Structural Coverability for Intelligent Automation Systems

Structural coverability for intelligent automation systems

Endre Erős[1], Kristofer Bengtsson[2], and Knut Åkesson[1]

Abstract— **In order to be flexible and handle complex scenarios, intelligent automation systems might benefit from automated planning techniques which rely on specifications and models describing their behavior. However, due to the presence of message passing, latency, jitter, timeouts, failures, and error handling, the verification of such behavior models using formal methods is often unfeasible. Therefore, testing has emerged as an approach to evaluating the behavior of intelligent automation systems. This paper presents a way to analyze structural coverability of behavior models for intelligent automation systems, which is inspired by the modified condition/decision coverage (MC/DC) criterion. This is paired with a testing procedure that enables each test case to influence both the controller and the simulated environment by injecting some specific state. As a result, the proposed coverability criterion can effectively identify segments of the behavior model that have not been adequately tested and suggest additional test cases to improve coverability. An example use case is presented to demonstrate the effectiveness of this approach.**

I. INTRODUCTION

The process of testing is a crucial element in the development of control software and plays a key role in ensuring the reliability, functionality, and performance of automation systems [1], [2]. Although formal verification can benefit subsystems in automated systems, it is generally impractical to use model checking [3] to verify a complete implementation due to various challenges such as lack of formal models. Therefore, it is essential to implement and utilize a comprehensive testing strategy that can identify software defects before deploying the system [4]. Despite its significance in ensuring system reliability, determining the appropriate level of required testing is a common challenge.

To address this challenge, different coverage criteria can help assess to which extent the code is exercised [5]. One coverage criterion that is widely used in the automotive and aerospace industries, is the Modified Condition/Decision Coverage (MC/DC) [6]. Naturally, MC/DC has also seen adoption in industrial automation systems [7].

Such criteria exist to ensure that the testing process provides sufficient coverage of the System Under Test (SUT) [8], otherwise the provided test has failed to exercise parts of the SUT. High coverage can help minimize the risk of unexpected software failures and ensure that the system performs as expected. Therefore, coverage metrics might be beneficial to assess the adequacy of the testing process and determine whether more tests are necessary [5]. This work focuses on increasing the coverability of the SUT using an MC/DC inspired approach.

In this work, the SUT is identified as the behavior model, which is used by the controller's planning and acting algorithms to control a set of resources. A *resource* refers to any physical or virtual component that is used to carry out a specific task or function. This can include hardware components such as sensors, actuators, and controllers, as well as software components such as algorithms, programs, and databases. To connect and control such resources, we use the Robot Operating System 2 (ROS2) [9].

The intelligent automation system framework presented in this work is based on our previous work from [10], and it is implemented in Rust [11] using the async Rust bindings for ROS2 [12]. In this framework, the control algorithms, resources, and corresponding drivers are distributed across a set of computational *nodes*. In order to achieve control over the system, these nodes must communicate with each other through a network, utilizing message-passing protocols.

The importance of testing and verification of ROS-based systems is addressed in [13], where the authors introduce a framework for analysis and quality improvement for software implemented with ROS. For example, [14] investigates property-based testing of various configurations of a ROS system, while [15] proposes a technique to automatically verify system-wide safety properties of ROS-based applications.

During the development phase of such ROS-based intelligent automation systems, resources are usually simulated to allow developers to test and verify the implemented behavior model before deploying it in a real-world environment In this work, we utilize such resource simulations which enable us to quickly test and iterate the behavior model.

The main contribution of this paper is the introduction of Structural Coverability of Behavior Models for Intelligent Automation Systems (SCBM-IAS), which is inspired by MC/DC criterion. We define a modeling abstraction called an *operation*, and define the SCBM-IAS criterion based on the coverability of its states during execution.

Additionally, we present a testing procedure inspired by mutation testing [16], which enables the tester to inject state to the controller and simulated resources during testing.

Finally, we use the SCBM-IAS criterion during the testing procedure to help us define additional test cases and catch modeling errors. This is exemplified with a use-case consisting of a robot manipulator, a structured light scanner, and a gripper.

[1]Endre Erős and Knut Åkesson are with Chalmers University of Technology, Electrical Engineering, Systems and Control Department, Automation Research Group, Gothenburg, Sweden. endree@chalmers.se, knut@chalmers.se

[2]Kristofer Bengtsson is with Volvo Group as Senior Researcher for Smart and Connected Operations, Research and Technology Development, Gothenburg, Sweden. kristofer.bengtsson@volvo.com

*This work has been supported by AIHURO (Vinnova) and the Swedish Research Council (VR) project SyTeC 2016-06204.

