Composing Multi-Agent Behavior Trees: A DSL Extension for BehaVerify

Jamie Fong
Vanderbilt University
Nashville, TN 37235, USA
jamie.fong@vanderbilt.edu

Taylor T. Johnson
Vanderbilt University
Nashville, TN 37235, USA
taylor.johnson@vanderbilt.edu

Behavior trees (BTs) are widely used in robotics, and although verification techniques for them exist, multi-agent scenarios require tedious hand-coding because existing BT verification tools do not support native multi-agent modeling. We present a DSL extension for BehaVerify that allows users to model multi-agent scenarios through agent type templates and an expansion-based architecture that does not modify the existing code-generation or verification pipeline. The expander instantiates templates prior to code generation, substitutes parameters, and unrolls quantified specifications. This avoids hard-coding while giving a model that the current DSL accepts unmodified. We demonstrate through case studies that verification results are semantically equivalent, and the DSL extension reduces lines of code by up to 58% for a five-agent scenario. This work allows users to model multi-agent scenarios without duplicating code for each agent.

1 Introduction

1.1 Context & Motivation

In recent years, behavior trees (BTs) have become increasingly popular in usage, with libraries such as BehaviorTree.CPP and PyTrees reaching almost ten times their usage from 2018 to 2021[4]. In particular, multi-agent systems have been tested in operations ranging from real-time-strategy games to search-and-rescue operations[6].

With such increasing usage, particularly in important operations, the verification of BTs has also become extremely significant. Because state machines have been the standard model for decades, there has been much literature written about them. The same cannot be said for BTs, with the usage of formal methods to ensure the correctness of BTs remaining scarce[6].

When it comes to multiple agents, each having its own BT, concurrency issues such as deadlocks, livelocks, and collisions can easily arise. If BTs are becoming increasingly prevalent in high-stakes fields, formal verification is paramount prior to deployment. However, this is made challenging by the fact that existing BT verification tools, such as BehaVerify, often assume a single agent[8].

1.2 The Problem

To verify multi-agent scenarios in BehaVerify's DSL, users must manually create duplicate variables, checks, and actions for each robot. Although this is feasible for two agents, this would scale poorly as more agents are added. A user would be practically copying-and-pasting for each robot. To put this in mathematical terms, given N agents, the tree file would require $O(N)$ variable declarations, $O(N)$ check nodes, $O(N)$ action nodes, and $O(N^2)$ pairwise specifications. This becomes increasingly cumbersome, and no other verification tool has native multi-agent support.

Submitted to:
FMAS 2026

© J. Fong & T.T. Johnson
This work is licensed under the
Creative Commons Attribution License.

1.3 Contributions

1. This work builds a backward-compatible DSL extension for BehaVerify. This allows users to specify an `agent_type` parameterized template for robots. Then, it allows users to instantiate these types.
2. We built a pre-processor that runs before the existing pipeline, expanding the multi-agent tree file into the old DSL, allowing the code generators (nuXmv, Python, C++, Haskell) to remain untouched.
3. We introduce gridworld case studies to demonstrate how this extension simplifies verification of multi-agent scenarios with respect to concurrency issues.
4. We demonstrate correctness by showing that the nuXmv results are semantically equivalent to the ones prior to our DSL extension.

2 Related Work

There are three aspects to our work: BT verification, multi-agent systems, and BT DSLs. Work has been done in each of these fields individually, but our work combines all three aspects into one: verifying multi-agent behavior trees natively.

2.1 Behavior Tree Verification

The main work that we seek to extend is BehaVerify, developed by Serena Serbinowska. BehaVerify works by taking stateful BTs described in its DSL, translating them into a .smv file (nuXmv's input language), and exhaustively checking the specifications by checking every reachable state. The main drawback for our purposes is that it does not natively support multi-agent composition.

Another tool developed by Wang et al. verifies BTs by translating BTs into the BIP (Behavior-Interaction-Priority) framework[11]. Within this model, each BT node is transformed into an extended finite state automaton (FSA). This differs from BehaVerify's model of a custom DSL and later using the nuXmv symbolic model checker. Like BehaVerify, this approach does not support multi-agent BT systems.

2.2 Multi-Agent Systems & Concurrency Verification

Verification of concurrent systems has been studied for decades. Tools such as SPIN and UPPAAL are both tools that verify properties of concurrent systems[5, 2]. SPIN models systems using PROMELA, which is then translated into finite automata and verified against specifications written in Linear Temporal Logic (LTL). UPPAAL verifies real-time systems by modeling systems as networks of timed automata, which are then verified against specifications written in a simplified version of Computation Tree Logic. Both SPIN and UPPAAL support template-based designs, which can efficiently model multi-agent systems. This template-based modeling is what our DSL extension seeks to bring to BTs — a feature currently lacking in existing verification tools.

Safety in multi-agent robotics situations has also been studied. Temporal logic and formal verification have been used to ensure safety properties with respect to collisions, deadlocks, and livelocks using LTL specifications. Shoukry et al. handle collisions by taking the LTL specifications and modeling them as a satisfiability modulo convex programming problem that generates a trajectory that avoids collisions[10].

Alonso-Mora et al. take a different approach by synthesizing an automaton from the LTL specification that allows robots to extract valid strategies[1]. Both of these approaches are correct-by-construction models, unlike BehaVerify, which uses model checking after constructing a user-designed BT and specifications. Additionally, with respect to control architecture, Alonso-Mora et al. use FSMs, and Shoukry et al. generate pre-computed trajectories, whereas we use BTs. These correct-by-construction approaches complement our post-hoc verification of BTs, which our work seeks to extend.

2.3 DSLs for Behavior Trees

As mentioned previously, BT frameworks and DSLs such as PyTrees and BehaviorTree.CPP exist and are popular, but lack support for formal verification. Meanwhile, BehaVerify's DSL provides verification, but is limited to single-agent scenarios. Although a user is able to compose multi-agent scenarios, as stated earlier, this does not scale well. Our work intends to extend this DSL to support multi-agent templates, combining previous work done in verification and multi-agent systems with respect to multi-agent BTs.

