given a BT and a ϕ, create a BTM that monitors ϕ such that BTM is capable of reacting to a violation of ϕ.** To demonstrate the utiltity of this process, we will also consider ϕ_1, a second specification that holds for BTM but not for BT.

To accomplish this goal we modified the Domain Specific Language (DSL) of BehaVerify (outlined below) to allow the use of monitors within BTs. We utilize the tool LTL2BA [16] and specifically the implementation at [1] to translate an LTL formula into a monitor in the form of a BA. Then we create an implementation of that BA for use with the BT. Finally, to verify that BTM satisfies the property, we also convert the output of LTL2BA into an implementation of the monitor within the DSL and utilize BehaVerify to create a nuXmv model that proves ϕ_1 is true for BTM but false for BT.

4.3 BehaVerify

BehaVerify uses a Domain Specific Language (DSL) that allows the user to specify a BT. As the DSL itself [2] is complex, we will provide an overview here.

The user defines a finite set of typed variables that will be used by the BT. The user also defines a finite set of leaf nodes. Each leaf node is a finite sequence of statements, exactly one of which is a return statement while the rest are variable statements. A variable statement consists of the variable whose value is being updated and a sequence of 'if statements' that determines the new value. Nondeterminism is allowed in variable statements. The return statement is similar, but is used to determined what status the node will return. Note that the node does *not* stop execution when the return statement is executed; it is only used to determine the return status, not to 'return' from the node. Finally, the user creates a finite tree consisting of composite (selector, sequence, and parallel), decorator (inverter and X_is_Y), and user-defined leaf nodes.

Monitors We add special syntax to our DSL for the creation and use of monitors. The user provides an LTL specification that is to be monitored. Furthermore, the user specifies where in the tree the monitor should be used, and how the tree should react to the possible outputs of the monitor. We describe the details of transforming the monitor in Subsection 4.4.

We present two pipelines for making monitors. The first is for generating a Python implementation of a BT and either Python or C code for the monitor(s). The second pipeline is for generating a nuXmv model of a BT and its monitor(s).

[1] https://github.com/utwente-fmt/ltl2ba

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

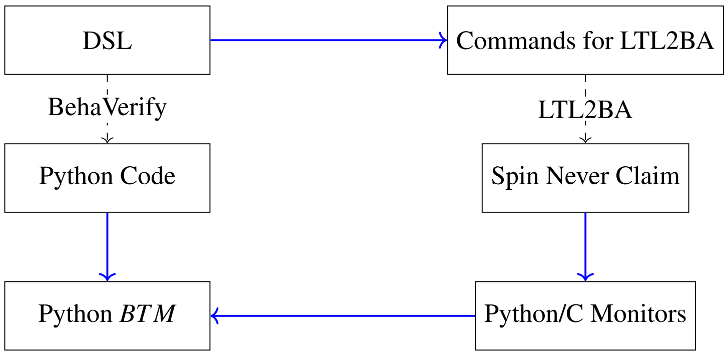


Figure 3: Diagram of how BehaVerify generates Python code for *BT*s with monitors. LTL2BA is a tool for converting an LTL specification to a *BA*. Spin is a model checker, and a Never Claim can be checked using Spin. Solid blue arrows mark new contributions.

4.4 Generating Implementations

We utilize the following process to generate a *BTM* using BehaVerify.

1. The user creates a DSL file specifying the *BT* and any monitors it uses.
2. BehaVerify creates a Python implementation for *BT*, ignoring the monitors.
3. For each monitor in the DSL file, BehaVerify creates a command for use with LTL2BA.
4. For each command, LTL2BA creates a Buchi Automaton (*BA*). The output is in the form of a 'never claim' for use with the Spin [20] model checker.
5. For each *BA*, BehaVerify creates a corresponding Python implementation.
6. The monitors are combined with the generated Python code.

This process can be seen in Figure 3. Below we provide some additional details.

Command Creation LTL2BA commands can contain temporal operators (e.g. 'globally'), boolean operators (e.g. 'and'), boolean constants ('true' and 'false'), and lowercase alphanumeric strings representing boolean variables. Our conversion process prioritizes making the resulting LTL formula as small as possible. Thus the formula $\Box(a \vee b)$ (here $\Box$ means globally) would be converted to $\Box p0$, where $p0$ is boolean predicate representing $a \vee b$.

Monitor We provide a quick refresher on *BA*. For details, see Subsection 3.2. A *BA* is a nondeterministic automaton with transition guards. Thus from a given state, *BA* can transition to any other state provided a transition to that state exists and the associated guard condition is true. Within the formal definition, these guards are encoded into the transition function. Some of the states in the *BA* are 'accepting' states. The *BA* accepts a trace if there exists a sequence where the *BA* is infinitely often in an accepting state.

