Canopy: Coverage Measurement for Behavior Trees

Samuel Hodges, John-Paul Ore[*†]

***Abstract*—Behavior Trees are an increasingly popular way to orchestrate high-level robot autonomy, but we lack ways to quantify coverage. This leaves robot system developers in the dark about how much of a behavior tree is exercised during system execution and what remains untested. In this work, we adapt ideas from model-based testing and propose a coverage metric tailored to behavior trees, enabling robot developers to focus system testing efforts on measurable increases to system test coverage. We implement our metric in Canopy, a lightweight, open-source tool that integrates with existing ROS2 systems with minimal toolchain impact. Canopy works with the two most popular behavior tree libraries, BehaviorTree.cpp and py_trees_ros. Our work contributes to efforts to deepen robotic software test tooling for model-driven engineering. To demonstrate how Canopy works, we apply Canopy to Navigation2 in a case study, and show how the results can guide robot test engineers seeking to improve their test suites. Our tool is available at https://github.com/RobotCodeLab/BT-Canopy**

***Index Terms*—Methods and Tools for Robot System Design; Software Tools for Robot Programming; Software Tools for Benchmarking and Reproducibility**

I. Introduction

Behavior Trees (BTs) [1] are a high-level and human-interpretable way to organize autonomy. From BT's origins in video games [2], BTs now are both guiding real-world robots in industry [3] and at the heart of cornerstone open-source robotic software like the impressive *Navigation2* (Nav2) [4]. The robot shown in Figure 1 is a simulated TurtleBot3[1] robot utilizing a Nav2 BT for waypoint navigation. Currently, robot system developers would have to build their own logging infrastructure if they wanted to understand how this system utilizes its BT. Popular BT implementations, like `BehaviorTree.cpp` and `py_trees_ros`, are often customized to fit specific needs [3]. Regardless of their application and customization, testing systems that use BTs, like testing robotic systems more generally, is challenging [5].

There are many factors that make testing robotic systems painful, including a dearth of automated testing tools and a bewildering landscape of testing frameworks for unit, integration, and systems tests compounded by continuous integration and build farms. All of these testing methods can contribute to reducing defects and finding bugs early, but robotic system tests are a particular challenge [5]. Robotic system developers sometimes rely on experience or intuition when building tests and require in-depth knowledge of the System Under Test (SUT).

[*]This work is supported by NSF-NRI-USDA-NIFA 2021-67021-33451 and North Carolina State University funds.

[†]Samuel Hodges and John-Paul Ore are with the Department of Computer Science, North Carolina State University, Raleigh, NC 27695, USA (email: {sshodge2, jwore}@ncsu.edu)

[1]TurtleBot3: https://www.turtlebot.com/

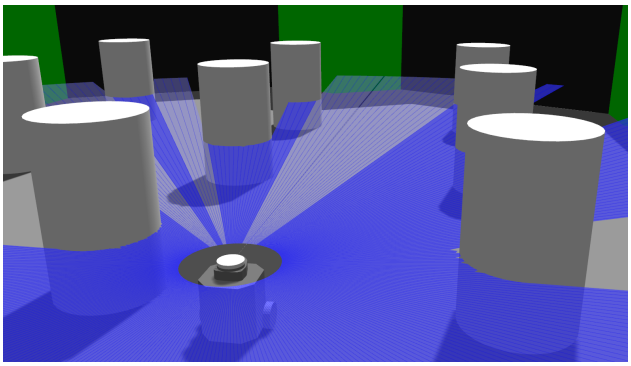

Fig. 1. Turtlebot3 with lidar simulated in Gazebo running the Behavior Tree from Nav2's test suite.

Although previous efforts for testing robotic systems have impressively strong theoretical guarantees using *assume-guarantee* reasoning and composition [6], these techniques require domain-specific languages and impose significant toolchain constraints, like special compilers and preprocessors. Other work in robot system coverage, such as *situation coverage* [7], relies on a catalog of previous experiences that are specific to each system. In contrast, this work proposes to come to roboticists where they are and re-examine coverage in the light of tools already used by roboticists, like BTs.

BTs are a kind of model, specifically a *model-at-runtime* [3], and BTs are part of a larger trend toward *model-driven engineering* [8]. Because BTs are a model, we can approach BT coverage using concepts from *model-based testing* [9], a well-trodden path in software engineering research. In model-based testing, a state-transition model is the mathematical object used to guide and direct testing. The mathematical object of a BT is a *tree graph*, with nodes and edges. This study investigates the feasibility of using a robot system's BT to quantify its coverage in the following research questions:

RQ1 Do existing testing approaches adequately address behavior tree coverage?

RQ2 To what degree is Navigation 2's behavior tree covered by its test suite?

To answer **RQ1**, we start by examining existing node and edge-based graph coverage metrics and find that existing concepts of coverage are close, but not completely sufficient, to fit the need of BTs. BTs, unlike other models, are built with several different 'statuses:' 'running,' 'failure,' and 'success.' Therefore, we present a new form of coverage we call ***BT status coverage***.

Coverage metrics are important because they give us a

way to quantitatively compare different tests and to measure whether new tests add to what is already tested in a test suite (a collection of tests). We envision that robot system developers can use our tool, CANOPY[2], as a systematic and quantifiable way to design and deepen robot system testing. CANOPY is a straightforward logging and aggregation tool, built on well-researched ideas from model-based testing. We utilize CANOPY to answer **RQ2** by performing a case study on Nav2's main BT and test suite. We run Nav2's entire suite of system tests alongside CANOPY for 10 trials and collect data on the BT's overall coverage. Additionally we express per-node coverage and identify areas of the tree which are least tested.

The contributions of this work include:

- A *coverage metric* for BTs, called *BT status coverage*.
- The open-source BT-coverage-measuring tool: CANOPY.
- A case study of the BT-based Navigation2 (Nav2), finding that Nav2's BT status coverage is 56%, with three BT nodes completely uncovered.