3 Background

This section will discuss the relevant background needed to understand our DSL extension. It gives a basic overview of what BTs are and relevant aspects of BehaVerify. It discusses aspects of BehaVerify's DSL and how it uses nuXmv to verify BTs.

3.1 Behavior Trees

BTs are a model of a plan of execution, typically used in robotics. As the name suggests, it is a tree with nodes such as composite nodes (sequence, selector, parallel), decorator nodes, and leaf nodes (actions, checks). Within this tree, a signal or "tick" is sent to the root and through the tree until it reaches a leaf node. Each node on the tree returns either `SUCCESS`, `FAILURE`, or `RUNNING`.

Concurrency semantics in BTs vary depending on what framework is used. `py_trees`, for example, has parallel nodes, but the parallelism is conceptual — not true concurrency[7]. The parallel nodes work by sequentially ticking all children within a single tick. The time between ticks being small and the stateful nature of the child nodes give the illusion of concurrency. There is no true concurrency in the sense of multiple threads and processes doing work — though some behaviors may have threads and processes in the background.

3.2 BehaVerify

BehaVerify is a formal verification tool for BTs. The pipeline for it works as follows: a tree file goes through a TextX-based parser. The code generators then take the resulting model and generate the BT in either Python, C++, Haskell, or nuXmv (a symbolic model checker)[3]. Verification works by taking the tree file, generating a nuXmv model which is checked against the specifications. These specifications can be `INVARSPEC` (must be true in every reachable state), `CTLSPEC` (branching-time properties), or `LTLSPEC` (what happens along every execution path). nuXmv then returns true or false, with a counterexample trace if false, for these properties.

BehaVerify's DSL has a `.tree` file with top-level blocks defining variables, environment updates, check and action definitions, tree structure, and temporal logic specifications. For more information, see [9].

BehaVerify's nuXmv code generator (which is used to verify the BT) also utilizes *variable stages*, which represent the value at each point in the tick where the variable may be changed. Because it is possible for one tick to change the value of a variable multiple times, BehaVerify appends each variable name with `_stage_0` at the start of the tick and `_stage_1, _stage_2, ...` for the values after each update in the nuXmv model. This prevents losing intermediate values per tick and the need to use multiple nuXmv transitions per tick, which is slow.

4 Multi-Agent DSL Extension

This section discusses the implementation of our DSL extension. It explains the design choices made, the new syntax that was added, and how this functions in the current pipeline.

4.1 Design Goals & Architecture

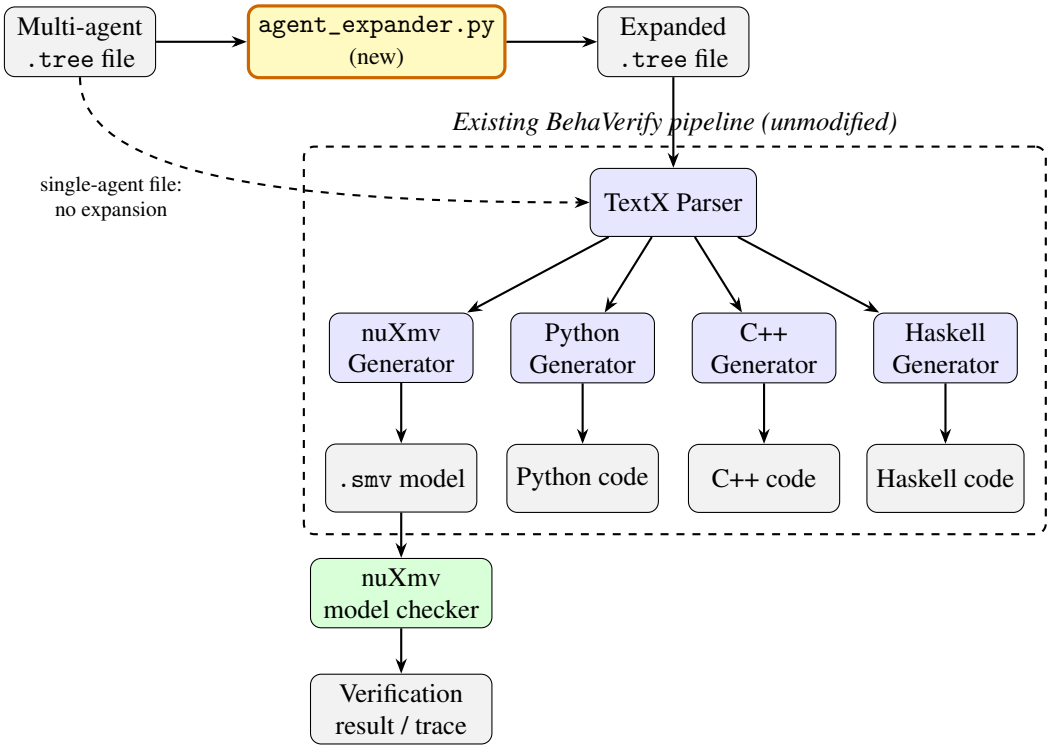

Figure 1: The BehaVerify pipeline with our multi-agent DSL extension. The new `agent_expander.py` (highlighted) pre-processes our multi-agent `.tree` file by expanding it into a single-agent file, which allows the existing pipeline to process it without any modification to the downstream code generators. Non-multi-agent `.tree` files are handled the same way as before and go straight to the parser (dashed line).

The goal of the DSL extension is to allow users to define multiple agents using a single template, with each agent having its own parameters that are resolved at compile time. Additionally, we want to avoid modifying the code generators. In other words, we want to extend the DSL such that a pre-processing step allows the existing pipeline to process it.