II. PRELIMINARIES

In this work we represent the behavior of intelligent automation systems with *variables*, *states*, *transitions*, and *operations*. With such a model in place, a *planning algorithm* can be employed to determine *plans*, considering defined objectives and the current state of the system. Such plans are then executed by a *runner* which communicates with the *resources* using *messages*. The descriptions below outline the components of modeling and execution:

- Variable: A *variable* v is a named unit of data that can be assigned a value x from a finite domain V.

- State: A *state* S is a set of tuples $S = \{\langle v_i, x_i \rangle\}$, where v_i is a variable with domain V_i and $x_i \in V_i$ is a value.

- Predicate: A *predicate* is an *equality logic formula* F that evaluates to either *true* or *false*.

- Equality logic formula: An *equality logic formula* F is defined with the following grammar:

$$\begin{aligned} &F : F \wedge F \mid F \vee F \mid \neg F \mid atom \\ &atom : term == term \mid true \mid false \\ &term : variable \mid value \end{aligned}$$

- Planning transition: A *planning transition* t contains a guard predicate $g : S \rightarrow \{false, true\}$, and a set of action functions A, where $\forall a \in A, a : S \rightarrow S$ models the updates of the state variables. If the guard predicate evaluates to *true*, the transition can occur, after which the actions of the transition describe how the variables are updated. The notation we use to represent a planning transition is $t : g/A$.

- Running transition: A *running transition* t_r extends the planning transition with an additional *running guard* g_r and additional *running actions* A_r. We write running transitions as $t_r : g/g_r/A/A_r$, where g and g_r are both guard predicates $g \wedge g_r : S \rightarrow \{false, true\}$, and A and A_r are both sets of action functions, where $\forall a \in A \cup A_r, a : S \rightarrow S$ models the updates of the values of the state variables.

While planning, only g and A are considered, i.e. the running transition is evaluated and taken as a planning transition. When the execution engine is running the plan, it is considering all components of t_r, i.e. the running transition guard becomes $g \wedge g_r$ and the set of transition actions becomes $A \cup A_r$.

- Operation: An *operation* O captures the behavior of tasks that can take some time to complete, and it is a convenient modeling abstraction for both planning and plan execution. A model of an operation can be in its *initial* (init) or *executing* (exec) state, see Fig. 1.

The *precondition* is a running transition associated with the *start* of the operation, switching it to the *executing* state. The operation will be in its executing state until the guard of the *postcondition* running transition is satisfied. The satisfaction of the postcondition implies that the operation is completed and can return to the *initial* state.

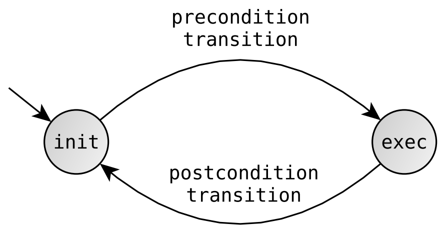

Fig. 1. A model of an operation.

- Automatic transition: Automatic transitions are running transitions that can be taken at any time assuming that the corresponding guard predicate is satisfied. These can be used to trigger some specific behavior, for example triggering safety mechanisms, resets, timeout behavior, etc.

- Behavior model: A *behavior model* M is a collection of variables, operations, and automatic transitions that model the behavior of that system.

- Planning problem: A *planning problem* Ψ is a 4-tuple $\Psi = \langle S, g, M, p_{max} \rangle$ where S is the current state of the system, g is the goal predicate, M is the behavior model of the system, and p_{max} is a limit on the plan length.

- Operation planner: An *operation planner* is an algorithm which given a planning problem Ψ, returns a sequence of operations that takes the system from its current state to a state where the goal predicate is satisfied. While planning, the operation planner is avoiding the running guards g_r and the running actions A_r, treating operation preconditions and postconditions as planning transitions.

- Plan: A *plan* P is a sequence of operations.

- Operation runner: An *operation runner* is an algorithm which executes the *plan* P based on the model M, the current state of the system S, and a goal predicate g. While running, the operation planner is considering both the planning and running components of guards and actions of operation pre- and postconditions. Moreover, the runner is taking all automatic transitions that are enabled as soon as possible, regardless of the queue.