II. RELATED WORK

Seshia *et al.* [6] explored test coverage in reactive systems based on *assume-guarantee* reasoning in a domain-specific programming language (DSL). Like their work we consider robotic systems to be a composition of interacting components, but unlike their work we do not require a DSL but do require a BT and instead seek to provide a tool and method with minimal impact to a robot developer's toolchain.

Alexander *et al.* [7] propose *situation coverage* for autonomous robots based on previous experience of situations or environments or circumstances that are known to cause problems, i.e. sensor degradation or adverse weather. Like situation coverage, we seek to direct testing toward edge cases, but unlike situation coverage we consider a BT as the basis for understanding what has been tested.

Our work is inspired by the venerable *Modified Condition Decision Coverage* (MCDC) criteria [10], wherein each possible way a true-or-false decision in software could be determined must be explored independently. Like MCDC, we seek a higher standard than just saying a BT component was executed, and propose coverage based on a BT component returning both 'success' and 'failure.' Unlike MCDC, we do not look inside BT nodes to see, for example, every reason why they could return 'failure.'

In an interview-based study of robot testing practices in real world development, Afzal *et al.* [5] found that outside of industrial automation, the robotics industry tends to value system safety and quality less than the value of being "first to market." For some robotics sub-domains however, a number of standards have been introduced. Additionally, in many safety critical applications outside of robotics, certification requires that systems achieve minimum test coverage scores as in the automotive [11], aviation [12], and railway [13] sectors.

[2]https://github.com/RobotCodeLab/BT-Canopy

A recent empirical study [3] on how BTs are used in practice found that BTs are a kind of 'model-at-runtime,' consistent with the burgeoning 'model-driven-engineering' (MDE) [8] paradigm. This empirical work includes an artifact identifying 75 BTs in 25 systems, and we examined all of them looking for whole executable systems with BTs. Unfortunately, only Nav2 included a runnable system "out of the box."

III. BACKGROUND

As robot systems grow more complex, developers are increasingly utilizing Model-Driven Engineering (MDE) as a means of abstracting out that complexity [14]. MDE is a methodology by which high-level models inform the system's design and execution. MDE is advantageous for robot systems as it provides a human-readable description of systems. Robotics is an inherently multidisciplinary field comprising programmers and non-programmers alike, which stands to benefit from Model-Driven Engineering [8].

A. Behavior Trees

In recent years, Behavior Trees (BTs) have gained attention in the robotics community as an alternative or complement to the venerable Finite State Machine (FSM) [15]. Figure 2 shows the BT included in the test suite for Nav2. BTs were originally devised as a control structure for the non-player characters (NPCs) in video games [2]. BTs are a form of hierarchical finite state machine that are represented via a directed acyclic graph [1]. BTs are as expressive as FSMs and aim to be more modular and extensible than FSMs for non-trivial systems. Additionally, the well-defined tree structure and top-down method of execution can make BTs more human readable [1].

As the name suggests, a BT is a kind of tree graph in which each node has exactly one parent (except the root, which has no parent). The purpose of a BT is to handle the execution of *actions* (also commonly referred to as *tasks*). BT actions may execute for some undetermined amount of time before they return. BT actions nodes are assumed to eventually either succeed or fail [1].

In BT implementations, the tree is "ticked" at a specified frequency. Starting at the root, ticks are propagated downwards through the tree's nodes. In most BT diagrams, the topmost *root* node is omitted as it is assumed. When a node is ticked, it must return its current status as defined as 'success', 'failure', or 'running'. Nodes which have at least one child are known as *interior* nodes whereas those without are *leaf* nodes. An interior node's status is dependent on the status of its children and when an interior node is ticked, it propagates that tick to its children depending on the node type's rules [1]. Conversely, the status of leaf nodes are determined by the BT's interactions with the larger software system.

In the most basic formulation, BTs are composed of Control Flow Nodes, Action Nodes, Conditional Nodes, and Decorator Nodes. Control Flow Nodes and Decorator Nodes represent the logic which orders the tree's execution. Action Nodes and Conditional Nodes are the leaves of a BT and tend to interact with the software system outside the tree.

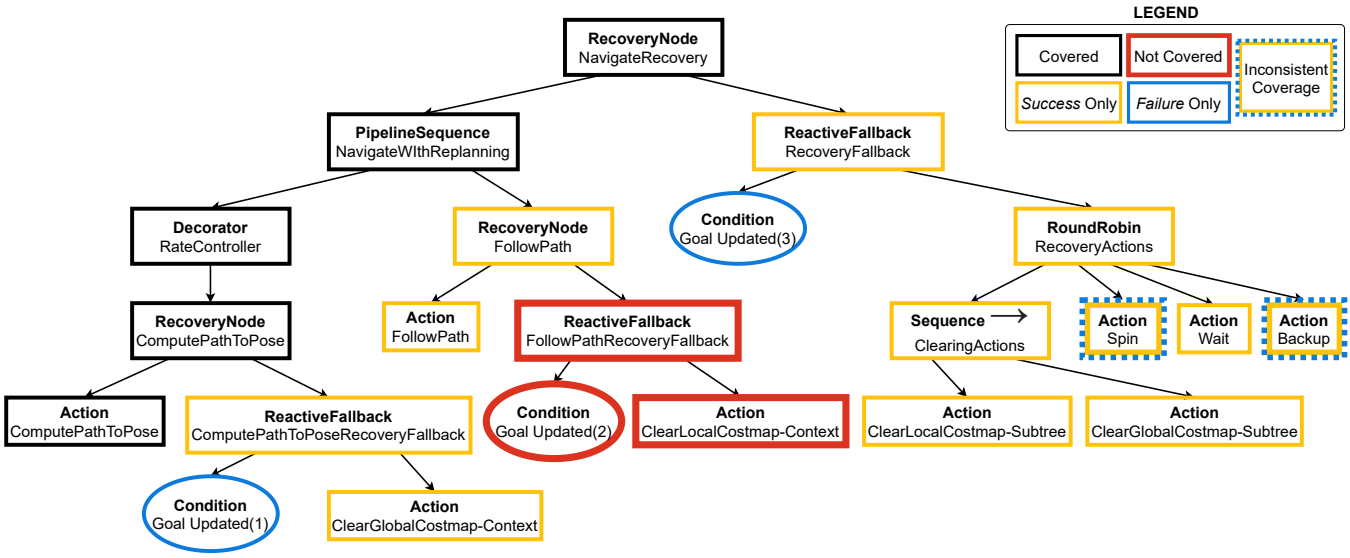

Fig. 2. Main Behavior Tree utilized in ROS2's *Navigation2* package, showing node BT status coverage.