This pre-processing step allows us to expand the multi-agent `.tree` file. This contrasts with requiring each code generator to handle multiple agents differently. The code generators are large and complex, so modifying them would be difficult. Adding an extra step in the pipeline prior to the existing one simplifies this process. The idea of expanding early allows us to keep much of the existing pipeline unmodified and prevents the possibility of errors with non-multi-agent trees if the code generators were to be modified.

4.2 Grammar Extension

The new grammar includes the `agent_types` block. This takes `agent_type` definitions with their parameters, variables, environment checks, actions, and tree. When this new grammar is expanded, the parameter values of the `agent_type` block are substituted into the output file. The variables become a state for every agent in the generated nuXmv model. This defines a template to instantiate multiple agents of the same type later. This instantiation is done with the `agents` block that takes the agent name, the type name, and parameters.

New syntax elements include the `[agents]` array size specifier, indexed variables `x_pos[i]`, agent parameter references `agents[i].start_x`, and quantifiers `forall, exists`. Note that all these blocks and elements are optional. Existing `.tree` files continue to function unchanged.

Listing 1: Without DSL Extension

```
variables {
    #{ Two variables for two agents }#
    variable ...
    variable ...
    variable ...
    variable ...
}
environment_update {
    #{ One for each agent }#
    variable_statement ...
    variable_statement ...
}
actions {
    action ...
    action ...
}

tree {
    composite { ...
        #{ Robot 1 }#
        #{ Robot 2 }#
    }
}
```

Listing 2: With DSL Extension

```
variables {
    variable { env x_pos VAR [min_val, max_val] [agents]}
        assign { result {agents[i].start_x} }
}
environment_update {
    variable_statement { x_pos[i]
        ...
    }
}
agent_types {
    agent_type { Robot
```

```
        parameters {
            start_x : [min_val, max_val]
            goal_x  : [min_val, max_val]
        }
        environment_checks {
            ...
        }
        actions {
            ...
        }
        tree {
            ...
        }
    }
}
#{ agent instantiation }#
agents {
    agent { r0 Robot start_x := min_val, goal_x := max_val }
    agent { r1 Robot start_x := max_val, goal_x := min_val }
}
```

These listings show how our DSL extension allows us to define a generic type and instantiate a robot with parameters. Although this change may not be as significant for two robots, with three or more robots, it becomes impractical.

4.3 Expansion Process

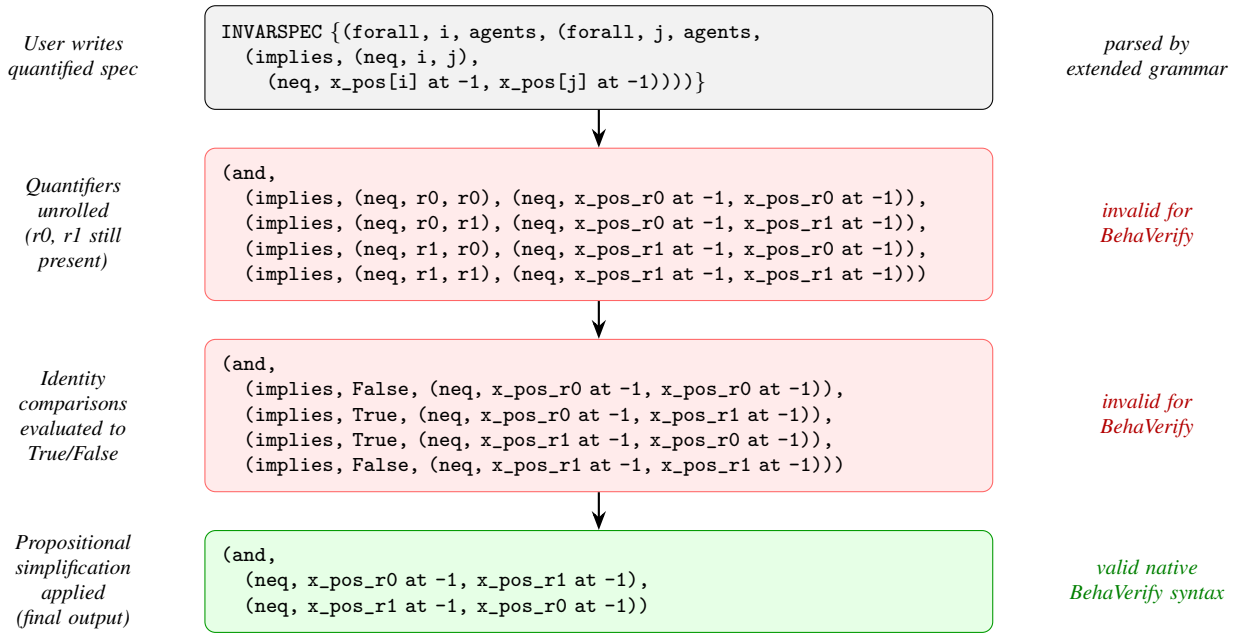

Figure 2: Step-by-step expansion of quantified collision avoidance specification for two agents. Note that the intermediate red never reach the grammar checker. Only the unrolled quantifiers that BehaVerify's existing pipeline accepts are passed. Vacuous truth statements such as the first and last `implies` statements are also resolved and removed.

The expander works by processing the parsed multi-agent file and producing a single-agent model that instantiates each agent's `agent_type`, substituting parameters with their values and labeling generated elements with each agent's name. The environment array variables are expanded so that each agent has its own variables. For example, `x_pos[agents]` unrolls to `x_pos_r0` and `x_pos_r1` for two robots. Each agent also gets its own checks, actions, and subtree. These agents' subtrees are wrapped by a top-level parallel node.

Specifications using `forall` and `exists` are also expanded into conditions for each agent. For instance, for two agents, `(forall, i, agents, P(i))` becomes `P(r0) AND P(r1)`, and `(exists, i, agents, P(i))` becomes `P(r0) OR P(r1)`. When writing these specifications manually for multiple agents, this would be $O(N)$ terms, with pairwise specifications being $O(N^2)$. With our quantified specs, users can write these specifications with a single statement.