- Communication: The *communication* protocol between resources and runners within a distributed system, as described in this work, is message-based. These messages are transmitted through various data channels, and depending on the number of receivers and the tasks they are executing, these channels may adopt either a publish/subscribe or request/response communication model. For the sake of simplicity, this work relies only on a request/reply communication model. The communication in this work is implemented with ROS2 which uses the DDS standard, however, the methods presented in this work can use any other message-based communication framework.

III. EXAMPLE

We investigate using a collaborative robot on a gantry to support the operators that use a pick-to-light system to assemble kits onto a trolley, see Fig 2. The robot can mount a battery-powered structured light scanner from a toolbox, which is used to detect and localize items that are to be picked from the blue boxes, see Fig 3. The toolbox also contains other tools which enable the robot to choose the appropriate tool to perform various operations. The system should allow the operators and robots to work in a shared zone, pick the ordered material together, and put it on an autonomous platform which will bring the material to an assembly station. Controlling such a system involves various challenges, including but not limited to ensuring the precision and reliability of scanning and localization of items, preventing collisions between the robot and operators or surrounding equipment, quickly and adequately responding to failures, and achieving a balance between execution speed to meet assembly cycle deadlines and ensuring operator safety and equipment protection.

IV. OPERATIONS

To define the Structural Coverability of Behavior Models for Intelligent Automation Systems (SCBM-IAS), we will discuss how they are executed. Consider the following:

```
operation: scan_box
  deadline: 10 seconds
  pre: start_scan_box
    g:  scan_req_state == initial &&
        scan_req_trigger == false &&
        box_is_scanned == false
    gr: true
    a:  [scan_req_trigger <- true]
    ar: []
  post: complete_scan_box
    g:  true
    gr: scan_req_state == succeeded
    a:  [scan_req_state <- initial,
        scan_req_trigger <- false,
        box_is_scanned <- true]
    ar: []
```

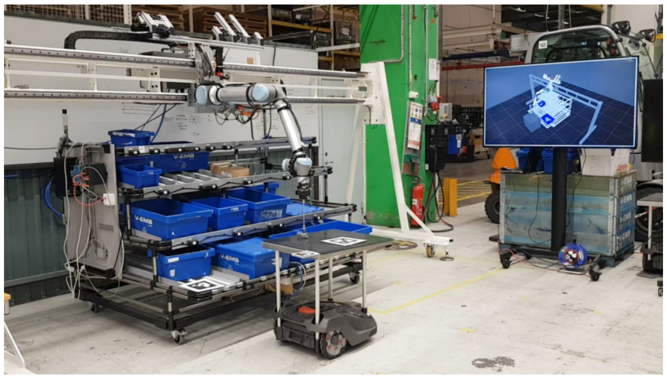

Fig. 2. Robot in the Air: An intelligent automation system.

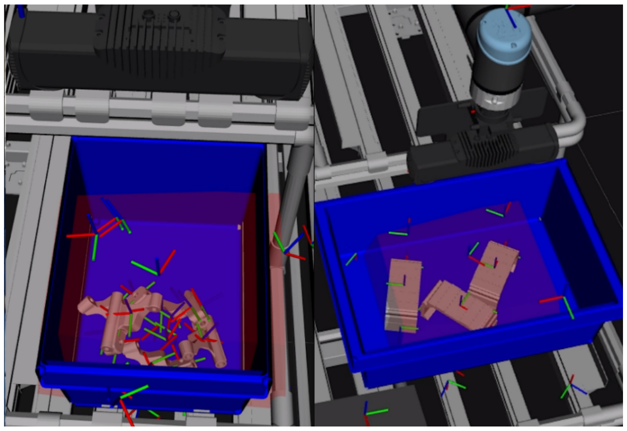

Fig. 3. A detail of scanning boxes for items in a simulation.

The operation `scan_box` models when and how the box containing some items should be scanned, and as shown on Fig 1, it has an `initial` and an `executing` state. Guards and actions of the operation are marked with `a`, `ar`, `g`, and `gr`, and are explained in more detail in Section 2. A `deadline` parameter specifies a time limit on the executing state, after which the operation should timeout.

Executing the `scan_box` operation triggers a request/response mechanism that will issue a command to the scanner and await a response. To do this efficiently, we keep track of the request state with the `scan_req_state` variable, and we enable the request to be issued with the `scan_req_trigger` variable. To keep track if the box has been scanned, we use the `box_is_scanned` variable.