Control Flow: In general, *Control Flow Nodes* handle the logic for ordering a tree's execution. Each node determines the execution order of its immediate children and is consequently always interior. The standard BT model includes *sequence* and *selector* nodes. Additionally, BT implementations may include useful custom Control Flow Nodes such as: parallel sequences[3], repeat `n`-times on failure sequences, or round-robin sequences[4].

Sequence: A *Sequence Node* is equivalent to a logical conjunction. In Figure 2, the 'Pipeline Sequence' is a kind of Sequence Node. If all children of a Sequence Node are successful, the sequence returns 'success,' but if any child fails, the sequence ends and returns 'failure.' If a child is currently running, the node returns 'running.'

Selector: In a BT diagram, a *Selector Node* (also commonly known as a "fallback") is equivalent to a logical disjunction. In Figure 2, the 'Reactive Fallback' nodes are kinds of Selector Nodes. If all children of a Selector Node fail, the selector returns 'failure,' but if any child succeeds, the selector ends and returns 'success.' If a child is currently running, the node returns 'running'.

Decorator: A *Decorator Node* has a single child and is responsible for modifying or blocking the state which is passed through it. In Figure 2, the 'RateController' on the left-hand-side blocks its child from being ticked more often than a specified frequency. The *inverter* is the most common form of decorator and is responsible for flipping 'success' to 'failure' and vice versa.

Condition: *Condition Nodes* are leaf nodes which return either 'success' or 'failure' depending on some global boolean variable. In Figure 2, the Condition Nodes are shown in ovals. This BT invokes the same underlying condition in three different parts of the BT.

Action: Also commonly referred to as task or execution nodes, *Action Nodes* are the primary mechanism by which a BT performs behaviors. In some BT implementations, Action Nodes link to a callback 'task' function (also called an action in ROS) which is executed when the node is ticked. While the task runs, the Action Node returns 'running' if ticked. Upon task completion, the task returns either 'success' or 'failure' to the Action Node which the Action Node subsequently returns upon being ticked.

B. Model Coverage

Testing is the primary method by which reliability[5] is estimated for in-development software [17]. Generally speaking, there are thought to be two primary approaches to testing and coverage: (1) *behavioral* in which only the system's end behavior is under examination, and (2) *structural* in which the underlying system's execution is under examination [18]. These terms are also commonly referred to as "black-box" and "white-box" testing techniques, respectively. In the language of Verification and Validation (V&V), behavioral testing is often associated with validation (are we building the right thing?) whereas structural testing is associated with verification (are we building the thing right?) [19] [20]. Because both test approaches tend to examine such different system characteristics, any well crafted test plan will likely include both behavioral and structural test techniques [21].

Behavioral testing techniques measure a system's inputs and outputs to estimate its capabilities in relation to its requirements [19]. For example, a test in which a robot

[3]Py Trees: https://py-trees.readthedocs.io/en/devel/composites.html

[4]Navigation 2: https://navigation.ros.org/behavior_trees/overview/nav2_specific_nodes.html

[5]Reliability is a measure of a system's ability to operate without error over periods of time. Previous work shows that robot reliability is a major influence on a user's trust in the system and that even temporary drops in reliability tend to leave a lasting impression in users [16].

is evaluated by an observer on its ability to move from one position to another would constitute a behavioral test. Existing approaches to robot test coverage such as *situation coverage* [7] (see §II) have tended to exclusively take a behavioral approach to testing.

Structural testing techniques measure characteristics about a system's underlying software architecture to evaluate the capabilities or reliability of its constituent parts [19]. Although structural testing is most commonly applied directly to source code, it may be performed on any level of abstraction such as a graph model of the system [17].

There are various notions of coverage to measure the adequacy of a system's test suite [9]. Coverage is commonly expressed as a percent value and is calculated according to the specifications of one or more coverage criteria [22]. A criterion may be something as simple as "the percent of lines in the source code executed" as is the case for structural *line coverage* [23]. Structural coverage techniques can differentiate insufficiently tested areas of the SUT which makes them useful for guiding developer test creation and for automated test generation [24] [25].

Most structural criteria measure test suite coverage on graphical models of the SUT [9]. If the system is represented graphically by a transition model such as an FSM, structural coverage will represent some measure of visited states, transitions, or paths [9]. However, if the system lacks an explicit model for execution, structural coverage may still be calculated using the program's control flow graph for which nodes represent basic blocks and directed edges represent jumps in the control flow [17]. Common structural criteria are defined below.

Node coverage: Also known as *statement coverage* or *block coverage* for control flow graphs and *all-states coverage* for FSMs, *node coverage* measures the percent of nodes which are executed at least once during testing [17], [22].

Edge coverage: Also known as *branch coverage* or *decision coverage* for control flow graphs and *all-transitions coverage* for FSMs, *edge coverage* measures the percent of edges between nodes which are visited during testing. If every edge in the graph is visited, it is implied that each node is also visited [26]. As such, edge coverage subsumes node coverage. Generally, a criterion is considered stricter and stronger than the criteria it subsumes [9].

Complete path coverage: This criterion requires that every distinct path through the graph be executed therefor subsumes other structural criteria [9]. If the graph has any form of cycle or loop (as control flow graphs commonly do), complete path coverage becomes infeasible due to the infinite possible paths through the graph.