A problem with expanding the quantifiers this way is that BehaVerify's type checker does not recognize agent identifiers such as `r0` or `r1` generated by our DSL extension. Thus, we need the expander to evaluate each identity comparison and propagate the truth values, removing vacuous truth statements along the way. Figure 2 illustrates this process step-by-step.

5 Case Studies

This section introduces four case studies in multi-agent scenarios. We give examples of how our DSL extension is used and also show the nuXmv verification results. The verification results confirm our initial predictions about where concurrency issues arise.

5.1 Case 1: 1D Corridor

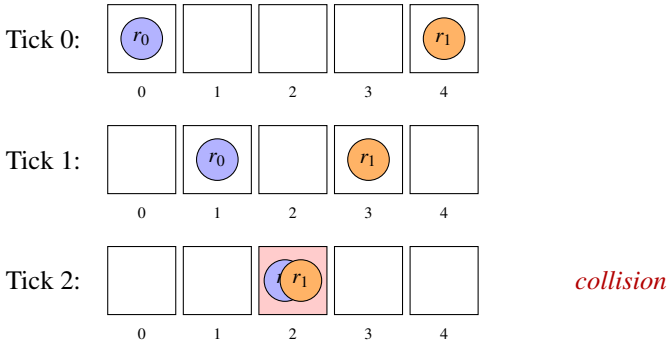

Figure 3: Counterexample trace for the 1D corridor scenario. Robot r_0 (blue) starts at position 0 with goal 4; robot r_1 (orange) starts at position 4 with goal 0. On tick 2, both robots move to position 2, violating the collision-avoidance invariant.

In this scenario, we consider a 1D grid with two robots at different ends of the grid trying to reach each other's positions. Because there is no coordination between the robots, the robots may collide. The BT has a parallel root node with two selector subtrees. Each selector has two children: an `AtGoal` check and a `Move` action. If the robot is at the goal, it does nothing; otherwise, the `Move` node is ticked, which non-deterministically chooses east or west movement.

We set up two specifications for this: `INVARSPEC`, which asserts that each pair of agents cannot be in the same cell simultaneously, and `CTLSPEC`, which asserts that each agent will eventually reach its goal position. See Listing 3 for the code.

Listing 3: 1D Corridor Specification

```
specifications {
    INVARSPEC {(forall, i, agents, (forall, j, agents,
        (implies, (neq, i, j), (neq, x_pos[i] at -1, x_pos[j] at -1))))}
    CTLSPEC {(forall, i, agents, (always_globally, (exists_finally,
        (eq, x_pos[i] at -1, goal_x[i]))))}
}
```

When examining the nuXmv verification results (see Listing 4), we see that it properly detects the collision and states that the invariant was false (line 1). It produces a counterexample trace and properly shows the two robots colliding (see Figure 3 for illustration). Lines 13-14 show that they collide at position 2. nuXmv also confirms the `CTLSPEC`: each robot will eventually reach its goal position (lines 15-16).

Listing 4: 1D Corridor Abridged nuXmv Result

```
1  -- invariant (system.x_pos_r0_stage_1 != system.x_pos_r1_stage_1 & system.
       ↪ x_pos_r1_stage_1 != system.x_pos_r0_stage_1)  is false
2  -- as demonstrated by the following execution sequence
3  Trace Description: AG alpha Counterexample
4  Trace Type: Counterexample
5    -> State: 1.1 <-
6      system.x_pos_r0_stage_0 = 0
7      system.x_pos_r1_stage_0 = 4
8    -> State: 1.2 <-
9      system.x_pos_r0_stage_0 = 1
10     system.x_pos_r1_stage_0 = 3
11     system.act_r0_stage_0 = Ea
12     system.act_r1_stage_0 = We
13     system.x_pos_r1_stage_1 = 2
14     system.x_pos_r0_stage_1 = 2
15 -- specification AG (EF system.x_pos_r0_stage_1 = 4)  is true
16 -- specification AG (EF system.x_pos_r1_stage_1 = 0)  is true
17 }
```

5.2 Case 2: 2D Livelock Scenario

This scenario has a livelock on a 2D grid (see Figure 4). The robots have an environment check `PathClear`, and if `PathClear` fails, they `SidestepSouth` (move south action) if on top row and north otherwise. This will continue indefinitely and fail to reach the goal positions.

`PathClear` is set up by top-level `environment_checks` by indexing the position array with the robot. This is not in the agent template as `PathClear` needs access to the other robots. This is resolved by the expander. For example, `x_pos[r0]` resolves to `x_pos_r0`. See Listing 5 for environment checks.

Listing 5: 2D Livelock Environment Checks

```
environment_checks {
    environment_check { PathClear_r0
        arguments {} read_variables {x_pos, y_pos}
        condition {(not, (and, (eq, x_pos[r1], (add, x_pos[r0], 1)), (eq, y_pos[r1],
            ↪ y_pos[r0])))}}
    environment_check { PathClear_r1
        arguments {} read_variables {x_pos, y_pos}
```

```
        condition {(not, (and, (eq, x_pos[r0], (sub, x_pos[r1], 1)), (eq, y_pos[r0],
            ↪ y_pos[r1])))}}
}
```

The first specification for this scenario checks if, for every path, every robot eventually reaches its goal position. The second specification checks only if, from every reachable state, there exists a path for `r0` to eventually reach its goal. The nuXmv results show that both are false: not every path will reach its goal for all robots and there is no path for `r0` to reach its goal. See Listing 6 for an abridged nuXmv result.