To mimic this behavior, the monitor takes as input a set of states that the *BA* could be in along with the current model state. The current model state provides all the necessary information to determine if a transition guard is true. This, combined with the possible states, is used to create a new set of possible states. If there are no possible states, then there is no longer any way for the specification to be true, meaning it must be false. If there is a possible state that is an accepting state with a transition to itself and the guard is always true, then the specification is guaranteed to be true. Otherwise, the specification could still prove to be true or false (unknown). The monitor returns both the new set of possible states and the verdict to the user. If a violation occurs, the monitor can be 'reset'. This is important as we want our monitor to be repeatably usable; without a reset it would have to continuously report that a violation occurred.

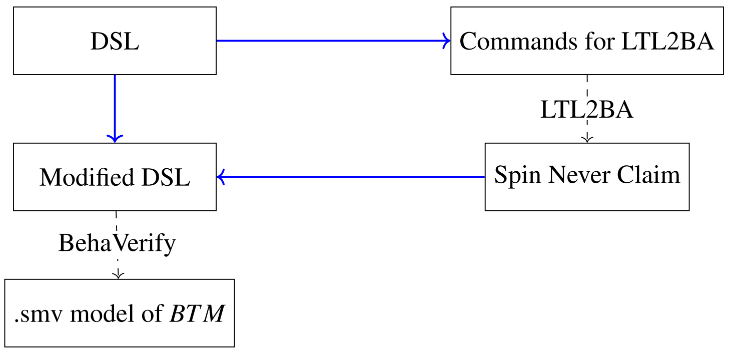


Figure 4: Diagram of how BehaVerify generates .smv files for *BT*s with monitors that can be verified using nuXmv. LTL2BA is a tool for converting an LTL specification to a *BA*. Spin is a model checker, and a Never Claim can be checked using Spin. Solid blue arrows mark new contributions.

4.5 nuXmv Model Generation

The nuXmv pipeline is very similar to the Python pipeline, and as such we will avoid going into the inner workings of this pipeline. The main difference is that we create a monitor using the DSL of BehaVerify allowing BehaVerify to create a .smv model for use with nuXmv.

5 Monitor Comparison

We created two scaling scenarios and ran timing comparisons for the generated monitors created by BehaVerify and Copilot. We planned to compare two monitors generated by NuRV, but did not ultimately do so (see Subsection 5.5 for details). Additionally, we utilized the generated Python code with monitors to generate example traces. While these example traces are not conclusive proof, they were sufficient to demonstrate some differences in the generated monitors. The results demonstrate that the monitors generated by BehaVerify are not outclassed by existing tools and in some cases are preferable from a correctness standpoint. Furthermore, this demonstrates the versatility of our setup; it is fairly painless to bring in outside monitors should the need arise.

The rest of this section will describe the scaling scenarios, present the results, and then reason about the results.

5.1 Scenarios

For our scenarios, a drone navigates a grid and tries to reach a destination. Once the destination is reached, a new destination is randomly generated. We generated grids from size 10 by 10 to 50 by 50 in two styles: dense fixed and sparse random. The dense fixed grids start with a 5 by 5 grid and copies this layout to fill the entire grid. The sparse random grids are randomly generated. See Figure 5 for details.

At each time step, the drone attempts to move towards the target according to the control logic. We want the drone to satisfy the following specifications

$$\phi_S = \Box(\forall (x_o, y_o) \in Obs, (x_d, y_d) \neq (x_o, y_o))$$
$$\phi_L = \Box(\Diamond((x_d, y_d) = (x_g, y_g) \vee \exists (x_o, y_o) \in Obs \text{ s.t. } (x_g, y_g) = (x_o, y_o)))$$

Here *Obs* is a predetermined finite set of obstacles. ϕ_S is a safety specification that states that the drone's position (x_d, y_d) is never equal to the position of an obstacle (x_o, y_o). ϕ_L is a liveness specification that states that the drone's position is eventually equal to the destination (x_g, y_g), or the destination is an obstacle. We utilize the quantifiers $\forall, \exists$ (for all and exists, respectively) for convenience here; in practice we write out each individual obstacle.

