Towards Adaptable and Uncertainty-aware Behavior Trees

Mehran Rostamnia[†], Gianluca Filippone[†], Ricardo Caldas[*†], Patrizio Pelliccione[†]
[†]*Gran Sasso Science Institute*, L'Aquila, Italy – email: {firstname.lastname}@gssi.it
[*]*Chalmers and University of Gothenburg*, Gothenburg, Sweden – email: {firstname.lastname}@chalmers.se

***Abstract*—Space robotic missions are taken on a highly uncertain ground, yet require high autonomy. In space, events are unknown and their effects are hard to predict. Mission designers are forced to make decisions despite an inherent lack of information and this results in complex and stiff specifications. Stiffness flags for brittleness. Towards flexibility and modularity, Behavior Trees foster a tractable notation for reactive behavior, attracting the spotlight of robotic mission specifications. However, they lack support for taming uncertainty at runtime. This paper proposes a first step towards the extension of behavior trees with adaptability in order to deal with uncertainty. Our implementation extends the behavior trees constructs with adaptable nodes, i.e., nodes that can be hot-swapped at runtime. Our framework relies on quasi-natural language requirements modeling in FRETISH notation, with transformations to uncertainty-aware behavior trees and deployment to space robotics scenarios in the context of Space ROS. We showcase the use of our framework within the simulation of a NASA mission on Mars.**

***Index Terms*—Space Robotics, Uncertainty, Robotics Mission Specification, Behavior Trees, Self-Adaptation.**

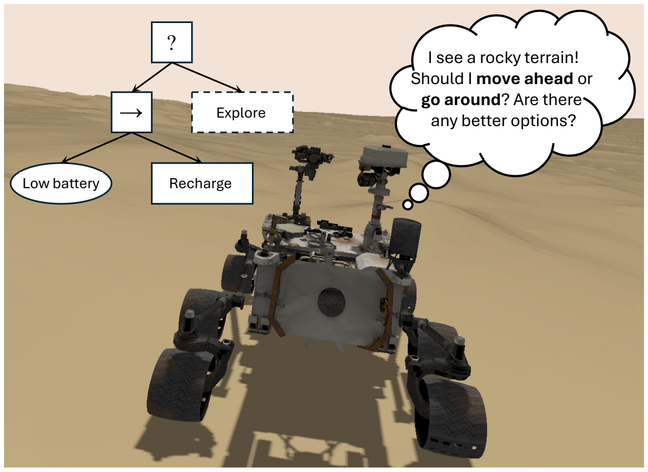
?
→
Explore
Low battery
Recharge
I see a rocky terrain! Should I **move ahead** or **go around**? Are there any better options?

Fig. 1: Adaptable Uncertainty-Aware Rover on Mars Mission

I. Introduction

Robotics has become instrumental to meeting economic and sustainable goals [1]. With robots increasingly deployed across various domains, software has become pivotal in enabling high autonomy and robustness in real-world robotic applications [2]–[4]. For instance, service-based software architectures have shown promising results in the development of space robots to be deployed on Mars [5]. However, the development of practices to high-autonomy robotics software is in the early stages; engineers lack sophisticated tools and methodologies needed to design and deploy systems that can reliably function in dynamic and unpredictable environments [3]. A key aspect is self-adaptation, allowing robots to maintain functionality despite runtime uncertainty [6], [7].

As the role of software in robotics grows, software engineering techniques are crucial across mission specification [8]–[10], architectural definition [11], [12], component design [13]–[15], and verification and validation [16]–[19]. In general, autonomy in robotics aims to reduce human intervention. Significant challenges remain to enable robots to adapt to real-world variability [4], [20], [21]. Resilient solutions are needed to address the openness and uncertainty of operational environments, necessitating adaptive capabilities.

Mission engineering approaches aim to simplify the design and deployment of robotic missions. Gil et al. [22] introduce a framework for specifying multi-robot missions with elements like objects, capabilities, and actions, emphasizing adaptability to real-world variability and task reusability. Garcia et al. [23], [24] propose a domain-specific language for generating and managing multi-robot missions. Rodrigues et al. [12] enable automated task reallocation in hierarchical plans. However, these approaches lack mechanisms to handle runtime uncertainty. This complicates mission specification and results in rigid designs; still, unpredictable changes at runtime hinder robotic mission engineers in taming uncertainty.

Robotic missions are taken on uncertain ground, events are unknown at design time, and the effects of such events are hard to predict. Mission designers, then, are forced to make decisions based on uncertainty when specifying the mission. This results in complex and stiff mission specifications. Behavior Trees (BTs) are known to enable the specification of reactive behavior [25], yet the support to express alternative behaviors to overcome runtime uncertainty is limited.