Listing 6: 2D Livelock Abridged nuXmv Results

```
-- specification AG (AF (system.x_pos_r0_stage_1 = 3 & system.y_pos_r0_stage_1 = 1))  is
    ↪ false

-- specification AG (AF (system.x_pos_r1_stage_1 = 0 & system.y_pos_r1_stage_1 = 1))  is
    ↪ false

-- specification AG (EF (system.x_pos_r0_stage_1 = 3 & system.y_pos_r0_stage_1 = 1))  is
    ↪ false
```

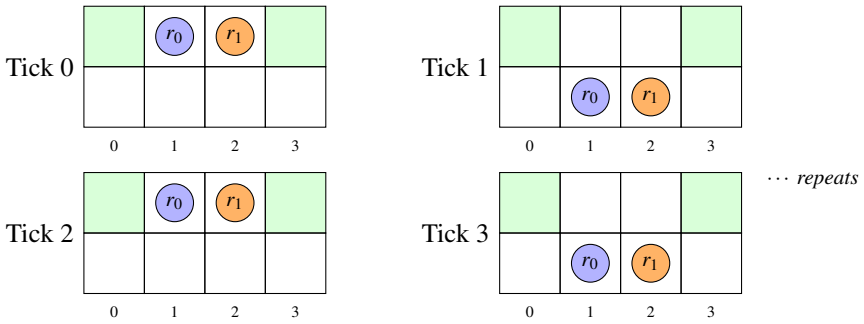

Figure 4: Counterexample for 2D Livelock Scenario. Robot `r0` (blue) and `r1` (orange) have goals at (3,1) and (0,1), respectively. These goal cells are in green. At tick 0, `PathClear` fails, so both robots `SidestepSouth` and move to the bottom row (tick 1). On the bottom row, `PathClear` fails again. `OnUpperRow` also fails, so both robots `StepNorth` and return to the upper row (tick 2). This will continue to repeat indefinitely with both robots failing to reach their goal.

5.3 Case 3: 3 Robots on a Ring

In this scenario, we have a deadlock caused by one robot reaching its goal and blocking the other robots from reaching theirs. Like in case 2, `PathClear` is also used here to check whether the next position (moving clockwise) is clear. The BTs use a selector node to check if the robot is at the goal. If it is, it stays at that position `SetStayGoal`. Otherwise, it uses `PathClear` and moves clockwise. If the path is not clear, it stays in its position and waits `SetStayWait`. In our scenario, `r0` and `r1` are not at their goal, and the path is not clear because `r2` blocks it. Thus, it deadlocks, and they loop indefinitely on `SetStayWait`.

Similar to case 2, we have a top-level environment check. The specifications check for three things: (1) no two robots occupy the same position, (2) every robot eventually reaches its goal, and (3) from every state, each robot can eventually reach its goal. See Listing 7 for these specifications.

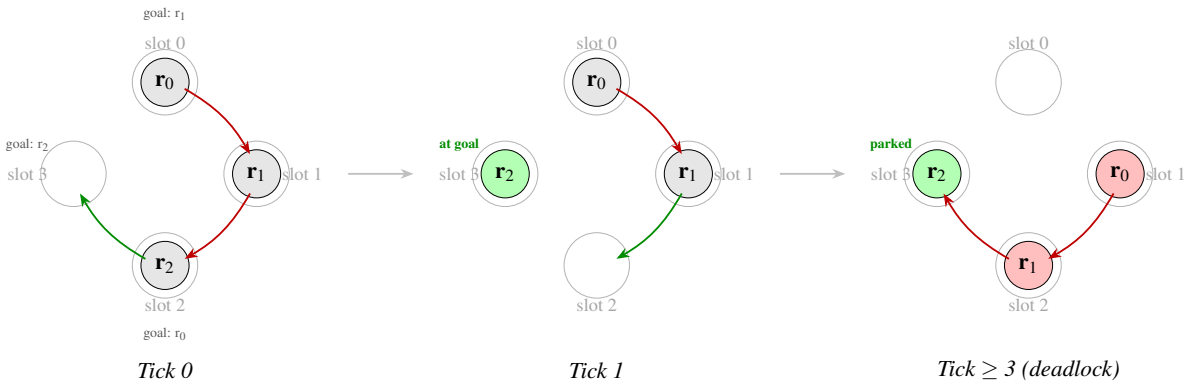

Figure 5: Deadlock on a four-position ring with three robots. Each robot moves clockwise. There is one free slot (slot 3), which is the goal of `r2`. On tick 0, `r2` is able to move to its goal slot because it is free (green arrow), whereas `r0` and `r1` are not (red arrows) as their slots are blocked. After the first tick, `r2` reaches its goal position and remains in that slot. `r1` is then able to move to slot 2, allowing `r0` to move into slot 1. This now permanently blocks `r0` and `r1` from moving clockwise to their goal positions, resulting in a deadlock.

Listing 7: 3 Robots on a Ring Specifications

```
specifications {
    INVARSPEC {(forall, i, agents, (forall, j, agents,
        (implies, (neq, i, j), (neq, x_pos[i] at -1, x_pos[j] at -1))))}

    CTLSPEC {(forall, i, agents, (always_globally, (always_finally,
        (eq, x_pos[i] at -1, goal_pos[i])))))}

    CTLSPEC {(forall, i, agents, (always_globally, (exists_finally,
        (eq, x_pos[i] at -1, goal_pos[i])))))}
}
```

The nuXmv results (see Listing 8) are as expected. The robots do not collide. However, the robots are permanently stuck after `r2` reaches its goal at slot 3. From this deadlocked state, there exists no path for which `r0` and `r1` can reach their goals. `r2`, on the other hand, does reach its goal after the first tick, as the verification results show.