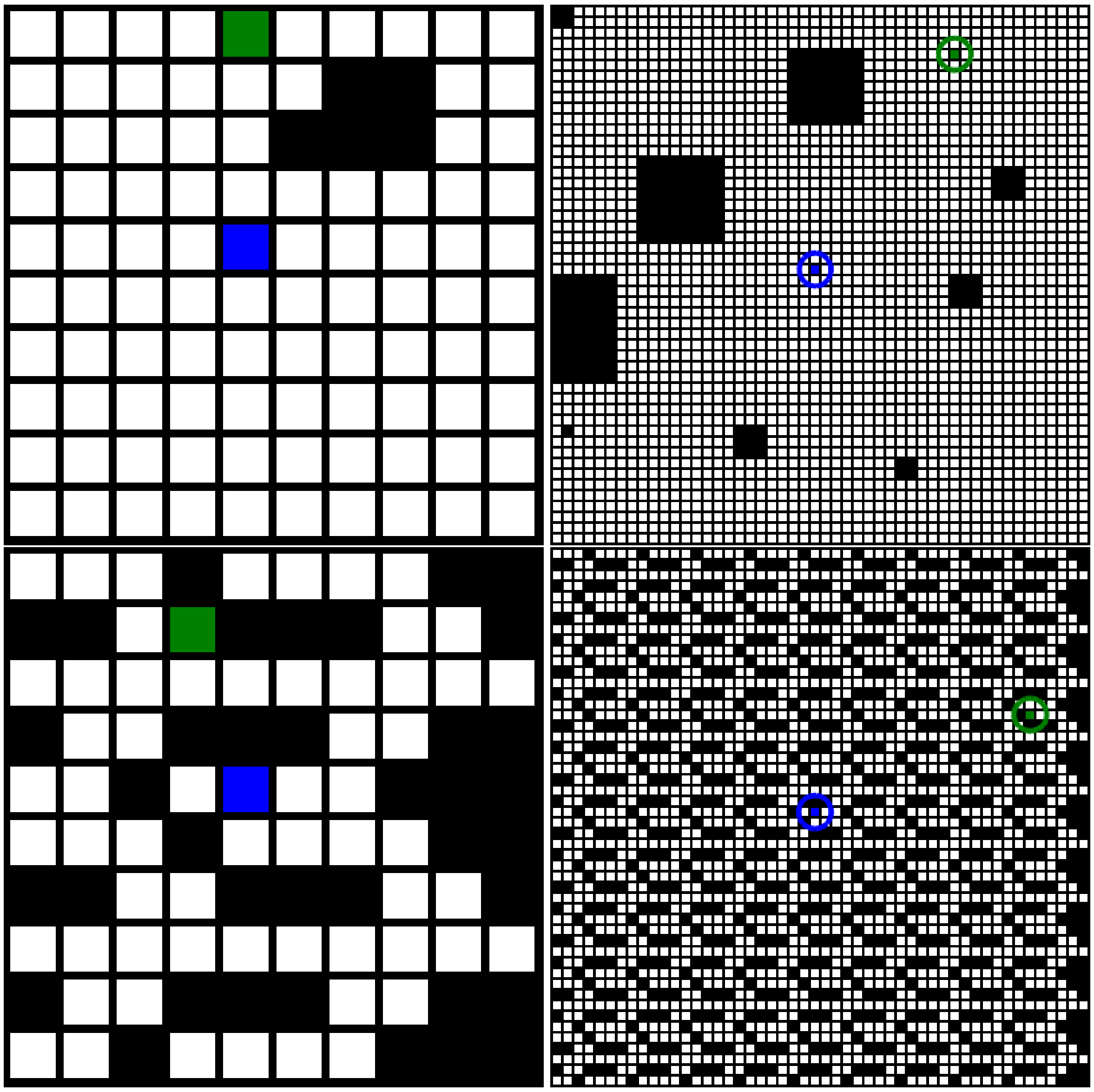


Figure 5: Images representing some of the grids used for the scaling experiments. The upper grids are sparse and were randomly generated. The lower grids are dense and were created by copying a 5 by 5 grid repeatedly. The left grids are 10 by 10 and the right grids are 50 by 50. Black squares are obstacles, the blue square is the drone, and the green square is the destination.

Then the specifications we monitor are

$$\phi_{S1} = \Box(\forall (x_o, y_o) \in Obs, (x_d + (x_\Delta * s), y_d + (y_\Delta * s)) \neq (x_o, y_o))$$

$$\phi_{L1} = \Box \begin{pmatrix} (x_\Delta, y_\Delta) = (1,0) \implies \bigcirc((x_\Delta, y_\Delta) \neq (-1,0)) \wedge \\ (x_\Delta, y_\Delta) = (-1,0) \implies \bigcirc((x_\Delta, y_\Delta) \neq (1,0)) \wedge \\ (x_\Delta, y_\Delta) = (0,1) \implies \bigcirc((x_\Delta, y_\Delta) \neq (0,-1)) \wedge \\ (x_\Delta, y_\Delta) = (0,-1) \implies \bigcirc((x_\Delta, y_\Delta) \neq (0,1)) \end{pmatrix}$$