IV. Technical Approach

Existing approaches to model coverage target a wide variety of graph models (of various levels of abstraction) including: FSMs [26], deterministic finite automatons (DFAs) [27], control flow graphs [9], UML diagrams [28], sequence diagrams [29], activity diagrams [30], and state charts [31]. To our knowledge, BTs have thus far been exempt from model-driven coverage research despite being graph models for system execution.

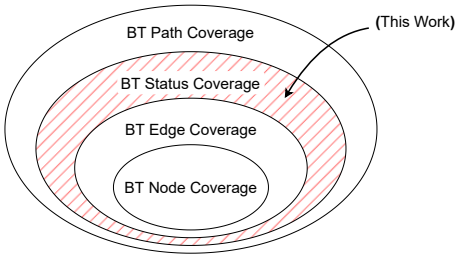

Fig. 3. Subsumption hierarchy of coverage criteria, showing that BT status coverage implies edge and node coverage.

To determine whether existing structural coverage techniques are sufficient (**RQ1**), we adapted two of the most common structural criteria: node and edge coverage. Additionally in this section, we introduce a new, BT-specific coverage criteria as well as an accompanying coverage evaluation tool. Our definitions for BT coverage are defined below.

1) BT Node Coverage: By our definition of BT node coverage, if a BT node is ticked, it is considered visited. Likewise, we define BT node coverage as the percent of nodes which are visited during testing. Recall that when ticked, a node may return a status of 'success', 'failure', or 'running'. Under our definition, a node which only returns 'running' is considered covered even if it does not finish.

2) BT Edge Coverage: Although BT transitions are visually represented by one-way edges, they are functionally a shorthand for two-way control transfers in which execution jumps to elsewhere but eventually returns to where the jump was made [1]. In modern programming languages, two-way control transfers are most commonly found in function calls [1]. Since a single BT transition represents both directions of the control transfer, in order for a BT edge to be fully covered, it must be traversed in both directions. By our definition, if a BT node is ticked and returns either 'success' *or* 'failure' during testing, the edge between it and its parent is considered visited. We define BT edge coverage as the percent of edges visited. Because BT edge coverage requires 'running' nodes to finish before being counted, it subsumes BT node coverage as described in Figure 3.

3) BT Status Coverage: Because each node is capable of returning 'success' or 'failure' during execution, we believe this necessitates a criteria which requires both values to be returned. BT status coverage requires that each BT node return both 'success' and 'failure' during testing and thus subsumes BT edge coverage as described in Figure 3. The 'running' status is not required for full status coverage since 'running' is not an inherent status type for every leaf node. For example, Condition Nodes and Action Nodes with short execution times may not ever return 'running' when ticked. If only 'success' or 'failure' are returned by a given node during execution, but not both, the node is only considered "half covered". The formula for status coverage is defined as:

$$\text{BT status coverage} = \frac{(n_s + n_f)}{2 * N} \tag{1}$$

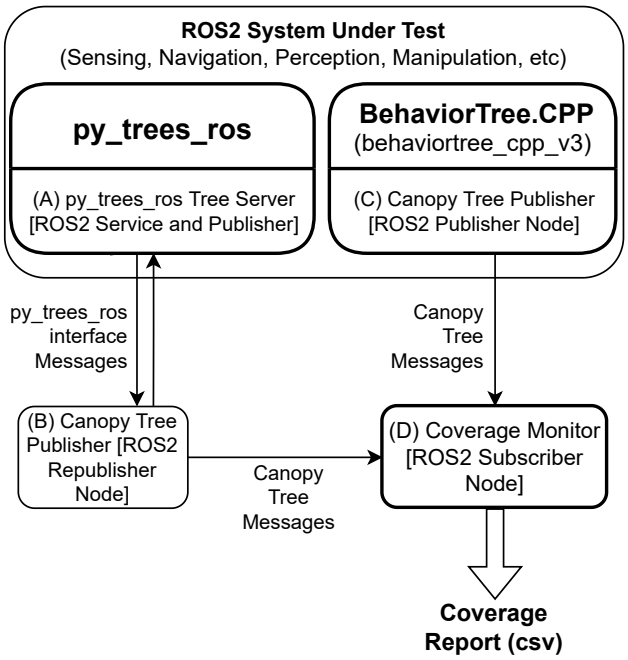

Fig. 4. High-level CANOPY architecture when monitoring a running ROS system under test. ROS2 nodes are labeled A, B, C, and D.

where n_s is the number of nodes which return 'success' at least once, n_f is the number of nodes which return 'failure' at least once, and N is the total number of nodes in the tree.

4) BT Complete Path Coverage: Theoretically, BT Complete path coverage would require every path through a sequence of BT states to be covered. As shown in Figure 3, if we could achieve BT complete path coverage, then BT status coverage would be implied, because BT complete path coverage subsumes BT status coverage. However, BT complete path coverage suffers from state-space explosion [32] and is, as yet, an unrealistic goal. Therefore, **we believe that BT status coverage is a pragmatic coverage metric choice** for robot developers seeking to measure and expand their test suites.

A. Implementation

Our tool, CANOPY, is a coverage evaluation tool for robot BTs which utilizes the *BT status coverage* criteria introduced in §IV-3. CANOPY targets ROS2 and has been tested with "Foxy Fitzroy" and "Galactic Geochelone" on Ubuntu 20.04. As a ROS2 application, CANOPY builds using the standard ROS2 `colcon` CMake system.

ROS2 and its predecessor ROS [33] utilize message passing nodes to facilitate inter-process communications via publisher-subscriber relationships. CANOPY primarily functions as a subscriber node (the *Coverage Monitor*) which logs activity on the robot system's BT as in Figure 4. For each node in the robot's BT, CANOPY stores a running total for the total number and types of status returns. These values may be useful to developers who wish to better understanding which nodes or sub-trees are being visited the most. In the case that a test suite includes multiple BTs (as with Nav2), CANOPY is capable of logging status data from multiple trees simultaneously.