Listing 8: 3 Robots on a Ring Abridged nuXmv Results

```
-- invariant (((system.x_pos_r0_stage_1 != system.x_pos_r1_stage_1 & system.
    ↪ x_pos_r0_stage_1 != system.x_pos_r2_stage_1) & ((system.x_pos_r1_stage_1 != system
    ↪ .x_pos_r0_stage_1 & TRUE) & system.x_pos_r1_stage_1 != system.x_pos_r2_stage_1)) &
    ↪  ((system.x_pos_r2_stage_1 != system.x_pos_r0_stage_1 & system.x_pos_r2_stage_1 !=
    ↪  system.x_pos_r1_stage_1) & TRUE))  is true
-- specification AG (AF system.x_pos_r0_stage_1 = 2)  is false
-- specification AG (AF system.x_pos_r1_stage_1 = 0)  is false
-- specification AG (AF system.x_pos_r2_stage_1 = 3)  is true
-- specification AG (EF system.x_pos_r0_stage_1 = 2)  is false
-- specification AG (EF system.x_pos_r1_stage_1 = 0)  is false
-- specification AG (EF system.x_pos_r2_stage_1 = 3)  is true
```

5.4 Case 4: 5 Robots on a Ring

This scenario is very similar to case 3. Here, we have five agents that move clockwise. `r4` starts next to its goal, slot 5. Thus, on the first tick, `r4` reaches its goal cell. The other agents move clockwise, and each takes the next slot in turn. `r0` to `r3` are then stuck behind `r4`, and they are unable to continue moving clockwise. These robots are permanently deadlocked (see Figure 6).

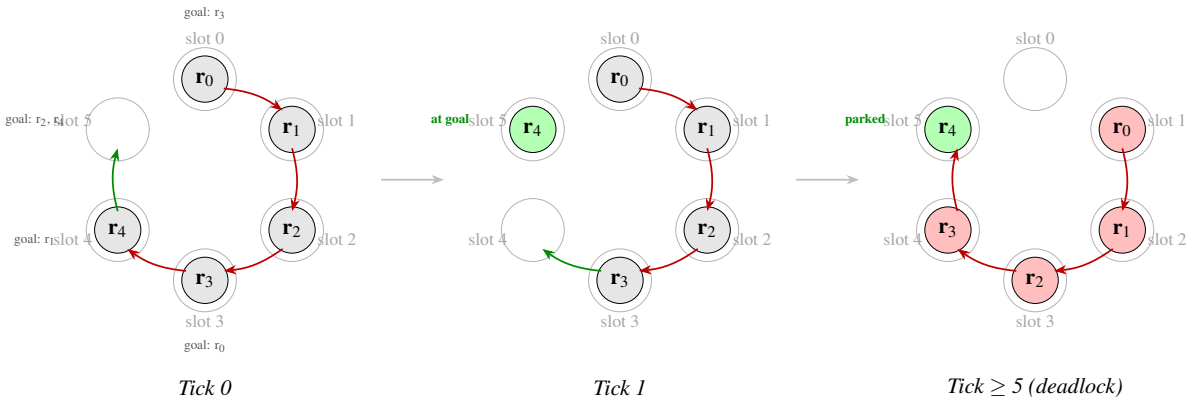

Figure 6: Deadlock on a six-position ring with five robots. Each robot moves clockwise. There is one free slot (slot 5), which is the goal of `r4`. On tick 0, `r4` is able to move to its goal slot because it is free (green arrow), whereas `r0` to `r3` are not (red arrows) as their slots are blocked. After the first tick, `r4` reaches its goal position and remains in that slot. This allows the other robots to continue moving clockwise in turn into the next position, until `r0`–`r3` are eventually stuck.

The specifications are also similar to case 3, and the verification results are as expected. `r4` reaches its goal successfully, and the robots never collide. However, `r0` to `r3` never reach their goal cells. See Listing 9 for the verification results.

Listing 9: 5 Robots on a Ring Abridged nuXmv Results

```
1 -- invariant (((((((system.x_pos_r0_stage_1 != system.x_pos_r1_stage_1 & system.
      ↪ x_pos_r0_stage_1 != system.x_pos_r2_stage_1) & system.x_pos_r0_stage_1 != system.
      ↪ x_pos_r3_stage_1) & system.x_pos_r0_stage_1 != system.x_pos_r4_stage_1) & ((((
      ↪ system.x_pos_r1_stage_1 != system.x_pos_r0_stage_1 & TRUE) & system.
      ↪ x_pos_r1_stage_1 != system.x_pos_r2_stage_1) & system.x_pos_r1_stage_1 != system.
      ↪ x_pos_r3_stage_1) & system.x_pos_r1_stage_1 != system.x_pos_r4_stage_1)) & ((((
      ↪ system.x_pos_r2_stage_1 != system.x_pos_r0_stage_1 & system.x_pos_r2_stage_1 !=
      ↪ system.x_pos_r1_stage_1) & TRUE) & system.x_pos_r2_stage_1 != system.
      ↪ x_pos_r3_stage_1) & system.x_pos_r2_stage_1 != system.x_pos_r4_stage_1)) & ((((
      ↪ system.x_pos_r3_stage_1 != system.x_pos_r0_stage_1 & system.x_pos_r3_stage_1 !=
      ↪ system.x_pos_r1_stage_1) & system.x_pos_r3_stage_1 != system.x_pos_r2_stage_1) &
      ↪ TRUE) & system.x_pos_r3_stage_1 != system.x_pos_r4_stage_1)) & ((((system.
      ↪ x_pos_r4_stage_1 != system.x_pos_r0_stage_1 & system.x_pos_r4_stage_1 != system.
      ↪ x_pos_r1_stage_1) & system.x_pos_r4_stage_1 != system.x_pos_r2_stage_1) & system.
      ↪ x_pos_r4_stage_1 != system.x_pos_r3_stage_1) & TRUE))  is true
2 -- specification AG (AF system.x_pos_r0_stage_1 = 3)  is false
3 -- specification AG (AF system.x_pos_r1_stage_1 = 4)  is false
4 -- specification AG (AF system.x_pos_r2_stage_1 = 5)  is false
5 -- specification AG (AF system.x_pos_r3_stage_1 = 0)  is false
6 -- specification AG (AF system.x_pos_r4_stage_1 = 5)  is true
7 -- specification AG (EF system.x_pos_r0_stage_1 = 3)  is false
8 -- specification AG (EF system.x_pos_r1_stage_1 = 4)  is false
```

```
-- specification AG (EF system.x_pos_r2_stage_1 = 5)  is false
-- specification AG (EF system.x_pos_r3_stage_1 = 0)  is false
-- specification AG (EF system.x_pos_r4_stage_1 = 5)  is true
```

6 Evaluation

This section evaluates how our DSL extension performs on multiple agents compared to before our extension was implemented. We validate our extension by showing the semantic equivalence of our DSL-extended and the hand-written `.tree` files' nuXmv results. We show the benefits of our DSL extension in modeling multi-agent scenarios.