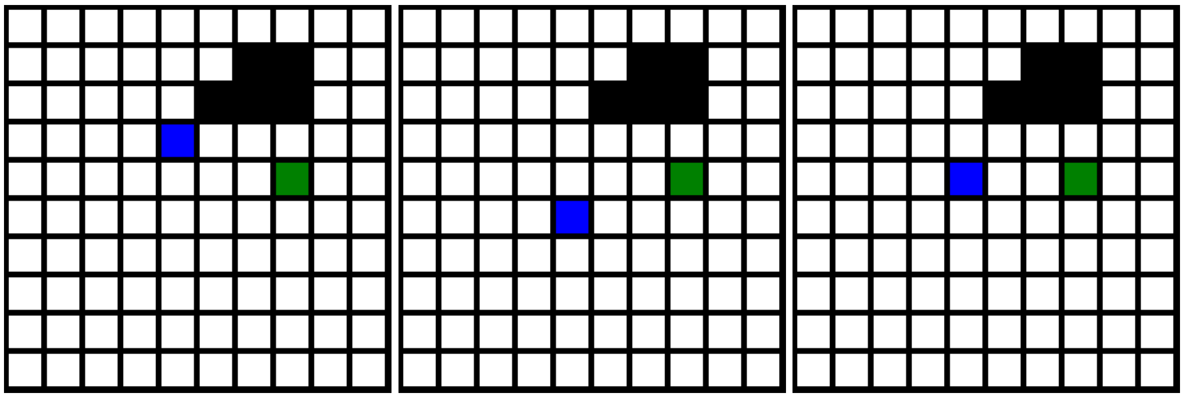


Figure 6: Pictures are ordered left to right. The drone (blue) is trying to reach the destination (green) while avoiding obstacles (black). In the left image, the control logic tells the drone to go down. Because neither monitor reported a violation, the drone moved 2 squares and is now in the situation shown in the middle image. The control logic now tells the drone to go up. The liveness monitor reports a violation; if we go up two squares, the drone will be stuck in a loop. Therefore, the drone only goes up one square and is now in the situation shown in the right image. If the drone has not been equipped with the liveness monitor, it would have gone back to the state shown in the left image, then middle, then left, etc.

(x_d,y_d) is the current location of the drone, (x_Δ,y_Δ) is one of $(1,0)$, $(-1,0)$, $(0,1)$, $(0,-1)$, $(0,0)$, describing the possible directions the drone moves in, and s is the speed of the drone (either 1 or 2). Thus the safety specification monitor is violated if we are on a collision course and the liveness specification monitor is violated if we do a 180 turn. We note that both specifications being monitored are safety specifications; however, we will refer to the second as a liveness specification as it is being used to ensure the liveness specification is not violated. Please note the following: we are *not* claiming that by monitoring these conditions any BT will satisfy the desired specifications; rather, the purpose is to demonstrate that the monitors can be used to correct specific flaws in a BT. Furthermore, it is possible to design and insert these monitors into the BT without the use of a special tool; however, such a task may prove more complex then writing an LTL specification and utilizing our tool.

By default, the drone will try to move 2 squares; if either the safety monitor or liveness monitor are triggered, it will only move one square. The safety monitor ensures that the drone does not move into obstacles. The liveness monitor ensures that the drone escapes potential loops, as seen in Figure 6. Thus both monitors are necessary for the drone to function as intended. Note that this example is not meant to illustrate good programming practice for drone controllers; there are no doubt better methods by which to control a drone. Rather, the purpose of this example is to demonstrate how one can use the monitors to ensure a system functions correctly.

5.2 Motivation

We were originally creating a controller for a drone in virtual neighborhood simulated using AirSim (see Figure 7). The drone would fly at a fixed height and knew where obstacles were before hand. We created a grid-world abstraction of the problem and created a controller that we were able to verify navigated correctly under certain assumptions. One of the assumptions was that the drone would always move at most one tile at a time. While it is simple to ensure that this is the case, it requires flying the drone slowly at all times, which is not desirable. Flying the drone at faster speeds, however, created both safety

