Fault localization for intelligent automation systems

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

***Abstract*—Conventional programming of explicit control code is unsuitable for flexible and collaborative production systems. A model-based approach, which focuses on defining capabilities of a system, instead of specifying how to achieve them, provides an alternative for creating complex, scalable, and reliable systems. This is accomplished through the use of behavior models, and tools such as planning, synthesis, verification, and testing. However, developing such models is not without challenges, as it is possible to overlook or incorrectly specify potential behavior and constraints. This can result in unsolvable planning problems or plans that are invalid for other reasons. When plans are unobtainable, developers receive no feedback, which makes model adjustments a difficult and time-intensive task. This paper recognizes these challenges as crucial barriers for adopting model-based development of intelligent automation systems. To facilitate the development of such systems, an approach for detecting and localizing faults in behavior models is presented. Drawing inspiration from software fault localization techniques, the proposed method involves identifying suspicious resources, variables, and operations. The effectiveness of this approach is illustrated with an example use case.**

I. INTRODUCTION

The aim of this paper is to explore a fault localization support system for the development of intelligent automation systems [1]. Such systems are based on automated planning [2] which enables them to *plan*, *act*, *re-plan*, and safely *recover* from undesired states when necessary. By adopting a planning approach to automation as described in [3], the behavior of the system can be modeled through variables, states, transitions, and operations. This enables the use of verification, synthesis, and testing tools, which can assist creating automation systems that are safe, flexible, and robust.

However, such an approach comes with its own set of challenges. While modeling a system, developers may introduce faulty behavior and constraints, or simply forget to specify certain behavior, resulting in unsolvable planning problems or anomalous planning results. Even with a test-driven development approach [4], where developers iteratively test the model, it is often hard to identify the root cause of *why* a test failed.

Automated planning and synthesis can be achieved using various approaches. For example, a commonly used method is the planning-as-model-checking approach [5] where the planning problem is transformed into a verification problem. In this approach, the negation of the planning problem's goal is established as a safety property to be verified. If it is possible to violate the safety property, the model-checker will generate a counter-example that can be used as the plan to reach the goal. Otherwise, if the model-checker determines that no counter-example exists, it serves as evidence that no feasible plan can be found. However, *explaining* the unsolvability of a planning problem is a challenging task even with this approach, since there is no usable counter-example that can explain why no plan exists.

Related work for explaining that no plan exists includes detecting the unsolvability of a plan [6], [7], generating certificates or proofs of unsolvability [8], [9], and identifying adjustments to the planning problem that could render the problem solvable [10]. In that work, the focus is on investigating the initial state as the reason behind the inability to find a plan. This approach assumes the correctness of the models, thereby making the modification of the initial state a feasible option. In [11], the authors address this issue by capturing the user's expectations through considering abstractions of the given problem. More specifically, the authors use state abstractions to produce potential solutions and subgoals at higher levels of abstraction.

However, none of the aforementioned research considers the inability to find plans due to modeling mistakes introduced by the model developers themselves. Such faults are inadvertently incorporated during model development. The purpose of iterative testing is to catch and correct such mistakes as early as possible, which can be done by unit testing the model, i.e. evaluating different initial and goal states with a planning algorithm. Such evaluations might reveal that a plan is *valid*, *anomalous*, or *unsolvable*. Passing a test allows the developer to continue refining the model. Anomalous results are not desirable, however they still offer developers valuable feedback. Sometimes, a test can quickly demonstrate that a goal state has been achieved in a manner that violates certain user-defined specifications. This allows a developer to identify potential flaws or shortcomings of the model.

The problem addressed in this paper, is to enhance the feedback provided to the developer after failing to generate a plan. This situation typically arises when the developer has inaccurately modeled the system's behavior. Such mistakes are typically easier to detect and handle in the early design phases.

In this paper, we are inspired by software fault localization techniques [12], which have been extensively studied and applied in the field of software engineering. For example, a technique based on combinatorial testing separates input parameters into faulty-possible and healthy-possible to identify minimal failure-inducing combinations of parameters [13], [14]. Moreover, predicate switching [15] is a program fault

[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.

localization technique that involves altering program states to force execution along different branches during a failed run. If switching a predicate results in a successful program execution, such predicate is identified as critical. Finally, model-checking approaches to fault localization also exist [16], [17], where a model checker can provide a counter-example if a program fails to meet its specification.