6.1 Semantic Equivalence

Our DSL extension's nuXmv results show semantic equivalence with the hand-written, hard-coded nuXmv results. See Listing 10 for the nuXmv verification results for Case 1: 1D Corridor's manually written version. The verification results are identical, and the counterexample traces differ only in variable names. See Listing 4 for the comparison. One example of this variable name change is when our DSL-extended version uses `x_pos_r0` whereas the handwritten version uses `x_d1`. For case 1, they both find that there is a collision, but both robots can still reach their goals. This semantic equivalence holds for cases 2, 3, and 4 as well. See Listings 11, 12, and 13 for cases 2, 3, and 4, respectively.

Listing 10: Handwritten 1D Corridor Abridged nuXmv Results

```
-- invariant !(system.x_d1_stage_1 = system.x_d2_stage_1)  is false
-- specification AG (EF system.x_d1_stage_1 = 4)  is true
-- specification AG (EF system.x_d2_stage_1 = 0)  is true
```

Listing 11: Handwritten 2D Livelock Abridged nuXmv Results

```
-- specification AG (AF (system.x_d1_stage_1 = 3 & system.y_d1_stage_1 = 1))  is false
-- specification AG (AF (system.x_d2_stage_1 = 0 & system.y_d2_stage_1 = 1))  is false
-- specification AG (EF (system.x_d1_stage_1 = 3 & system.y_d1_stage_1 = 1))  is false
```

Listing 12: Handwritten 3 Robots Abridged nuXmv Results

```
-- invariant ((system.x_pos0_stage_1 != system.x_pos1_stage_1 & system.x_pos1_stage_1 !=
    ↪ system.x_pos2_stage_1) & system.x_pos0_stage_1 != system.x_pos2_stage_1)  is true
-- specification AG (AF system.x_pos0_stage_1 = 2)  is false
-- specification AG (AF system.x_pos1_stage_1 = 0)  is false
-- specification AG (AF system.x_pos2_stage_1 = 3)  is true
-- specification AG (EF system.x_pos0_stage_1 = 2)  is false
-- specification AG (EF system.x_pos1_stage_1 = 0)  is false
```

Listing 13: Handwritten 5 Robots Abridged nuXmv Results

```
-- invariant (((((((((system.x_pos0_stage_1 != system.x_pos1_stage_1 & system.
    ↪ x_pos0_stage_1 != system.x_pos2_stage_1) & system.x_pos0_stage_1 != system.
    ↪ x_pos3_stage_1) & system.x_pos0_stage_1 != system.x_pos4_stage_1) & system.
    ↪ x_pos1_stage_1 != system.x_pos2_stage_1) & system.x_pos1_stage_1 != system.
    ↪ x_pos3_stage_1) & system.x_pos1_stage_1 != system.x_pos4_stage_1) & system.
    ↪ x_pos2_stage_1 != system.x_pos3_stage_1) & system.x_pos2_stage_1 != system.
    ↪ x_pos4_stage_1) & system.x_pos3_stage_1 != system.x_pos4_stage_1)  is true
-- specification AG (AF system.x_pos0_stage_1 = 3)  is false
-- specification AG (AF system.x_pos1_stage_1 = 4)  is false
-- specification AG (AF system.x_pos2_stage_1 = 5)  is false
```

```
-- specification AG (AF system.x_pos3_stage_1 = 0)  is false
-- specification AG (AF system.x_pos4_stage_1 = 5)  is true
-- specification AG (EF system.x_pos0_stage_1 = 3)  is false
-- specification AG (EF system.x_pos1_stage_1 = 4)  is false
-- specification AG (EF system.x_pos2_stage_1 = 5)  is false
-- specification AG (EF system.x_pos3_stage_1 = 0)  is false
```

6.2 Conciseness & Scalability

Manually handwriting each agent and its respective parameters and actions does not scale well past 2 agents, given the duplication needed with each agent's variables, checks, and actions, as well as pairwise specifications, which grow quadratically. This is especially seen in Case 4: 5-Robot Ring, which requires ten pairwise inequalities compared to the single quantified statement in our extended DSL. The DSL extension significantly reduces line count as the number of agents and complexity of the scenario increase. Table 1 compares the total lines of code in the `.tree` for each of the case studies in section 5 (not counting lines with only closing braces, blank lines, or comments). As can be seen, the reduction is not as significant for our two agents 1D corridor scenario because there were few variables per-agent. Because of this, the templating took up most of the lines saved. However, the reduction is significant for 3 and 5 agents. This trend suggests that growing the fleet of agents will cause this reduction to increase even more, especially given the $O(N^2)$ growth for pairwise specifications.

Table 1: Line count comparing hand-written and DSL-extended `.tree` files.

Scenario	Hand-written	DSL	Reduction
1D Corridor (2 agents)	50	47	6.0%
2D Livelock (2 agents)	108	77	28.7%
3-Robot Ring (3 agents)	142	80	43.7%
5-Robot Ring (5 agents)	243	102	58.0%

7 Discussion & Future Work

7.1 Limitations

In this section, we will discuss the limitations of our DSL extension as well as underlying issues that continue to make verifying multi-agent BTs difficult and hard to scale.