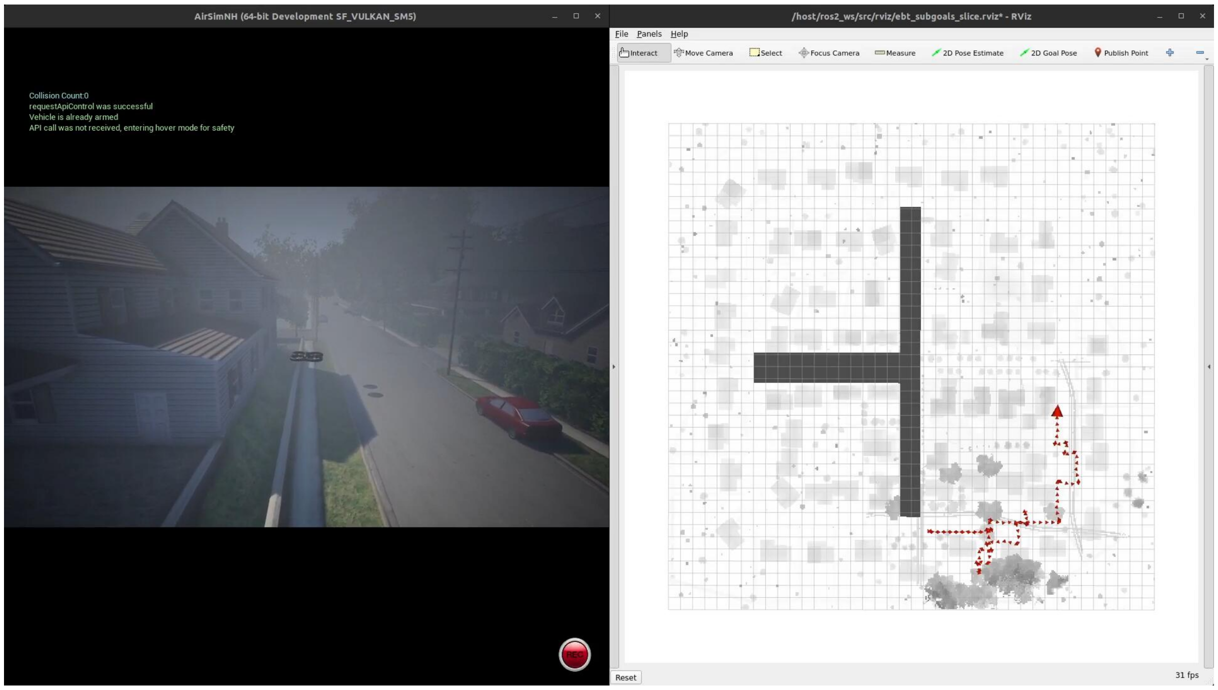

```
Collision Count:0
requestApiControl was successful
Vehicle is already armed
API call was not received, entering hover mode for safety
```

Figure 7: A screenshot of the drone flying in AirSim and a grid visualization.

and liveness violations. Thus by equipping our BT with the described monitors, we were able to safely increase speed without compromising safety. Finally, we were able to utilize nuXmv to verify that the BTM is safe (see Section 6 for details).

5.3 Results

All results (Figure 8) were generated on a computer with a 24 core 13th Gen Intel(R) Core(TM) i7-13700K and 64GB of DDR5 RAM. Results were run in a quiet environment when no other user run processes were active. The code for the experiments is available at [3].

5.4 Analysis

Timing We ran 10000 simulation runs with 1000 iterations (number of times the drone tried to move) and took the median for both Copilot and BehaVerify. To ensure that we are comparing only the monitors, we also ran a version of the code with no monitor with the same settings. This monitorless value was then subtracted out from the timing results for both BehaVerify and Copilot. We believe that the timing results demonstrate that the monitors generated by BehaVerify and Copilot are reasonably close. As expected, the dense fixed grid pattern produces a far more linear timing relation than the sparse random grid. This is because the number of obstacles in the dense fixed grid is always greatly increasing, while the number of obstacles in the sparse random setup is random, and it is the number of obstacles that is largely responsible for the complexity of monitoring the specifications.

[3] https://github.com/verivital/behaverify/tree/main/REPRODUCIBILITY/2024_FMAS_BTM