More specifically, this paper draws inspiration from delta debugging techniques [18] [19] which identify the cause of software failures by comparing the program states between successful and failed tests. Suspicious variables are identified by replacing their values from the successful test with their corresponding values from the same point in the failed test and then executing the program again. If the same failure occurs, the variable is considered suspicious, otherwise, it is no longer considered as a potential cause of the failure.

However, we do not directly apply such techniques to behavior models of intelligent automation systems. Instead, we define modeling abstractions called *operations* and *resources*, and test them for *suspiciousness* in a three-step approach. Firstly, we test the failed problems with relaxed versions of the model, where we remove complete resources in an effort to isolate potentially problematic ones. Depending on the outcome of the test, this might give an indication of which resources are incorrectly modeled. Next, we iteratively test the relaxed versions of the model by removing and adding back variables from suspicious resources, which provides us with a list of suspicious variables. However, this does not indicate *where* the problem in the model might be, so in the next step we identify operations that update the suspicious variables. Finally, we provide a list of achievable initial-goal state combinations to iteratively reduce the list of operations that were not triggered during random testing [20].

The main contribution of this paper is a fault localization method for aiding the development of intelligent automation systems. This method can be used to identify certain faults in the behavior model during testing and test-driven development. This paper is based on our previous work from [1], which relies on Robot Operating System 2 (ROS2) [21]. The effectiveness of this approach is exemplified by a use-case consisting of a robotic manipulator, a gantry, a structured light scanner, and a gripper.

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.

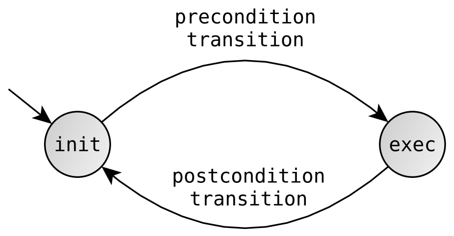

Fig. 1. A model of an operation.

- 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 action* 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 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.

- 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. The operation planner can return a plan which is *unsolvable*, *anomalous*, or *valid*. An *unsolvable* plan means that the planner was not able to find a plan to a state where the goal predicate is satisfied. An *anomalous* plan means that a plan was found to a state where the goal predicate is satisfied, but in a way where the safety specifications are violated. A *valid* plan means that the planner was able to find a plan to a state where the goal predicate is satisfied without violating the safety specifications.

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.

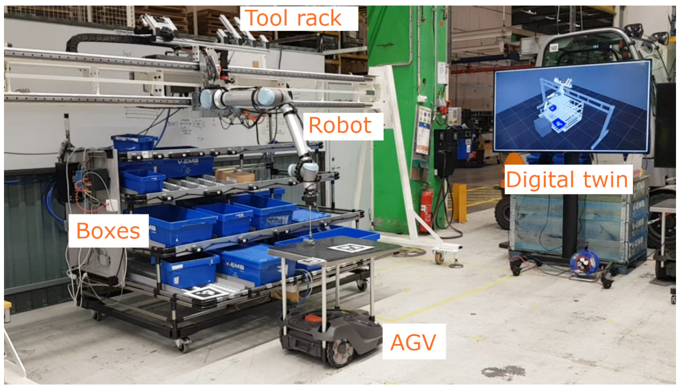

Fig. 2. Robot in the Air: An intelligent automation system.

IV. OPERATIONS

To identify faults that can occur while planning and executing operations, let us consider the following:

```
operation: scan_box_a
  pre: start_scan_box_a
    g:  scan_req_state == initial &&
        scan_req_trigger == false &&
        item_a_scanned == false &&
        robot_mounted == scanner &&
        robot_position == box_a &&
        gantry_position == box_a
    gr: true
    a:  [scan_req_trigger <- true]
    ar: []
  post: complete_scan_box_a
    g:  true
    gr: scan_req_state == succeeded
    a:  [scan_req_state <- initial,
        scan_req_trigger <- false,
        item_a_scanned <- true]
    ar: []
```

The operation `scan_box_a` models when and how the box containing some items should be scanned, and as shown in Fig. 1, it has an `initial` and an `executing` state. Executing the `scan_box_a` 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 item in the box has been scanned, we use the `item_a_scanned` variable.