Because CANOPY receives BT data over the ROS2 network, it is functionally agnostic to the BT library's implementation. In order for a BT library to support CANOPY, it needs to be able to publish its state change data to the ROS2 network via the included set of *tree messages*. For most imaginable implementations, this should be a fairly simple addition. In order to provide out-of-the-box functionality, CANOPY includes general purpose tree state publisher nodes for the two most popular BT implementations within the ROS community: `BehaviorTree.cpp` and `py_trees_ros` [3]. Figure 4 provides a high level view of how these tree state publisher nodes (nodes B and C) connect CANOPY's coverage monitor (node D) to the larger system.

1) `py_trees_ros` Integration Details: Out of the box, `py_trees_ros` includes a ROS2 tree state publisher node (A in Figure 4). This "Tree Server" is capable of publishing the tree state information which is required for CANOPY's tree messages, but must be switched on for each tree via a ROS2 service request. To facilitate sending theise requests and to handle the subsequent `py_trees_ros` tree data, we have created the ROS2 *re-publisher* node (B in Figure 4). To summarize, the re-publisher node does the following: (1) periodically scans the ROS2 network for signs of any new `py_trees_ros` trees, (2) if a new tree is found, sends a request to enable its tree state publisher, and (3) for any incoming `py_trees_ros` tree state messages, formats the data to CANOPY's tree messages and re-publishes the data to be received by the monitor.

2) `BehaviorTree.cpp` Integration Details: Out of the box, `BehaviorTree.cpp` does not integrate with the ROS2 network[6]. Nevertheless, `BehaviorTree.cpp` can be made to integrate fairly easily and is popular among robot developers for a variety of reasons [3]. To publish `BehaviorTree.cpp`'s state information to the ROS2 network, we have created a ROS2 publisher node (C in Figure 4) which extends the existing `StatusChangeLogger` class. The `StatusChangeLogger` class requires a pointer to a tree object at initialization and includes a pure virtual (abstract) `callback` function in which we have implemented the ROS2 publishing. Because our publisher requires a tree pointer, it must be imported into the existing code base (unlike the `py_trees_ros` re-publisher which can be launched as an independent ROS2 node). In the next section, we discuss a working example of the `BehaviorTree.cpp` ROS2 publisher node in an existing project.

V. CASE STUDY OF CANOPY WITH NAV2

Navigation2 (Nav2) is a popular open-source navigation system for ROS2 robots [4], which on GitHub has over 1200 stars and 728 forks. Nav2 primarly utilizes configurable BTs for orchestrating the system's "planning, control and recovery tasks" [4]. Since its initial publishing, the system's

[6]BehaviorTree.ROS (https://github.com/BehaviorTree/BehaviorTree.ROS) is an in-development `BehaviorTree.cpp` wrapper with built-in ROS2 support.

TABLE I
BT UTILIZATION REPORT FOR *Navigation2*'S TEST SUITE AVERAGED OVER 10 RUNS.

			BEHAVIOR TREE STATES (COUNTS)				
NODE NAME	TYPE	# VISITS	'Failure'	'Success'	'Running'	COVERED?	STATUS COVERAGE %
NavigateRecovery	RecoveryNode	24.5	2.2	9.6	12.7	✓	100
NavigateWithReplanning	PipelineSequence	50.9	15.4	9.6	25.9	✓	100
RateController	Decorator	217	15.4	87.9	113.7	✓	100
ComputePathToPose	RecoveryNode	206.7	15.4	87.9	103.4	✓	100
ComputePathToPose	Action	238.1	31.1	87.9	119.1	✓	100
ComputePathToPoseRecoveryFallback	ReactiveFallback	15.7	0.0	15.7	0.0	✓	50
GoalUpdated(1)	Condition	15.7	15.7	0.0	0.0	✓	50
ClearGlobalCostmap-Context	Action	15.7	0.0	15.7	0.0	✓	50
FollowPath	RecoveryNode	20.4	0.0	9.6	10.8	✓	50
FollowPath	Action	20.4	0.0	9.6	10.8	✓	50
FollowPathRecoveryFallback	ReactiveFallback	0	0.0	0.0	0.0	✗	0
GoalUpdated(2)	Condition	0	0.0	0.0	0.0	✗	0
ClearLocalCostmap-Context	Action	0	0.0	0.0	0.0	✗	0
RecoveryFallback	ReactiveFallback	22.5	0.0	13.2	9.3	✓	50
GoalUpdated(3)	Condition	13.2	13.2	0.0	0.0	✓	50
RecoveryActions	RoundRobin	26.4	0.0	13.2	13.2	✓	50
ClearingActions	Sequence	8.8	0.0	4.4	4.4	✓	50
ClearLocalCostmap-Subtree	Action	4.4	0.0	4.4	0.0	✓	50
ClearGlobalCostmap-Subtree	Action	4.4	0.0	4.4	0.0	✓	50
Spin	Action	8.6	0.2	4.1	4.3	✓	60
Wait	Action	5.8	0.0	2.9	2.9	✓	50
BackUp	Action	4.6	0.5	1.8	2.3	✓	65
			TOTAL BEHAVIOR TREE STATUS COVERAGE				56%

main BT, `navigate_to_pose_w_replanning_and_recovery.xml` (see Figure 2), has grown from 9 nodes to 22 nodes over the course of 30 git commits. Nav2 implements its BT by way of the `BehaviorTree.cpp` package. Nav2 extends `BehaviorTree.cpp` in order to define custom nodes in addition to the standard types discussed in §III-A. For more information on Nav2's custom node types, see the documentation at [34].

In addition to subsystem specific unit tests, Nav2 also includes a large number of simulation-based system tests. These tests utilize a differential drive Turtlebot3 robot across multiple scenarios in the open-source `Gazebo` simulator [35]. According to the main Nav2 repository, the suite tests for correct robot navigation, successful system life cycle across multiple launches, proper handling and recovery for system failures, among other things. "*[The tests] are primarily for use in Nav2 CI[continuous integration] to establish a high degree of maintainer confidence when merging in large architectural changes to the Nav2 project.*"

A. Applying Canopy to Nav2

As a working example of CANOPY, we evaluated Nav2's test suite for its v.1.0.12 Galactic release. To integrate CANOPY with Nav2, we import and initialize our general purpose `BehaviorTree.cpp` tree state publisher node (see §IV-A) inside the `bt_action_server.hpp` file where Nav2 initializes its BT. While there, we also disable the redundant `RosTopicLogger` as it is currently unused by Nav2. Lastly, we add our publisher node `behaviortree_cpp_v3_ros2_publisher` as a dependency to `CMakeLists.txt` and `package.xml`.