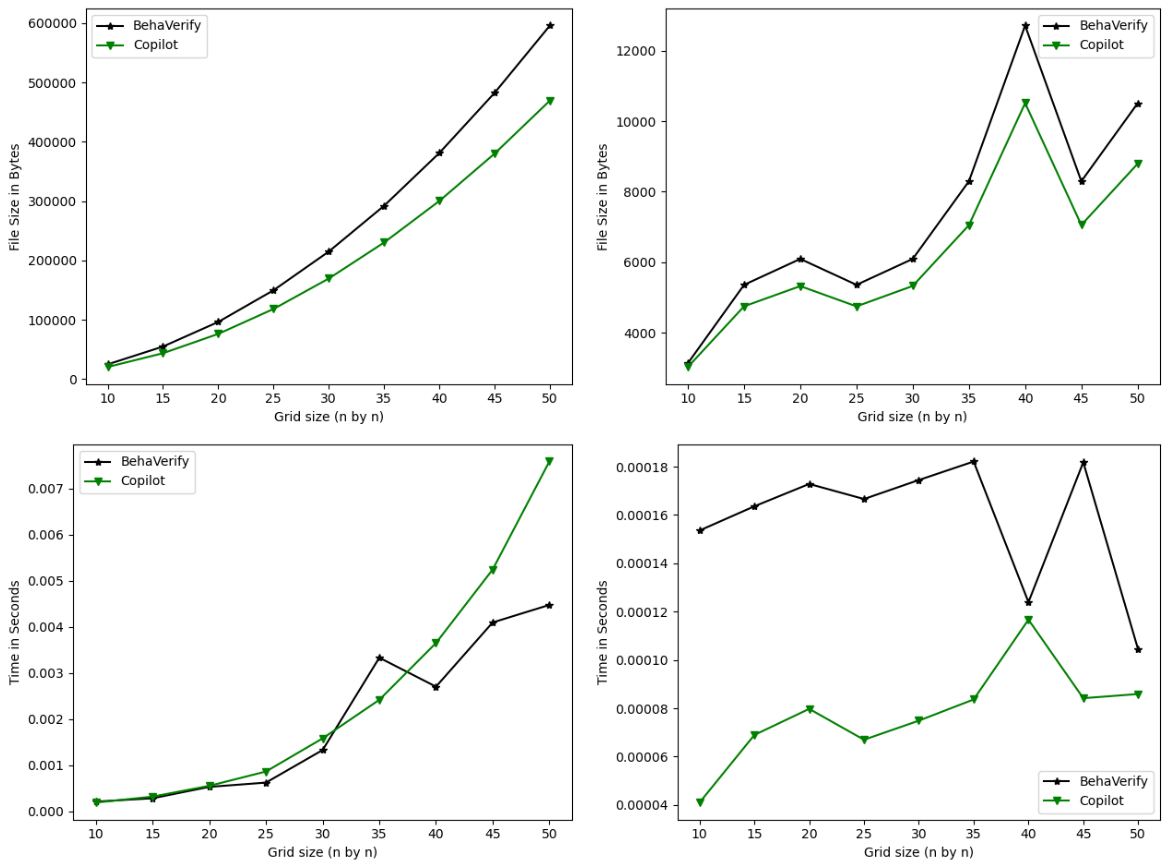


Figure 8: Top left: file sizes of safety monitors for dense fixed. Top right: file sizes of safety monitors for sparse random. Bottom left: median time (in seconds) for safety and liveness monitors on dense fixed. Bottom right: median time (in seconds) for safety and liveness monitors on sparse random. We did not include the size of liveness monitors as it did not change with the size of the grid. The timing results are the median of 10000 runs. Each run had 1000 iterations (number of times the drone tried to move). We subtracted how long it took to run the code without a monitor.

File Size We measured the file size as an indicator of the complexity of the monitoring algorithm. While BehaVerify clearly generates larger files than Copilot, we do not believe that the difference is sufficient to prefer Copilot's Monitors. This is in contrast to NuRV, which is discussed below in Subsection 5.5.

Correctness The safety monitors that Copilot generated worked as intended, but the liveness monitor had an issue: it would report a violation with a one-step delay. Specifically, if a violation occurred in the i^{th} state, then Copilot would report it in the $i+1^{th}$ step. This behavior is documented in the Copilot tutorial in example 7 [4]. By contrast, both the safety and liveness monitors generated by BehaVerify worked as intended, ensuring the drone functioned as intended.

This brings us to an important note about how our process is laid out. While BehaVerify is capable of generating monitors, the generated Python code can utilize *any* provided monitor. Indeed, we confirmed that it is possible to utilize Copilot for the safety monitor and BehaVerify for the liveness monitor and that

[4] https://copilot-language.github.io/downloads/copilot_tutorial.pdf

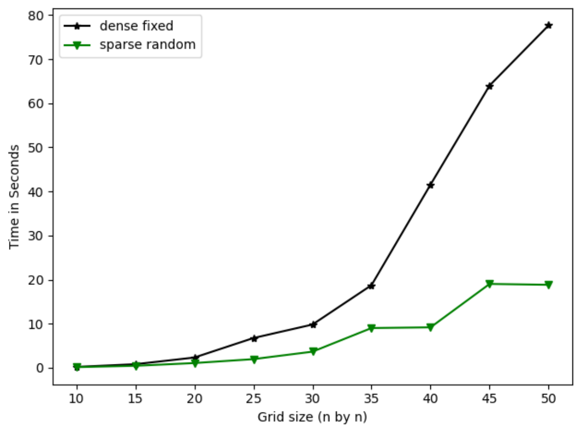


Figure 9: The graph shows the time (in seconds) to verify that the BTM is safe (does not crash into obstacles). The verification was done with nuXmv. Liveness specifications are considerably harder to verify, with the 9 by 9 sparse grid taking about 25 minutes to verify. As this is the easiest grid we have for this task, we did not complete liveness verification for any other grids.

this works as intended.