7.1.1 Cross-Agent References

One limitation that can be seen even in the earlier case studies is the inability to reference other agents abstractly within a template. Each agent template is only able to reference its own state (e.g., `x_pos[self]` and `x_pos[i]` are supported but not `x_pos[other]`). The only way to name another agent is to hard-code its identifier. This comes out most clearly in Case 2: 2D Livelock Scenario. Each robot needed to check the cell ahead of it to see if another robot is there using `PathClear`. There was no way to write `(eq, x_pos[other], (add, x_pos[self], 1))`. Our workaround was to have top-level `environment_checks` for each agent. See Listing 5 for the two PathClears, `PathClear_r0` and `PathClear_r1`.

This workaround is more feasible with two agents where there is no ambiguity about the "other" agent. Top-level checks work well here and are not tedious to write. However, with three or more agents, this becomes difficult to hand-write as each agent needs to consider every other agent. Ultimately, given N agents, we would need N hand-written checks enumerating the $N-1$ other agents. This gives $O(N^2)$ hand-written code. Although this means there will still be some duplication of code, Section 6.2 shows that the reduction of code compared to the hand-written approach is still significant.

7.1.2 State-Space Scalability

State-space explosion is not a limitation of our DSL extension itself. However, the problem with verifying multi-agent BTs is that the nuXmv model checker may exceed memory and time limits as the complexity of our model and the number of agents increase. Although this did not impact our case studies as the complexity of the trees and number of agents are relatively low, scaling to larger fleets of robots may suffer from this limitation. BehaVerify's fast-forwarding mechanism mitigates state-space growth and has been shown to optimize the verification process — performing better than other tools[9].

7.1.3 Heterogeneous Agent Types

Our current DSL extension theoretically supports heterogeneous configurations through multiple `agent_type` blocks. However, this has not been tested, and our case studies are all homogeneous fleets of a single type. Because real-world scenarios such as leader-follower setups require heterogeneous fleets, there may be existing edge cases that our expander does not currently handle correctly.

7.2 Future Work

Addressing the limitation in 7.1.1, future work can extend the DSL to allow quantified cross-agent references. This would allow users to write `(forall, other, agents, (neq, other, self), <condition involving x_pos[other]>)`. The expander should then be able to unroll it for each agent. The unrolling and identity simplification already exist in our expander, and similar logic would be used here to unroll quantified template checks. However, edge cases such as nested quantifiers in template bodies, quantifiers over a subset of agents, and interactions with parameters would require more careful design.

As for the heterogeneous agent type limitation, further case studies that include heterogeneous configurations with multiple `agent_type` blocks are needed to confirm that our DSL extension is valid for heterogeneous fleets. This would allow users to model more complex and real-world scenarios. Potential case studies include leader-follower setups and sensor-actuator teams.

8 Conclusion

We presented a DSL extension for BehaVerify that allows users to natively model multi-agent BTs through templates and a pre-processing architecture. We showed through our case studies that the verification results of our hand-written multi-agent files are semantically equivalent to the DSL-extended files. We also showed that there was a dramatic reduction in the lines of code when using our DSL extension. From our case study scenarios, there is up to a 58% reduction in the lines of code for a five-agent scenario. However, our DSL extension still does not allow us to template cross-agent references, and

there is testing to be done with heterogeneous types. Our work is a step towards practical verification of multi-agent BT systems.

References

[1] Javier Alonso-Mora, Jonathan A. DeCastro, Vasumathi Raman, Daniela Rus & Hadas Kress-Gazit (2018): *Reactive mission and motion planning with deadlock resolution avoiding dynamic obstacles*. *Autonomous Robots* 42(4), pp. 801–824, doi:10.1007/s10514-017-9665-6. Available at `https://doi.org/10.1007/s10514-017-9665-6`.

[2] Gerd Behrmann, Alexandre David & Kim G. Larsen (2004): *A Tutorial on Uppaal*, pp. 200–236. Springer Berlin Heidelberg, Berlin, Heidelberg, doi:10.1007/978-3-540-30080-9_7. Available at `https://doi.org/10.1007/978-3-540-30080-9_7`.

[3] 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 Armin Biere & Roderick Bloem, editors: *Computer Aided Verification*, Springer International Publishing, Cham, pp. 334–342.

[4] Razan Ghzouli, Thorsten Berger, Einar Broch Johnsen, Andrzej Wasowski & Swaib Dragule (2023): *Behavior Trees and State Machines in Robotics Applications*. *IEEE Transactions on Software Engineering* 49(9), p. 4243–4267, doi:10.1109/tse.2023.3269081. Available at `http://dx.doi.org/10.1109/TSE.2023.3269081`.

[5] Gerard Holzmann (1997): *The Model Checker SPIN*. *IEEE Transactions on Software Engineering* 23, pp. 279–295.

[6] Matteo Iovino, Edvards Scukins, Jonathan Styrud, Petter Ögren & Christian Smith (2020): *A Survey of Behavior Trees in Robotics and AI*. arXiv:2005.05842.

[7] py_trees: *py_trees Documentation*. `https://py-trees.readthedocs.io`. Accessed: 2026-04-09.

[8] Serena S. Serbinowska & Taylor T. Johnson (2024): *BehaVerify: Verifying Temporal Logic Specifications for Behavior Trees*. arXiv:2208.05360.

[9] Serena S. Serbinowska, Preston Robinette, Gabor Karsai & Taylor T. Johnson (2024): *Formalizing Stateful Behavior Trees*. *Electronic Proceedings in Theoretical Computer Science* 411, p. 201–218, doi:10.4204/eptcs.411.14. Available at `http://dx.doi.org/10.4204/EPTCS.411.14`.

[10] Yasser Shoukry, Pierluigi Nuzzo, Ayca Balkan, Indranil Saha, Alberto L. Sangiovanni-Vincentelli, Sanjit A. Seshia, George J. Pappas & Paulo Tabuada (2017): *Linear temporal logic motion planning for teams of underactuated robots using satisfiability modulo convex programming*. In: *2017 IEEE 56th Annual Conference on Decision and Control (CDC)*, pp. 1132–1137, doi:10.1109/CDC.2017.8263808.

[11] 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.