Canopy: Quantifying Behavior Tree Coverage

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/

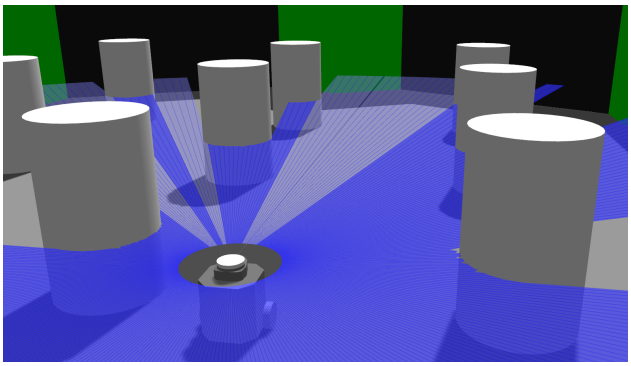

Fig. 1. Turtlebot3 with lidar simulated in Gazebo running the Behavior Tree from Nav2's test suite.

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. We start by examining existing node and edge-based 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.

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

CANOPY is a straightforward logging and aggregation tool, built on well-researched ideas from model-based testing.

The contributions of this work include:

- A *coverage metric* for BTs, called *BT Status Coverage*.
- An open-source BT-coverage-measuring tool CANOPY.
- A case study of the BT-based Navigation2 (Nav2), finding that Nav2 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.'

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 [11]. 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) [12]. 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.

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,

[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

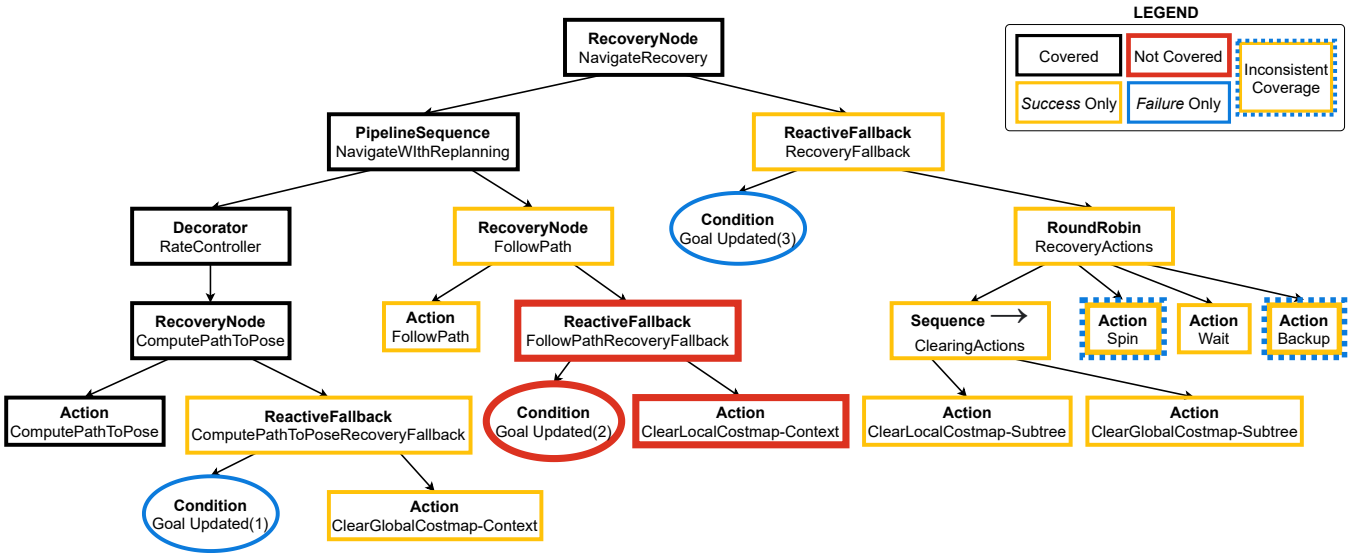

Fig. 2. Main Behavior Tree utilized in ROS2's *Navigation2* package, showing node BT Status Coverage.

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

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 [13]. A criterion may be something as simple as "*The percent of lines in the source code executed*" as is the case for *line coverage* [14].

Structural criteria measure test suite coverage on graphical models of the System Under Test [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 [15]. Common structural criteria are defined below.

Node coverage: measures the percenta of nodes which are executed at least once during testing [13], [15].

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 [16]. As such, edge coverage subsumes node coverage. Generally, a criterion is considered stronger than the criteria it subsumes [9].

IV. Technical Approach

There exists an abundance of model-driven coverage research in the software engineering domain [17] [18]. In this section, we propose that existing structural coverage techniques may be adapted for use in robot BTs. For this study, we have chosen to adapt the two most basic techniques: node and edge coverage. Additionally in this section, we introduce a new, BT-specific coverage criteria as well as an accompanying coverage evaluation tool.

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 somewhere else 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 as such subsumes BT edge coverage as described in Figure 3. 'Running' is not required for full status coverage since the 'running' status 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:

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

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 Path Coverage: Theoretically, BT 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 path coverage, then BT Status Coverage would be implied, because BT path coverage subsumes BT Status Coverage. However, BT path coverage suffers from state-space explosion [19] and is, as yet, an unrealistic goal. Therefore, **we believe that BT Status Coverage is a pragmatic coverage metric choice** for most robotic systems and can be used to help guide robot test developers to 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

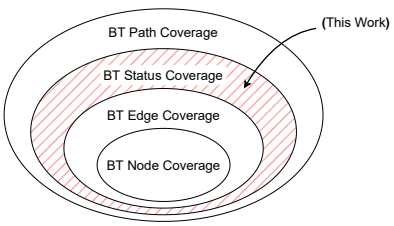

Fig. 3. Subsumption hierarchy of coverage criteria, showing that BT Status Coverage implies edge and node coverage.

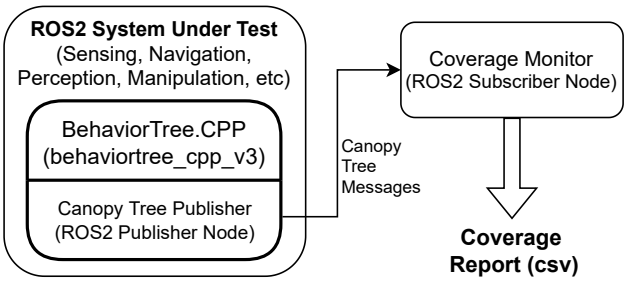

Fig. 4. High-level CANOPY architecture when monitoring a running ROS system under test.

"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 [20] utilize message passing nodes to facilitate inter-process communications via publisher-subscriber relationships. CANOPY functions as a subscriber node 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. 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].

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 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 [21].

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 [22]. According to the main Nav2 repository, the

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%

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[5] 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.

[5]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. Results

Upon averaging our results from the 10 trials, we calculate a BT Status Coverage score of 56% with a standard deviation of 1.2%. By using CANOPY's node logging functionality, we also collected data on individual nodes. 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. 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.
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.
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.

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. In the future, CANOPY could be extended to track paths through a sequence of BT states, which would be used to detect anomalous and potentially undesired system behavior.

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] 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

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

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

[14] 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.

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

[16] 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

[17] R. France and B. Rumpe, "Model-driven development of complex software: A research roadmap," in *Future of Software Engineering (FOSE '07)*, 2007, pp. 37–54.

[18] 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.

[19] 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.

[20] 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.

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

[22] 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.