5.5 NuRV

Because BehaVerify creates .smv files, we originally tried using the output of BehaVerify as input to NuRV. Unfortunately, a variety of issues prevented this from being feasible. For instance, NuRV monitors are aware of the transition system. This enable NuRV monitors to potentially detect violations well in advance or to verify liveness conditions, but it also means the files are much larger. To combat this, we created simplified .smv models with much simpler transition systems. This proved ineffective; the smallest file generated by NuRV was 26.04 MB, while the largest file generated by BehaVerify was 582 KB.

6 Design Time Verification

BehaVerify was originally created for Design Time Verification. As such, when approaching the topic of Runtime Verification, we were interested if we could use Design Time Verification for the Runtime Monitors. As such, we translated the monitors that were created by BehaVerify back into the DSL for BehaVerify and then utilized nuXmv to verify that the BT with monitors satisfied both the safety and liveness specification. While we created such translations for each combination of grid type and grid size presented in Section 5, some of the resulting models proved to complex for liveness analysis in nuXmv. The results for safety verification can be seen in Figure 9. As you can see, it is entirely feasible to use design time verification to confirm that the safety monitors are correct and ensure the system works as intended. If the safety monitor is removed, nuXmv will demonstrate that the system is not safe by providing a counter example trace resulting in a crash. If the liveness monitor is removed, nuXmv will demonstrate that the system can get stuck in a loop by providing a counter example trace.

The liveness situation is somewhat trickier, as the specifications are substantially harder to verify. However, considering the fact that the same liveness monitor is used for all grids and that we verified it for one grid, even this limited verification process can provide some evidence to indicate that the monitor is correct. As with the safety monitor, the removal of the liveness monitor results in a specification violation that nuXmv detects. In this case, nuXmv returns a counterexample where the drone becomes stuck in a loop, going back and forth between two points without reaching the destination.

7 Conclusions and Future Work

We presented a formal problem statement for incorporating contingency monitors within BTs, thus creating BTMs. On the implementation side, we expanded the DSL of BehaVerify to incorporate these monitors, and demonstrated that our code is capable of generating implementations of the monitors that are on par with existing tools. However, our overall approach also brings the advantage of Design Time Verification for the entire BTM. We subsequently hope to expand the target range of BehaVerify, specifically to create .cpp implementations that make use of BehaviorTrees.cpp.

Acknowledgements

The material presented in this paper is based upon work supported by the National Science Foundation (NSF) through grant numbers 2220426 and 2220401, the Defense Advanced Research Projects Agency (DARPA) under contract number FA8750-23-C-0518, and the Air Force Office of Scientific Research (AFOSR) under contract numbers FA9550-22-1-0019 and FA9550-23-1-0135. Any opinions, findings, and conclusions or recommendations expressed in this paper are those of the authors and do not necessarily reflect the views of AFOSR, DARPA, or NSF.

References

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

[2] Stanley Bak, Deepti K. Chivukula, Olugbemiga Adekunle, Mu Sun, Marco Caccamo & Lui Sha (2009): *The System-Level Simplex Architecture for Improved Real-Time Embedded System Safety*. In: *2009 15th IEEE Real-Time and Embedded Technology and Applications Symposium*, pp. 99–107, doi:10.1109/RTAS.2009.20.

[3] Andreas Bauer, Martin Leucker & Christian Schallhart (2011): *Runtime Verification for LTL and TLTL*. *ACM Trans. Softw. Eng. Methodol.* 20(4), doi:10.1145/2000799.2000800.

[4] 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.

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

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

[7] Alessandro Cimatti, Chun Tian & Stefano Tonetta (2019): *NuRV: A nuXmv Extension for Runtime Verification*. In Bernd Finkbeiner & Leonardo Mariani, editors: *Runtime Verification*, Springer International Publishing, Cham, pp. 382–392, doi:10.1007/978-3-030-32079-9_23.

[8] 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.