Suppose that our system is in the following state:

```
goal: box_is_scanned == true
scan_req_state: initial
scan_req_trigger: false
box_is_scanned: false
```

While searching for a solution, the planning algorithm evaluates only the precondition guard `g`, and since it is evaluated to `true`, it first takes the precondition actions marked with `a` and then the postconditions actions marked with `a`. The planner constantly evaluates the goal predicate, and since it is now satisfied, it returns the following plan that contains one operation: `scan_box`.

The *runner* will now take this plan and try to execute it, which in the context of this work, means executing the operations in the order of the plan, one at a time. The procedure of executing operations is shown in Fig 4.

Continuing with the scanning example, the procedure of executing the `scan_box` operation is as follows. If the operation is the next one in the plan to be executed, the runner evaluates the precondition guard $\texttt{g} \wedge \texttt{gr}$. If the precondition guard $\texttt{g} \wedge \texttt{gr}$ is `false`, the operation will be in its `disabled` state until the guard becomes true. Otherwise, the runner will take all precondition actions from $\texttt{a} \cup \texttt{ar}$, and put the operation it its `executing` state.

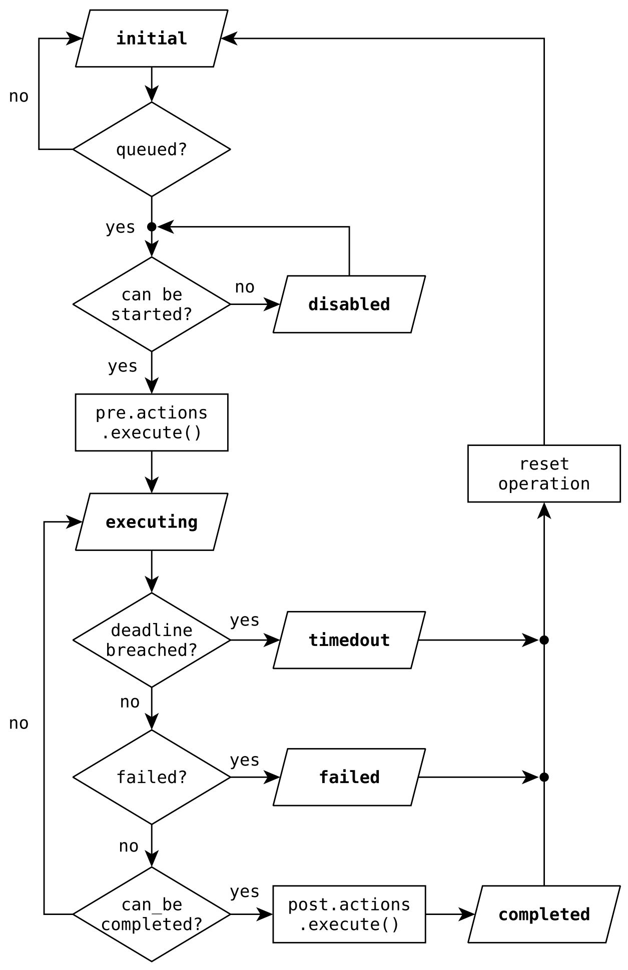

Fig. 4. An operation and its states while being executed by the runner.

At this point, a timer will start to keep track of how long the operation has been executing. If the operation does not complete, nor fail, within the next ten seconds, the operation will be `timedout`.

The `scan_box` operation can fail if the scanning response returns that the scanning process was unsuccessful, which could mean that something bad has happened. For example, causes for this could be that a very reflective surface has produced an invalid point cloud, or that the scanner itself has lost power, or that its driver was in an invalid state, etc. Such failures are caught with *automatic transitions*, after which the runner will put the operation in its `failed` state.

Finally, the operation can be `completed` if it is executing and if the postcondition guard $g \wedge gr$ evaluates to true. The runner will then take all postcondition actions from $a \cup ar$, and put the operation it its `completed` state. After the operation has completed, failed, or timedout, it is re-initialized so that it can be taken later, if necessary.

V. STRUCTURAL COVERABILITY

Measuring structural coverability involves quantifying the adequacy of the testing process and providing insights into the completeness of the test suite. This can be achieved by defining a set of coverage measures that indicate the degree to which the behavior model has been exercised during testing. For example, during the process of testing a system for a given set of requirements, it is possible to monitor and quantify the frequency and extent to which the behavior model is exercised. This evaluation can offer valuable insights into how to optimize the initial set of requirements and improve the overall coverability. By using this feedback to refine testing, we can ensure that the system is comprehensively tested to meet the required standards.