For our evaluation of Nav2's test coverage of its main[7] BT, we launch CANOPY's logger node alongside the test suite included in the `nav2_system_tests` folder. We performed this process for 10 trials to obtain the averaged values contained in Table I. For each trial, we utilized a fresh `docker` container running on an Intel(R) Core(TM) i5-6600K CPU @ 3.50GHz processor with access to 32 gigabytes of RAM. The mean run-time for each trial was just over 20 minutes with a standard deviation of 19 seconds. We chose to run `Gazebo`'s graphical interface `gzclient` during testing for visual test observation. From our experimentation, we could not find any significant differences in run-time or test behavior from running `Gazebo` headless versus with its GUI.

[7]Nav2's suite tests two trees: the primary tree and a secondary tree. For this study, we exclude the secondary tree because it only has a single test.

B. Experimental Results

To answer **RQ2** we averaged our results from the 10 trials, and found that Nav2's test suite earns a BT status coverage score of 56% with a standard deviation of 1.2%. We also collected data on individual nodes by using CANOPY's node logging functionality. Table I shows per-node coverage values as well the total number of returns for each of the three status types. From the individual node data, we have determined the following characteristics about Nav2's BT and test suite:

1) Three nodes (such as ClearLocalCostmap-Context) are never covered by the suite, meaning that these components are never invoked. These uncovered nodes are colored red in Figure 2. Adding tests directed at this area would increase BT status coverage the most and make sure the tests at least visit every component.
2) Some nodes (such as FollowPath) are always partially covered: they return 'failure' or 'success' one or more times during testing, but not both. For these nodes, the untested status type is consistent between trials. Graphically, we represent these nodes with blue and yellow in Figure 2. Partially covered nodes would require tests that cause the node to return the uncovered status.

TABLE II
BT COVERAGE SCORE FOR *Navigation2*'S TEST SUITE AVERAGED OVER 10 RUNS.

BT CRITERION	AVERAGE COVERAGE	STANDARD DEV.
Status Coverage	56%	1.2%
Node Coverage	86%	0.0%
Edge Coverage	86%	0.0%

3) Some nodes are inconsistent and are only sometimes fully covered by the test suite. Specifically, the Spin and Backup nodes always succeed during testing but only failed in 2 and 3 of the trials (respectively). This uncertainty between trials is likely caused by randomly-generated waypoint poses in one of the test scenarios.
4) Five nodes (such as ComputePathToPose) are always fully covered by the test suite, and these are some of the most frequently visited nodes, meaning that these components should be prioritized during testing and carefully modified during system evolution.

C. Discussion

For each trial, we computed BT node and BT edge coverage scores represented as averages in Table II. Although BT edge coverage is technically more strict than BT node, we see that both criteria score the same 86% coverage score without any deviation between trials. Recall from §IV that BT edge coverage is more strict only in that it requires each node to finish running. In the case of Nav2's test suite, each visited node was able to finish running during each trial– hence the identical scores.

In comparison with BT edge coverage, Nav2 earns a significantly lower BT status coverage score due to its BT containing a large portion of partially tested nodes. Of 22 total BT nodes, 12 exclusively return 'success' and 2 exclusively return 'failure' as seen in Table I. In our opinion, the large skew towards 'success' status types for partially covered nodes suggest that Nav2's test suite lacks sufficient negative test cases: a leading cause of missed defects in software testing [36]. Like MCDC, Status coverage is advantageous because it requires both positive and negative test cases. Additionally, of the three metrics, only status coverage was able to capture differences between individual trials as indicated by its non-zero standard deviation score. Although BT node and BT edge coverage are able to identify Nav2's three untested nodes, we believe that only Status Coverage adequately addresses BT coverage as considered in **RQ1**.

Robot system developers can use CANOPY's node results to design additional tests which will increase coverage the most. For example, we know from Figure 2 that the *FollowPath* Action Node never returns 'failure', and as a result, a group of 3 nodes are never executed at all. If however, a test were created in which the *FollowPath* Action failed, the *FollowPath* RecoveryNode would subsequently execute its next child and at least 2 previously untested nodes would be partially covered (as well as the now fully tested *FollowPath* Action).

VI. CONCLUSION

In this work, we explore notions of 'coverage' for robotic systems that use Behavior Trees and propose a coverage metric, BT status coverage, that can be used to measure how much of a Behavior Tree has been tested or utilized. We describe the implementation of this coverage metric in the open-source tool CANOPY, and present a case study where we apply CANOPY to the popular ROS2 package *Navigation2*, finding that the tests provided by Nav2 cover 56% of possible BT statuses. We also describe how CANOPY can help robot system developers better understand how their BT is utilized at runtime.

VII. FUTURE WORK: PATH COVERAGE

In addition to node and edge coverage, there exists a plethora of structural criteria which quantify coverage in terms of graph paths in order to catch defects that only occur under specific sequences of system execution (in our case, nodes) [17]. These so called "path-based" criteria all subsume edge coverage and include (but are not limited to): edge-pair [9], prime-path [26], and complete path [9] coverage. From our initial investigation, we have identified 3 major design challenges that we believe need to be addressed when developing practical path-based coverage criteria for BTs.

1) Paths With Node Status: Based on the results from this study, we believe the first challenge in defining BT path coverage is deciding how to incorporate node status. Under a simple BT path definition in which a node's status is not considered, not all path-based criteria subsume BT status coverage. For example, consider edge-pair coverage in which "every possible path of length 2" must be executed. Although this definition is stricter than BT edge coverage, there are scenarios for which edge-pair neglects to cover both statuses for each node. Specifically, for any Sequence Node or Selector Node with n children, if the first $n-1$ children return both status types, the last child only needs to return a single status type to enable full edge-pair coverage. Consequently, BT edge-pair coverage is weaker than BT status coverage because it lacks status awareness.