Assume that the planning algorithm has calculated the following plan:

```
robot_mount_scanner
gantry_move_to_box_a
robot_move_to_box_a
scan_box_a
```

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. Continuing with the scanning example, the procedure of executing the `scan_box_a` 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 this precondition guard is false, the operation will be `disabled` until the guard becomes true. Otherwise, the runner will take all precondition actions from $\texttt{a} \cup \texttt{ar}$, and put the operation in its `executing` state.

Finally, the operation can be `completed` if it is executing and if the postcondition guard $\texttt{g} \wedge \texttt{gr}$ evaluates to true. The runner will then take all postcondition actions from $\texttt{a} \cup \texttt{ar}$, and put the operation back to it its `initial` state.

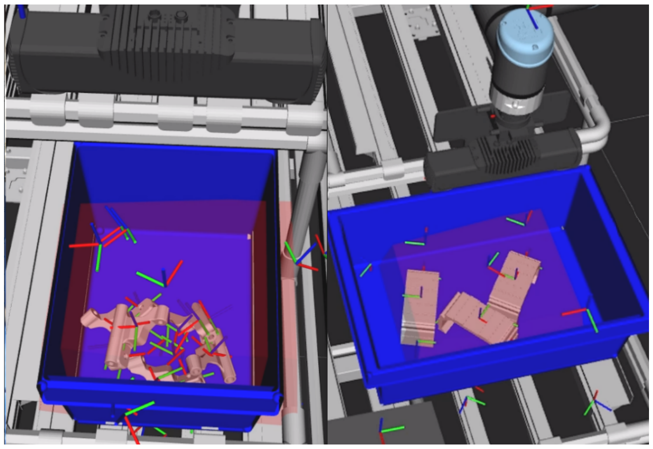
Fig. 3. A detail of scanning boxes for items in a simulation.

V. FAULT LOCALIZATION

Developing a behavior model for an intelligent automation system is often an iterative process where the model is developed incrementally. Modeling the behavior in incremental steps makes it possible to catch and correct mistakes early on, and might reduce the total time to develop a full model. However, sometimes it is not easy to anticipate how adding additional variables, adding new or changing transitions and operations will affect the model. Often, the developer has no knowledge if the model is broken until it is tested again for some provided initial state and goal predicate.

We identify different outcomes when testing an intelligent automation system, and separate the testing process into two parts. Initially, only the *planning* part of the model is tested, by manually supplying the initial and goal states, which are then provided to a planning algorithm. At this point, we are only using the operation planner which is avoiding the running guards g_r and the running actions A_r, treating operation preconditions and postconditions as planning transitions. This can return a plan that is:

- *Valid* - Verifies that the model does what it is supposed to do, namely that the plan can be used to reach the goal state without breaking some user-defined specifications.
- *Anomalous* - This result is unfavourable, however it still provides some feedback to the developers as it can sometimes quickly show that the goal state was reached in a way that breaks some user-defined specifications. For example, it might show that an item was moved to its goal destination before it was even picked up, indicating an incomplete guard condition for the *move* operation.
- *Unsolvable* - This is the most unfavourable result as it gives no feedback to the developers other than that the goal state can not be reached. Assuming that the planning algorithm is correct and that the developer has provided initial and goal states which should be solvable, this indicates that the mistake is located in the model. We are addressing this case in this paper.

After the initial testing of the *planning* part of the model, the complete model is verified using a simulation (and later a real) environment by executing plans which also include the running guards g_r and the running actions A_r. The model and the goal are provided by the developer, while the initial state can be observed from the environment. This can result in executions which are:

- *Valid* - A plan was found and its execution (using a simulation or a real environment) verifies that it is correct.
- *Anomalous* - A plan is found, however a previously unknown mistake in the model results in an anomalous plan. Executing this plan shows irregular behavior indicating that there is a mistake in the model.
- *Blocked* - A plan is found, however its execution only proceeds until a certain state where the next operation is blocked due to a runner guard predicate g_r forbidding the operation to be taken.
- *Unsolvable I* - A plan is not found because it can not be calculated from the currently observed initial state. For example, this can happen if a robot has to unlock a door to be able to open it and go into the other room, however in the initial state, the door is locked and the key is behind the door, making the problem unsolvable.
- *Unsolvable II* - A plan is not found because the model contains a previously unknown fault. We are addressing this case in this paper.