In this paper, we are inspired by the Modified Condition/Decision Coverage (MC/DC) criterion, however, this criterion cannot be directly applied to evaluate the coverage of behavior models. Therefore, we must shift our focus to the operation runner and compile an overview of the various running states of operation:

- ***Initial:*** The operation is not the next one in the queue.
- ***Disabled:*** The operation is next in the queue for execution, but the precondition guard is not yet enabled.
- ***Executing:*** The precondition guard is enabled and the actions of the precondition are taken.
- ***Timedout:*** The operation was in the executing state for more time than its deadline allows.
- ***Failed:*** The operations has failed due to an error.
- ***Completed:*** The postcondition guard is enabled and the actions of the postcondition are taken. The operation is successfully completed.

Using this overview, we can start defining the Structural Coverability of Behavior Models for Intelligent Automation Systems (SCBM-IAS) criterion for a test set as follows:

- Every queued operation in the behavior model has visited its Disabled, Executing, Timedout, Failed, and Completed, state at least once.
- Every operation has been queued in a plan at least once.
- Every automatic transition has been taken at least once.

Fulfilling these criteria for a set of requirements ensures that the behavior model has been exercised and can be used as a sanity check. Just as with MC/DC, meeting the requirements of SCBM-IAS does not guarantee that no defects remain. However, failing to meet this criterion indicates that certain portions of the model have not been sufficiently exercised, highlighting potential areas of concern. Similarly to MC/DC, SCBM-IAS is given as a percentage.

Improving the SCBM-IAS is an iterative process. When the criteria are not met, it is necessary to go back to the model and identify the missed parts, then create specific test cases to cover them. Alternatively, the tester can be allowed to influence the simulation nodes to cover the missed aspects faster. After additional tests, the SCBM-IAS can be reevaluated, and the process repeated until the desired level of coverage is achieved.

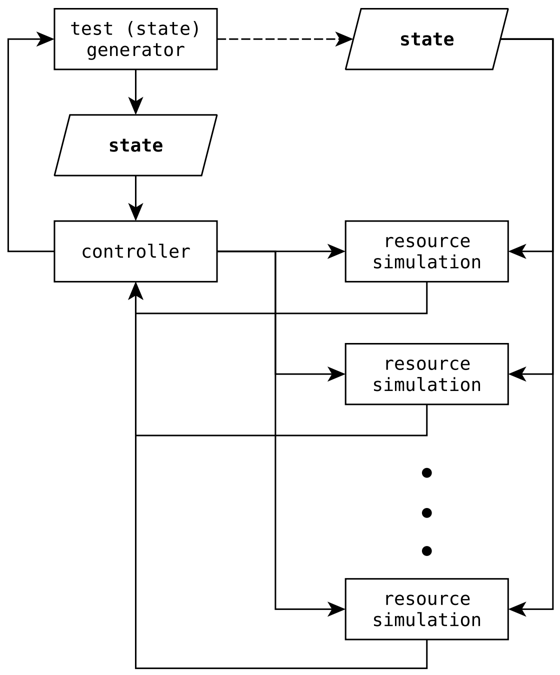

Fig. 5. The mutation inspired testing framework.

VI. TESTING

To test Intelligent Automation Systems, we propose a procedure that involves testing requirements, random test case generation, coverability analysis, and creating finishing test cases. We differentiate three groups of nodes: the controller, the simulated resources, and the tester, as seen in Fig. 5.

Our testing procedure consists of the following steps:

1) Specification of a set of achievable goals.
2) Specification of requirements that must be met while achieving these goals.
3) Generation of a set of test cases by the tester, which consists of randomly chosen achievable goals and random initial states, which are then transmitted one by one to the controller and the simulated resources.
4) Execution of each test case by the controller and with the results reported back to the tester. During the execution of a test, the simulated resources can randomly choose to fail, timeout, or succeed.
5) Collection of data from the controller in order to determine the degree of SCBM-IAS after the entire set of generated test cases has been exhausted.
6) If the SCBM-IAS is deemed unsatisfactory, identification of the specific areas of the behavior model that have not been sufficiently covered by the test cases.
7) Creation of additional targeted test cases that address the missed areas of the behavior model, with the aim of increasing SCBM-IAS.