2) Paths With Custom Nodes: In order to get a path coverage score in the form of a percentage, there must be: 1) some way of measuring the order in which nodes are traversed, and 2) a mechanism for predicting the set of valid paths which would result in full coverage (according to the select criterion). Since we already have CANOPY, we will assume that measuring a BT's node traversal order is just an implementation detail. Generating valid paths is a fairly straightforward process under the standard formulation of control flow nodes described in §III-A, since Sequence, Selector, and Inverter nodes are simple and universally defined. The real challenge is to devise a way for generating valid paths in BTs with custom nodes (such as in Nav2's main tree in Figure 2). Custom nodes are supported in both `py_trees_ros` and `BehaviorTree.cpp` and allow BT designers to introduce new rules for how BTs can be traversed. For example, both Nav2's *RecoveryNode* and `BehaviorTree.cpp`'s *RetryNode* can be used to loop

within a BT by repeatedly ticking child behaviors until they return 'success' or until reaching a maximum number of tries. Additionally, Nav2 introduces the *RoundRobin* node which loops over its children in an order partially determined by an internal variable that persists between calls to the BT. As evident by the node descriptions above, developers can create BTs with custom nodes which are capable of producing valid paths more complex and dissimilar than anything possible under the standard formulation alone. Given that custom nodes generally lack explicit descriptions for how they modify paths on the tree, we believe that the most efficient use of resources is to manually define said path descriptions for a limited subset of the most commonly used custom nodes.

3) Concurrency: The third challenge is how to handle BTs with concurrent sub-trees. Many BT libraries including `py_trees_ros` and `BehaviorTree.cpp` provide *parallel* control flow nodes for concurrent child execution. In a concurrent BT, combinations of multiple nodes are able to execute simultaneously which may cause the tree's state-space size to explode exponentially [37]. As a further complication, concurrent programs are usually inherently non-deterministic as a result of thread scheduling [38]. Generally speaking, previous structural approaches to concurrent coverage have attempted to avoid state-space explosion by defining subsets of paths where bugs are thought to be most likely to occur such as in All-Definition Use-Path coverage [39] [40] [17]. Determining a useful subset of BT paths will likely be one of the largest challenges in designing path coverage criteria for concurrent BTs.

Given the challenges described above, we feel that designing path-based coverage criteria for BTs is a complex problem which warrants a deeper investigation beyond the scope of this paper.

REFERENCES

[1] M. Colledanchise and P. Ögren, "Behavior trees in robotics and AI: an introduction," *CoRR*, vol. abs/1709.00084, 2017. [Online]. Available: http://arxiv.org/abs/1709.00084

[2] D. Isla, "Handling complexity in the halo 2 ai," 2006. [Online]. Available: https://www.gamasutra.com/view/feature/130663/

[3] R. Ghzouli, T. Berger, E. B. Johnsen, S. Dragule, and A. Wasowski, "Behavior trees in action: A study of robotics applications," in *Proceedings of the 13th ACM SIGPLAN International Conference on Software Language Engineering*, ser. SLE 2020. New York, NY, USA: Association for Computing Machinery, 2020, p. 196–209. [Online]. Available: https://doi.org/10.1145/3426425.3426942

[4] S. Macenski, F. Martín, R. White, and J. G. Clavero, "The marathon 2: A navigation system," in *IEEE/RSJ International Conference on Intelligent Robots and Systems, IROS 2020, Las Vegas, NV, USA, October 24, 2020 - January 24, 2021*. IEEE, 2020, pp. 2718–2725. [Online]. Available: https://doi.org/10.1109/IROS45743.2020.9341207

[5] A. Afzal, C. L. Goues, M. Hilton, and C. S. Timperley, "A study on challenges of testing robotic systems," in *13th IEEE International Conference on Software Testing, Validation and Verification, ICST 2020, Porto, Portugal, October 24-28, 2020*. IEEE, 2020, pp. 96–107. [Online]. Available: https://doi.org/10.1109/ICST46399.2020.00020

[6] A. Desai, A. Phanishayee, S. Qadeer, and S. A. Seshia, "Compositional programming and testing of dynamic distributed systems," *Proceedings of the ACM on Programming Languages*, vol. 2, no. OOPSLA, pp. 1–30, 2018.

[7] R. Alexander, H. R. Hawkins, and A. J. Rae, "Situation coverage–a coverage criterion for testing autonomous robots," 2015.

[8] G. L. Casalaro, G. Cattivera, F. Ciccozzi, I. Malavolta, A. Wortmann, and P. Pelliccione, "Model-driven engineering for mobile robotic systems: a systematic mapping study," *Software and Systems Modeling*, vol. 21, no. 1, pp. 19–49, aug 2021. [Online]. Available: https://doi.org/10.1007/s10270-021-00908-8

[9] M. Utting and B. Legeard, *Practical Model-Based Testing: A Tools Approach*, 1st ed. Morgan Kaufmann, 2007.

[10] P. Ammann, J. Offutt, and H. Huang, "Coverage criteria for logical expressions," in *14th International Symposium on Software Reliability Engineering, 2003. ISSRE 2003.*, 2003, pp. 99–107.

[11] ISO, "Road vehicles – Functional safety," 2011.

[12] B. Brosgol, "Do-178c: The next avionics safety standard," *Ada Lett.*, vol. 31, no. 3, p. 5–6, nov 2011. [Online]. Available: https://doi.org/10.1145/2070336.2070341

[13] J. Boulanger, *CENELEC 50128 and IEC 62279 Standards*, ser. Control, systems and industrial engineering series. Wiley, 2015.