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

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

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

[12] Antoine El-Hokayem & Yliès Falcone (2018): *Bringing Runtime Verification Home*. In Christian Colombo & Martin Leucker, editors: *Runtime Verification*, Springer International Publishing, Cham, pp. 222–240, doi:10.1007/978-3-030-03769-7_13.

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

[14] Angelo Ferrando, Rafael C. Cardoso, Michael Fisher, Davide Ancona, Luca Franceschini & Viviana Mascardi (2020): *ROSMonitoring: A Runtime Verification Framework for ROS*. In Abdelkhalick Mohammad, Xin Dong & Matteo Russo, editors: *Towards Autonomous Robotic Systems*, Springer International Publishing, Cham, pp. 387–399, doi:10.1007/978-3-030-63486-5_40.

[15] Angelo Ferrando & Vadim Malvone (2022): *Towards the Combination of Model Checking and Runtime Verification on Multi-agent Systems*. In Frank Dignum, Philippe Mathieu, Juan Manuel Corchado & Fernando De La Prieta, editors: *Advances in Practical Applications of Agents, Multi-Agent Systems, and Complex Systems Simulation. The PAAMS Collection*, Springer International Publishing, Cham, pp. 140–152, doi:10.1007/978-3-031-18192-4_12.

[16] Paul Gastin & Denis Oddoux (2001): *Fast LTL to Büchi Automata Translation*. In Gérard Berry, Hubert Comon & Alain Finkel, editors: *Computer Aided Verification*, Springer Berlin Heidelberg, Berlin, Heidelberg, pp. 53–65, doi:10.1007/3-540-44585-4_6.

[17] Felipe Gorostiaga & César Sánchez (2021): *HLola: a Very Functional Tool for Extensible Stream Runtime Verification*. In Jan Friso Groote & Kim Guldstrand Larsen, editors: *Tools and Algorithms for the Construction and Analysis of Systems*, Springer International Publishing, Cham, pp. 349–356, doi:10.1007/978-3-030-72013-1_18.

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

[19] 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.

[20] G.J. Holzmann (1997): *The model checker SPIN*. *IEEE Transactions on Software Engineering* 23(5), pp. 279–295, doi:10.1109/32.588521.

[21] 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.

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

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

[24] Ivan Perez, Frank Dedden & Alwyn Goodloe (2020): *Copilot 3*. Technical Report Technical Report NASA/TM-2020-220587, NASA.

[25] Ivan Perez, Anastasia Mavridou, Tom Pressburger, Alwyn Goodloe & Dimitra Giannakopoulou (2022): *Automated Translation of Natural Language Requirements to Runtime Monitors*. In Dana Fisman & Grigore

Rosu, editors: *Tools and Algorithms for the Construction and Analysis of Systems*, Springer International Publishing, Cham, pp. 387–395, doi:10.1007/978-3-030-99524-9_21.

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

[27] 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.

[28] D. Seto, B. Krogh, L. Sha & A. Chutinan (1998): *The Simplex architecture for safe online control system upgrades*. In: *Proceedings of the 1998 American Control Conference. ACC (IEEE Cat. No.98CH36207)*, 6, pp. 3504–3508 vol.6, doi:10.1109/ACC.1998.703255.

[29] D. Seto & L. Sha (1999): *A Case Study on Analytical Analysis of the Inverted Pendulum Real-Time Control System*. Technical Report, DTIC and NTIS. 10.21236/ADA373286.

[30] Shital Shah, Debadeepta Dey, Chris Lovett & Ashish Kapoor (2018): *AirSim: High-Fidelity Visual and Physical Simulation for Autonomous Vehicles*. In Marco Hutter & Roland Siegwart, editors: *Field and Service Robotics*, Springer International Publishing, Cham, pp. 621–635, doi:10.1007/978-3-319-67361-5_40.

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

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

[33] Alexander Will, Aidan Collins, Robert Grizzard, Smitha Gautham, Patrick Martin, Evan Dill & Carl Elks (2023): *An Integrated Runtime Verification and Simulation Testbed for UAM Hazard Assessment*. In: *2023 53rd Annual IEEE/IFIP International Conference on Dependable Systems and Networks - Supplemental Volume (DSN-S)*, pp. 42–48, doi:10.1109/DSN-S58398.2023.00023.

[34] Sebastián Zudaire, Felipe Gorostiaga, César Sánchez, Gerardo Schneider & Sebastián Uchitel (2021): *Assumption Monitoring Using Runtime Verification for UAV Temporal Task Plan Executions*. In: *2021 IEEE International Conference on Robotics and Automation (ICRA)*, pp. 6824–6830, doi:10.1109/ICRA48506.2021.9561671.

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

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