By utilizing this testing procedure, we can ensure that the behavior model of our Intelligent Automation Systems is thoroughly tested for the specified set of requirements.

VII. EVALUATION

To limit the example from Section 3, we consider the following objective: An order arrives which consists of only one item from the material facade, `item_a`. The robot should use the scanner to scan a box `box_a` for items, after which it should be able to pick one of them using a gripper and place it on the autonomous platform, AGV. As the robot can only carry one tool at a time, either the gripper or the scanner can be mounted and unmounted as required. Meanwhile, the gantry has to move as well to allow the robot to reach the boxes and the AGV.

To further simplify the example, we implement the control for all resources as a request/reply mechanism, thus all operations in our model have a similar structure to the scanning operation shown before. The behavior model of the system consists of the following sixteen operations:

- `robot_move_to: home, toolbox_gripper, toolbox_scanner, box_a, agv`
- `robot_mount: scanner, gripper`
- `robot_unmount: scanner, gripper`
- `scanner: scan_item_a`
- `gantry_move_to: box_a, agv`
- `gripper: open, close`
- `pick: item_a`
- `place: item_a`

Next, we define goals that should be possible to achieve:

- `scanned_a == true`
- `gripper_act == closed` $\wedge$ `robot_at == agv`
- `item_a_at == agv` $\wedge$ `robot_at == box_a`
- `item_a_at == gripper`

As a first effort, after defining a few requirements that we want to test our system for, we will let the tester generate random initial states, and we will let the resource simulations choose random responses. We allow the tester to generate twenty test cases, and collect the data:

```
Coverabilty report: CSBM-IAS:...41.86 %
Testing duration:..............134.17s
=======================================
Visited disabled state:.........0  / 16
Visited executing state:........13 / 16
Visited failed state:...........7  / 16
Visited timedout state:.........0  / 16
Visited completed state:........13 / 16
Automatic transitions taken:....3  / 6
```

We observe that no operation has visited its `timedout` state, so we can allow the tester to influence the simulators and force the executions to take longer so that the timeouts are triggered. Moreover, we see that only a few operations have visited its `failed` state, which turns out only to be a lucky run. We can allow the test to influence the outcome of the request call, for example, to increase the chance to fail, or even to force it to fail, however for the next run we did not change those parameters, which shows the nature of random testing. After the next run, we get the following result:

```
Coverabilty report: CSBM-IAS:...65.12 %
Testing duration:...............212.24s
=======================================
Visited disabled state:.........0  / 16
Visited executing state:........14 / 16
Visited failed state:...........11 / 16
Visited timedout state:.........12 / 16
Visited completed state:........14 / 16
Automatic transitions taken:....5  / 6
```

We see from the report that the disabled states are not visited. If an operation is in its disabled state, it means that the operation is queued but the precondition guard disables the operation from being taken. Most often, it is the runner guard `gr` that disables the operation, indicating that something has to happen in the environment before we can take the operation. An example of this can be a sensor, awaiting an object to be present before the operation can start and the execution of the plan can continue.

In our case though, we have to simulate that $g \land gr$ is false when the operation is queued. Thankfully, we can influence the simulated environment, and continuing with the scanner operation example, force `scan_req_state` to be something else than `initial`. Practically, this could mean that the scanner resource is occupied by another process.

Moreover, we can see that two operations and one automatic transition were not taken, so if we look at the detailed report, we can see that the following were missed:

```
at: abort_if_scanning_timedout_5_times
op: robot_move_to_home
op: robot_unmount_gripper
```

To address the first issue, we allow the test to further increase the scanning time in order to cover scenarios where scanning has `timedout` five times while executing a plan. The next missing operation reveals a deeper problem, which involves a missing requirement specification. In particular, during gantry movement, it is essential for the robot to remain in its home position to prevent collisions with operators and other obstacles. This can be fixed by extending the list of requirements and repairing the behavior model to incorporate an additional guard condition for the `gantry_move` operations. This condition would ensure that the robot remains in its `home` position while the gantry is in motion.