In this paper, we address the *Unsolvable* and *Unsolvable II* cases from the previous classifications, and the first step towards doing so is to collect variables in certain groups. A natural way to do that is to use *resources*, however, the presented algorithms do not require a very specific variable grouping. This is mostly only necessary to enable a hierarchical search for faults, which means that certain *product*, *memory*, and other variables can also be grouped.

Algorithm 1: *identify_suspicious_resources*

```
Input: $M$, $resources$, $goal$, $d_{max}$, $p_{max}$
Output: $resources$
1  $d \leftarrow 1$
2  $\mathcal{R}_{sus} \leftarrow \emptyset$
3  while $d \leq d_{max}$ do
4      let combinations = resources.combinations(d)
5      for comb in combinations do
6          let relaxed_m = model.relax(comb)
7          let relaxed_g = goal.relax(comb)
8          let result = plan(relaxed_m, relaxed_g, $p_{max}$)
9          if result.found and result.len() != 0 then
10             $\mathcal{R}_{sus}$.push(comb)
11             return $\mathcal{R}_{sus}$
12         end
13     end
14     d = d + 1
15 end
16 return $\mathcal{R}_{sus}$
```

In automation systems, 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. During the development phase of an automation system, such resources can be simulated to allow developers to test and validate the implemented behavior model before deploying it in a real-world environment.

Algorithm 1 abstracts the model and the goal by removing such variable groups (resources) and solving relaxed version of the problem, where M is the model, d_{max} is the maximum number of resources to be removed, and p_{max} is the plan length limit. If a solution exists, the removed resources are marked as suspicious and the algorithm terminates.

Now that the suspicious resources have been identified, we would ideally like to determine which variables are the cause of the fault. To do so, Algorithm 2 takes a list of suspicious resources which the previous algorithm has identified, and collects all the variables that model those resources. Algorithm 2 now abstracts the original model and the goal by removing such variables and solving a relaxed version of the problem, where M is the model, d_{max} is the maximum number of variables to be removed, and p_{max} is the plan length limit. If a solution exists, the removed variables are marked as suspicious and the algorithm terminates.

Algorithm 2 returns a set of suspicious variables, which means that their values are either incorrectly updated somewhere in the model, or that a guard predicate containing the variable is too restrictive or incorrect. Such variables are identified as suspicious based on the fact that when removed from an operation, the planner is able to use that operation to determine a plan.

Algorithm 2: *identify_suspicious_variables*

```
   Input: $M, resources, goal, d_{max}, p_{max}$
   Output: variables
1  $d \leftarrow 1$
2  $\mathcal{V}_{sus} \leftarrow \emptyset$
3  let variables = resources.collect_variables()
4  while $d \leq d_{max}$ do
5      let combinations = variables.combinations(d)
6      for comb in combinations do
7          let relaxed_m = model.relax(comb)
8          let relaxed_g = goal.relax(comb)
9          let result = plan(relaxed_m, relaxed_g, $p_{max}$)
10         if result.found and result.len() != 0 then
11             $\mathcal{V}_{sus}$.push(comb)
12             return $\mathcal{V}_{sus}$
13         end
14     end
15     d = d + 1
16 end
17 return $\mathcal{V}_{sus}$
```

Algorithm 3: *identify_suspicious_operations*

```
   Input: M, variables
   Output: operations
1  $\mathcal{O}_{sus} \leftarrow \emptyset$
2  for op in M.operaions do
3      for var in variables do
4          if op.contains(var) then
5              $\mathcal{O}_{sus}$.push(op)
6          end
7      end
8  end
9  return $\mathcal{O}_{sus}$
```

However, this does not indicate *where* the problem in the model might be, so in the next algorithm identifies operations which update the suspicious variables. Algorithm 3 is a procedure that filters out and returns a set of operations that contain suspicious variables, either in the guard predicate or the action function of the planning preconditions and postconditions. Quite often, this step can clearly identify which operations are problematic and correctly localize the faults. However, let's assume that a suspicious variable is contained in all operations and that it is incorrectly updated only in one operation. In this case, Algorithm 3 will return the set of all operations as all of them contain the suspicious variable, which gives no indication of where the mistake might originate.

To handle such situations and to possibly further reduce the set of suspicious operations, Algorithm 4 can be used. Algorithm 4 takes a set of tuples which are pairs of user-defined initial state and goal predicate combinations. The purpose of this is to test the model for all such provided combinations and track which operations have been taken. Ideally, the provided initial-goal combinations should cover a large section of the model, and at the same time not produce very long plans. The reasoning behind this is to try to cover as much of the model possible with shorter plans, and *take* as many operations as possible.