[14] C. Schlegel, A. Steck, D. Brugali, and A. C. Knoll, "Design abstraction and processes in robotics: From code-driven to model-driven engineering," in *Simulation, Modeling, and Programming for Autonomous Robots - Second International Conference, SIMPAR 2010, Darmstadt, Germany, November 15-18, 2010. Proceedings*, ser. Lecture Notes in Computer Science, N. Ando, S. Balakirsky, T. Hemker, M. Reggiani, and O. von Stryk, Eds., vol. 6472. Springer, 2010, pp. 324–335. [Online]. Available: https://doi.org/10.1007/978-3-642-17319-6_31

[15] B. Siciliano, O. Khatib, and T. Kröger, *Springer handbook of robotics*. Springer, 2008, vol. 200.

[16] J. L. Wright, J. Y. C. Chen, and S. G. Lakhmani, "Agent transparency and reliability in human–robot interaction: The influence on user confidence and perceived reliability," *IEEE Transactions on Human-Machine Systems*, vol. 50, no. 3, pp. 254–263, 2020.

[17] P. Ammann and J. Offutt, *Introduction to Software Testing*. Cambridge: Cambridge University Press, 2016.

[18] S. Nidhra and J. Dondeti, "Black box and white box testing techniques-a literature review," *International Journal of Embedded Systems and Applications (IJESA)*, vol. 2, no. 2, pp. 29–50, 2012.

[19] L. Williams, "A (partial) introduction to software engineering practices and methods," *NCSU CSC326 Course Pack*, vol. 2009, no. 5, pp. 33–63, 2008.

[20] B. W. Boehm, "Verifying and validating software requirements and design specifications," *IEEE Software*, vol. 1, no. 1, pp. 75–88, Jan 1984, copyright - Copyright IEEE Computer Society Jan/Feb 1984; Last updated - 2021-09-09; CODEN - IESOEG.

[21] J. Pan, "Software testing," *Dependable Embedded Systems*, vol. 5, p. 2006, 1999.

[22] G. ISTQB, "Istqb/gtb standard glossary for testing terms," v2. 2. Tech. rep, Tech. Rep., 2013.

[23] M. Ivankovic, G. Petrovic, R. Just, and G. Fraser, "Code coverage at google," in *Proceedings of the 2019 27th ACM Joint Meeting on European Software Engineering Conference and Symposium on the Foundations of Software Engineering*, 2019, pp. 955–963.

[24] C. Jard and T. Jéron, "Tgv: theory, principles and algorithms," *International Journal on Software Tools for Technology Transfer*, vol. 7, no. 4, pp. 297–315, 2005.

[25] C. Nebut, F. Fleurey, Y. Le Traon, and J.-M. Jezequel, "Automatic test generation: A use case driven approach," *IEEE Transactions on Software Engineering*, vol. 32, no. 3, pp. 140–155, 2006.

[26] V. Rechtberger, M. Bures, and B. S. Ahmed, "Overview of test coverage criteria for test case generation from finite state machines modelled as directed graphs," 2022. [Online]. Available: https://arxiv.org/abs/2203.09604

[27] P. Wang and K. Stolee, "How well are regular expressions tested in the wild?" 10 2018, pp. 668–678.

[28] R. D. Ferreira, J. P. Faria, and A. C. Paiva, "Test coverage analysis of uml state machines," in *2010 Third International Conference on Software Testing, Verification, and Validation Workshops*, 2010, pp. 284–289.

[29] A. Rountev, S. Kagan, and J. Sawin, "Coverage criteria for testing of object interactions in sequence diagrams," in *International Conference on Fundamental Approaches to Software Engineering*. Springer, 2005, pp. 289–304.

[30] M. Shirole and R. Kumar, "Concurrency coverage criteria for activity diagrams," *IET Software*, vol. 15, no. 1, pp. 43–54, 2021. [Online]. Available: https://ietresearch.onlinelibrary.wiley.com/doi/abs/10.1049/sfw2.12009

[31] Y. D. Salman, N. L. Hashim, M. M. Rejab, R. Romli, and H. Mohd, "Coverage criteria for test case generation using uml state chart diagram," in *AIP Conference Proceedings*, vol. 1891, no. 1. AIP Publishing LLC, 2017, p. 020125.

[32] S. Bardin and P. Herrmann, "Pruning the search space in path-based test generation," in *2009 International Conference on Software Testing Verification and Validation*, 2009, pp. 240–249.

[33] M. Quigley, K. Conley, B. Gerkey, J. Faust, T. Foote, J. Leibs, R. Wheeler, A. Y. Ng *et al.*, "Ros: an open-source robot operating system," in *ICRA workshop on open source software*, no. 3.2. IEEE, 2009, p. 5.

[34] S. Macenski, "Nav2 — navigation 2 1.0.0 documentation," 2019. [Online]. Available: https://navigation.ros.org/index.html

[35] N. Koenig and A. Howard, "Design and use paradigms for gazebo, an open-source multi-robot simulator," in *2004 IEEE/RSJ International Conference on Intelligent Robots and Systems (IROS) (IEEE Cat. No.04CH37566)*, vol. 3, 2004, pp. 2149–2154 vol.3.

[36] Y. Chernak, "Validating and improving test-case effectiveness," *IEEE Software*, vol. 18, no. 1, pp. 81–86, 2001.

[37] A. Valmari, "A stubborn attack on state explosion," in *International Conference on Computer Aided Verification*. Springer, 1990, pp. 156–165.

[38] V. Terragni and S.-C. Cheung, "Coverage-driven test code generation for concurrent classes," in *2016 IEEE/ACM 38th International Conference on Software Engineering (ICSE)*, 2016, pp. 1121–1132.

[39] V. Arora, R. Bhatia, and M. Singh, "A systematic review of approaches for testing concurrent programs," *Concurrency and Computation: Practice and Experience*, vol. 28, no. 5, pp. 1572–1611, 2016.

[40] C.-S. D. Yang, A. L. Souter, and L. L. Pollock, "All-du-path coverage for parallel programs," in *Proceedings of the 1998 ACM SIGSOFT International Symposium on Software Testing and Analysis*, ser. ISSTA '98. New York, NY, USA: Association for Computing Machinery, 1998, p. 153–162.