Finally, the third missing piece can be fixed by extending the list of achievable goals with the following:

- `mounted == scanner` $\land$ `item_a_at == agv`

When chosen by the random tester, this goal will force the gripper to be unmounted after the object has been placed. After fixing all the previous missing points, increasing the test case generation size to thirty, and two runs where some states were missed due to the nature of random testing, we finally get the following report:

```
Coverabilty report: CSBM-IAS:...100   %
Testing duration:...............379.84s
```

VIII. Conclusion

The Structural Coverability of Behavior Models for Intelligent Automation Systems (SCBM-IAS) criterion was introduced to assist in the development of an proper test suite. This criterion is helpful in identifying areas of the model that have not been adequately exercised, thus guiding the creation of additional test cases. In this work, we used the proposed criterion to manually come up with new test cases but the long-term goal is to use the same criteria to automatically modify the behavior of the system under test in order to increase the coverability.

References

[1] R. Hametner, B. Kormann, B. Vogel-Heuser, D. Winkler, and A. Zoitl, “Test case generation approach for industrial automation systems,” in *The 5th International Conference on Automation, Robotics and Applications*, 2011, pp. 57–62.

[2] D. Winkler, R. Hametner, T. Östreicher, and S. Biffl, “A framework for automated testing of automation systems,” in *2010 IEEE 15th Conference on Emerging Technologies and Factory Automation (ETFA 2010)*, 2010, pp. 1–4.

[3] P. Godefroid and K. Sen, “Combining model checking and testing,” in *Handbook of Model Checking*, 2018.

[4] S. Sebastian, S. Magnus, M. Thron, H. Zipper, U. Odefey, V. Fäßler, A. Strahilov, A. Kłodowski, T. Bär, and C. Diedrich, “Test methodology for virtual commissioning based on behaviour simulation of production systems,” in *2016 IEEE 21st International Conference on Emerging Technologies and Factory Automation (ETFA)*, 2016, pp. 1–9.

[5] X. Cai and M. R. Lyu, “The effect of code coverage on fault detection under different testing profiles,” in *Proceedings of the 1st International Workshop on Advances in Model-Based Testing*, ser. A-MOST ’05. New York, NY, USA: Association for Computing Machinery, 2005, pp. 1–7.

[6] K. Hayhurst and D. Veerhusen, “A practical approach to modified condition/decision coverage,” in *20th DASC. 20th Digital Avionics Systems Conference (Cat. No.01CH37219)*, vol. 1, 2001, pp. 1B2/1–1B2/10 vol.1.

[7] K. Maruchi, H. Shin, and M. Sakai, “MC/DC-Like structural coverage criteria for function block diagrams,” in *2014 IEEE Seventh International Conference on Software Testing, Verification and Validation Workshops*, 2014, pp. 253–259.

[8] H. Hemmati, “How effective are code coverage criteria?” in *2015 IEEE International Conference on Software Quality, Reliability and Security*, 2015, pp. 151–156.

[9] S. Macenski, T. Foote, B. Gerkey, C. Lalancette, and W. Woodall, “Robot operating system 2: Design, architecture, and uses in the wild,” *Science Robotics*, vol. 7, no. 66, may 2022.

[10] M. Dahl, E. Erős, K. Bengtsson, M. Fabian, and P. Falkman, “Sequence planner: A framework for control of intelligent automation systems,” *Applied Sciences*, vol. 12, no. 11, 2022.

[11] “Rust: A language empowering everyone to build reliable and efficient software.” https://www.rust-lang.org/, accessed: 2023-03-17.

[12] “R2r - easy to use, runtime-agnostic, async rust bindings for ROS2.” https://github.com/sequenceplanner/r2r, accessed: 2023-03-17.

[13] A. Santos, A. Cunha, and N. Macedo, “The high-assurance ROS framework,” in *2021 IEEE/ACM 3rd International Workshop on Robotics Software Engineering (RoSE)*, 2021, pp. 37–40.

[14] ——, “Property-based testing for the robot operating system,” in *Proceedings of the 9th ACM SIGSOFT International Workshop on Automating TEST Case Design, Selection, and Evaluation*, ser. A-TEST 2018. New York, NY, USA: Association for Computing Machinery, 2018, p. 56–62.

[15] R. Carvalho, A. Cunha, N. Macedo, and A. Santos, “Verification of system-wide safety properties of ROS applications,” in *2020 IEEE/RSJ International Conference on Intelligent Robots and Systems (IROS)*, 2020, pp. 7249–7254.

[16] M. Papadakis, M. Kintis, J. Zhang, Y. Jia, Y. L. Traon, and M. Harman, “Chapter six - mutation testing advances: An analysis and survey,” in *Advances in Computers*, ser. Advances in Computers, A. M. Memon, Ed., vol. 112. Elsevier, 2019, pp. 275–378.