Some of these provided combinations will be solvable, eliminating the operations contained in the calculated plan, and making the set of suspicious operations smaller, as seen in Fig. 4. Finally, operations that were missed by not providing enough initial-goal combinations and which do not contain suspicious variables are filtered out. What we are left with is a set of operations that are incorrectly modeled and a set of variables that indicate what is wrong in such operations.

Sometimes though, another approach might seem more natural and it doesn't involve using Algorithm 4. Instead, specific operations (from the resulting set returned by Algorithm 3) can be targeted with regular unit testing, by providing initial states and goal predicates that will ideally only need that single operation in the plan to reach the goal. If such a plan doesn't exist, the fault is found. Ultimately, it is the developer's choice and expert knowledge that will determine how a certain system will be modeled, developed, and tested.

Algorithm 4: *localize_suspicious_operations*

```
Input: $M, variables, [\langle initial, goal\rangle], p_{max}$
Output: operations
1  $\mathcal{O}_{sus} \leftarrow \emptyset$
2  $\mathcal{O}_{diff} \leftarrow \emptyset$
3  $\mathcal{O}_{taken} \leftarrow \emptyset$
4  for $\langle$initial, goal$\rangle$ in $[\langle$initial, goal$\rangle]$ do
5      let result = plan(M, initial, goal, $p_{max}$)
6      if result.found and result.len() != 0 then
7          for op in result.plan do
8              $\mathcal{O}_{taken}$.push(op)
9          end
10     end
11 end
12 $\mathcal{O}_{diff} \leftarrow$ M.operations.diff($\mathcal{O}_{taken}$)
13 for op in $\mathcal{O}_{diff}$ do
14     for var in variables do
15         if op.contains(var) then
16             $\mathcal{O}_{sus}$.push(op)
17         end
18     end
19 end
20 return $\mathcal{O}_{sus}$
```

VI. 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.

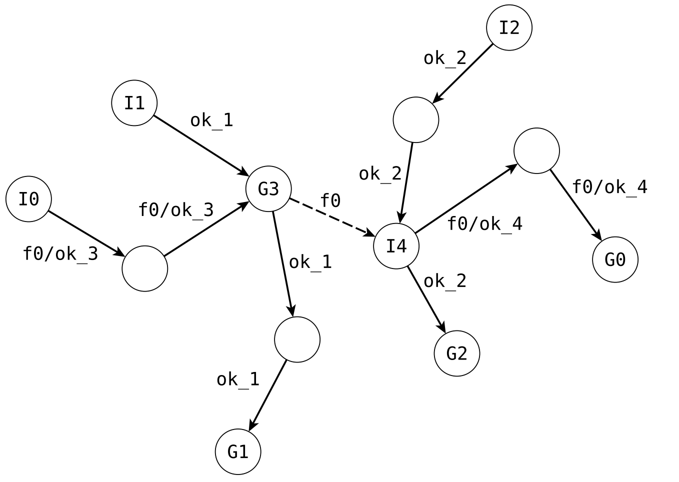

Fig. 4. Localizing a faulty operation with Algorithm 4 by removing taken operations. Four initial state and goal predicate combinations were provided, I0..3/G0..3, where I0/G0 was unsolvable because of a faulty operation. After solving I1/G1, I2/G2, I0/G3, and I4/G0, the taken operations marked with *ok* are removed. What is left is the faulty operation f0.

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. The behavior model of the system is modeled with the following eighteen variables grouped in five resources:

```
robot:   request_trigger, command,
         position, request_state,
         actual_state, mounted
gantry:  request_trigger, command,
         request_state, actual_state
gripper: request_trigger, command,
         request_state, actual_state
scanner: request_trigger, request_state
item_a:  scanned, position
```

Such variables are used to model the following 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
```

Let us assume that we have modeled the previous operations and that we would like to test our model for the following initial state and goal predicate:

```
init: item_a_position == box_a,
      robot_position == home,
      robot_mounted == none,
      ...
goal: item_a_position == agv
```

If the model were correct, the planner would produce the following sequence of operations: *gantry move to box a, robot move to toolbox scanner, robot mount scanner, robot move to box a, scan box a, robot move to toolbox scanner, robot unmount scanner, robot move to toolbox gripper, robot mount gripper, open gripper, robot move to box a, pick a, gantry move to agv, robot move to agv, place a.*

However, since there is a fault somewhere in the model, the only feedback that is provided for the developers is:

```
found: false
length: 0
plan: [ ]
```

The model of the presented system is relatively small, however, manually finding the cause of this unsolvability is still an involved task.

Instead, we can utilize algorithms 1, 2, and 3, and get:

```
Suspicious resources: [scanner, gripper]
Suspicious variables:
  Variable 1: scanner_request_state
  Variable 2: gripper_request_state
Suspicious operations:
  Operation 1: scan_box_a
  Operation 2: open_gripper
  Operation 3: close_gripper
```

After a closer look at these operations, it turns out that the guard predicates in operations *scan_box_a* and *open_gripper* were incorrectly modeled, expecting the variables *scanner_request_state* and *gripper_request_state* to have the values *initial* instead of *enabled*. Correcting such mistakes makes the tests pass with no suspicious resources.

However, what if the mistake was hidden in an action which incorrectly updates some variables? Let's assume that, by accident, we have not updated the next value of the *robot_position* in the *move* operations correctly, and that we have left the previous value as the next state update. We get:

```
Suspicious resources: [robot]
Suspicious variables:
  Variable 1: robot_position
Suspicious operations:
  Operation 1: scan_box_a
  Operation 2: robot_move_to_home
  Operation 3: robot_move_to_tb_gripper
  Operation 4: robot_move_to_tb_scanner
  Operation 5: robot_move_to_box_a
  Operation 6: robot_move_to_agv
  Operation 7: robot_mount_scanner
  Operation 8: robot_mount_gripper
  Operation 9: robot_unmount_scanner
  Operation 10: robot_unmount_gripper
  Operation 11: pick_a
  Operation 12: place_a
```

From this result it is quite clear that the problem is with the *robot* resource, specifically how the *robot_position* variable is being updated. However, the *robot_position* variable models twelve operations, which even in this small example makes it inefficient to manually search for the mistake. One way to localize this fault would be to provide a set of simpler achievable initial-goal state combinations and utilize Algorithm 4 to reduce the number of suspicious operations.

However, as the only suspicious variable that has been identified is the *robot_position*, it might be faster to just test a some of these operations and see if the planner can use them to solve the following trivial problem:

```
initial: robot_position <- home, ...
goal:    robot_position == box_a
```

Immediately, we get a no plan found, which is a clear indication that the *robot_position* is incorrectly updated in the *move* operations.

A situation that can benefit from Algorithm 4 is when Algorithm 3 returns a large number of suspicious variables or operations, or when it is not quickly clear which operation might be the cause of the issue. To keep the example small and clear, we continue with the previous example and define two init-goal combinations which should force the gripper to open and close. As algorithm 4 keeps track of taken operations during this testing, we get a smaller set of suspicious operations:

```
Suspicious resources: [scanner, gripper]
Suspicious variables:
  Variable 1: scanner_request_state
  Variable 2: gripper_request_state
Suspicious operations:
  Operation 1: scan_box_a
  Operation 2: open_gripper
  Operation 3: close_gripper
Suspicious locations:
  Operation 1: op_scan_box_a
  Operation 2: op_open_gripper
```

The effectiveness and reliability of this approach were tested with modeling ROS2 based intelligent automation systems within the framework of our previously published work from [1]. Thus, even if shown effective when modeling an industrial use-case presented in this paper, using this approach with other modeling frameworks might have some limitations.

VII. CONCLUSION

A fault localization approach for developing behavior models for intelligent automation systems was introduced, with the aim to assist model developers in creating and testing such models. This approach can identify suspicious resources, variables, and operations, and assist the developers in the iterative test-driven model development process. In this work, we have used the presented algorithms to localize potential faults in the model, but the long-term goal is to try to automatically synthesize suggestions in the form of operations and show how to improve the model.

REFERENCES

[1] 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.

[2] M. Ghallab, D. Nau, and P. Traverso, *Automated Planning and Acting*, 1st ed. USA: Cambridge University Press, 2016.

[3] M. Dahl, *Preparation and control of intelligent automation systems*. Gothenburg: Chalmers tekniska högskola, 2021.

[4] D. Astels, *Test Driven Development: A Practical Guide*. Prentice Hall Professional Technical Reference, 2003.

[5] F. Giunchiglia and P. Traverso, "Planning as model checking," in *Recent Advances in AI Planning: 5th European Conference on Planning, ECP'99, Durham, UK, September 8-10, 1999. Proceedings 5*. Springer, 2000, pp. 1–20.

[6] C. Bäckström, P. Jonsson, and S. Ståhlberg, "Fast detection of unsolvable planning instances using local consistency," in *Proceedings of the International Symposium on Combinatorial Search*, vol. 4, no. 1, 2013, pp. 29–37.

[7] J. Hoffmann, P. Kissmann, and A. Torralba, "" distance"? who cares? tailoring merge-and-shrink heuristics to detect unsolvability." in *ECAI*, 2014, pp. 441–446.

[8] S. Eriksson, G. Röger, and M. Helmert, “Unsolvability certificates for classical planning,” *Proceedings of the International Conference on Automated Planning and Scheduling*, vol. 27, no. 1, pp. 88–97, Jun. 2017. [Online]. Available: https://ojs.aaai.org/index.php/ICAPS/article/view/13818

[9] ——, “A proof system for unsolvable planning tasks,” *Proceedings of the International Conference on Automated Planning and Scheduling*, vol. 28, no. 1, pp. 65–73, Jun. 2018. [Online]. Available: https://ojs.aaai.org/index.php/ICAPS/article/view/13899

[10] M. Göbelbecker, T. Keller, P. Eyerich, M. Brenner, and B. Nebel, “Coming up with good excuses: What to do when no plan can be found,” *Proceedings of the International Conference on Automated Planning and Scheduling*, vol. 20, no. 1, pp. 81–88, May 2021. [Online]. Available: https://ojs.aaai.org/index.php/ICAPS/article/view/13421

[11] S. Sreedharan, S. Srivastava, D. Smith, and S. Kambhampati, “Why can’t you do that hal? explaining unsolvability of planning tasks,” in *International Joint Conference on Artificial Intelligence*, 2019.

[12] W. E. Wong, R. Gao, Y. Li, R. Abreu, and F. Wotawa, “A survey on software fault localization,” *IEEE Transactions on Software Engineering*, vol. 42, no. 8, pp. 707–740, 2016.

[13] C. Nie and H. Leung, “The minimal failure-causing schema of combinatorial testing,” *ACM Trans. Softw. Eng. Methodol.*, vol. 20, no. 4, sep 2011. [Online]. Available: https://doi.org/10.1145/2000799.2000801

[14] X. Niu, C. Nie, Y. Lei, and A. T. Chan, “Identifying failure-inducing combinations using tuple relationship,” in *2013 IEEE Sixth International Conference on Software Testing, Verification and Validation Workshops*, 2013, pp. 271–280.

[15] X. Zhang, N. Gupta, and R. Gupta, “Locating faults through automated predicate switching,” in *Proceedings of the 28th International Conference on Software Engineering*, ser. ICSE ’06. New York, NY, USA: Association for Computing Machinery, 2006, p. 272–281. [Online]. Available: https://doi.org/10.1145/1134285.1134324

[16] A. Griesmayer, S. Staber, and R. Bloem, “Fault localization using a model checker,” *Softw. Test. Verif. Reliab.*, vol. 20, no. 2, p. 149–173, jun 2010.

[17] ——, “Automated fault localization for c programs,” *Electronic Notes in Theoretical Computer Science*, vol. 174, no. 4, pp. 95–111, 2007, proceedings of the Workshop on Verification and Debugging (V&D 2006).

[18] A. Zeller, “Isolating cause-effect chains from computer programs,” in *Proceedings of the 10th ACM SIGSOFT Symposium on Foundations of Software Engineering*, ser. SIGSOFT ’02/FSE-10. New York, NY, USA: Association for Computing Machinery, 2002, p. 1–10. [Online]. Available: https://doi.org/10.1145/587051.587053

[19] A. Zeller and R. Hildebrandt, “Simplifying and isolating failure-inducing input,” *IEEE Transactions on Software Engineering*, vol. 28, no. 2, pp. 183–200, 2002.

[20] G. J. Myers, C. Sandler, and T. Badgett, *The art of software testing*. John Wiley & Sons, 2011.

[21] 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.