In this paper, we focus on managing robotic missions, particularly space exploration, by integrating user-defined requirements in FRETISH [26] with BTs. We enhance BTs with *adaptable nodes*, enabling runtime hot-swapping to build Uncertainty-aware Behavior Trees. This allows mission specifications to evolve during execution as new information becomes available. New nodes can be added to address emerging needs without redeploying the entire BT.

Figure 1 illustrates how our extension to BTs, i.e., Uncertainty-aware BT, can be used in practice in the context of NASA's Mars Science Laboratory (MSL) mission.

We take inspiration from the real challenges faced by NASA's Curiosity rover, which experienced an anomalous and premature degradation of the wheels. Ultimately, we showcase this scenario as an end-to-end process, from requirements as mission descriptions to deployment of the mission in the robot, and to mission re-definition and re-deployment.

The rest of the paper is as follows: A running example of the methodology is presented in Section II. An overview of the solution is presented in Section III. Section IV showcases adaptable and uncertainty-aware BTs in space robotics. Related works are reviewed in Section V. Finally, the article is concluded in Section VI.

II. Running and Motivating Example

In 2017, the images sent back to Earth by NASA's MSL Curiosity rover revealed that the wheels of the rover suffered anomalous and premature wear [27]. Sharp and wind-eroded pyramidal rocks that were on the route of the robot caused holes and tears in the rover's wheels, representing a serious concern regarding the rover's life expectancy. In order to solve this issue, MSL started an assessment of the damage and its causes to find measures aimed at minimizing the damage progression [28]. Interestingly, previous Mars rovers, whose wheel design was leveraged for the Curiosity rover, did not experience the same issue. The rocky terrain causing the wheels' unexpected wear was different from the one found in the previous landing sites and was not accounted for during testing. Curiosity's wheels were designed according to a very limited knowledge of the terrain that would eventually have been found on the rover's landing site on Mars. Moreover, the navigation algorithm employed was irrespective of the (uncertain) terrain surface, with wheels moving at a too-high speed on pointed rocks. Although the design and functionality of the rover had been extensively tested, the lack of precise knowledge of the actual terrain features represented an *uncertainty* that, before the mission started, could not be solved. As a solution to this problem, engineers provided a set of guidelines concerning the robot's navigation strategy over what they called "wheel-hostile terrains". These guidelines involve the reduction of the robot's speed, the minimization of turning movements, and driving backward over this kind of terrain to preserve the more damaged front wheels. However, their implementation required an update of the robot's software, while a new wheel design was developed for the subsequent Mars rovers.

Summarizing, the lack of knowledge about the Mars terrain represented an uncertainty that prevented the complete envisioning and evaluation of the possible runtime scenarios that might arise during mission execution, which eventually led to the design of multiple alternative behaviors required to address them. In general, accounting for uncertainty calls for the ability to envision and develop multiple strategies (i.e., alternative behaviors) to respond to the high variety of uncertain conditions that a robotic system may face during its operation [29], [30]. However, it is not always possible to foresee all the possible scenarios at design time due to the lack of knowledge of the actual operational environment [31]. In fact, as in the MSL's case reported above, new operating scenarios and related challenges could arise over time, while their solutions can not be known in advance.

For this reason, in many domains such as space exploration, robot's behavior specification is challenging since:

- It is impossible to anticipate all the possible runtime scenarios;
- It is impractical to specify all the alternative behaviors in a unique model (e.g., a unique BT);
- The behavior model should allow the specification of the condition leading to the execution of the proper behavior;
- The behavior model should be amenable to evolve and accommodate new behaviors.

In this paper, we propose a solution to the challenges listed above. To guide the reader through the description of the solution we present a simplified, yet informative example that considers an exploration task performed by a robot. In Section IV, we will return to the MSL-inspired example to showcase our approach at work. The robot has to move through the environment to get images; if its battery is low, the robot should go back to its base for charging. As for the Curiosity rover, the exploration task is affected by uncertainties since the navigation should account for the variety of terrains, hence, different navigation modes need to be defined. In particular, if the terrain is composed of sand, the robot can move at its maximum speed; if it is made of sharp rocks, the robot needs to reduce the speed to the minimum to avoid damage. However, as discussed above, the robot behavior model should consider the possibility of encountering additional surfaces (i.e., mud, cobblestones) that, at design time, cannot be foreseen and for which the most appropriate robot behavior is unknown.

III. Uncertainty-Aware Mission Specification

To tame uncertainty in the mission specification in response to the challenges discussed in the previous section, the mission specification must encode multiple alternative behaviors and the conditions associated with them to dynamically react to changing conditions. Moreover, it must be adaptable and amenable to be easily extended in response to new knowledge acquired during runtime. This can (i) allow changing the behavior of the robot under different conditions at runtime, (ii) avoid the hard-coding of all strategies into the mission at design time, and (iii) increase modularity and flexibility.

Figure 2 depicts our approach for realizing uncertainty-aware robotic missions. Key-enabler to embedding uncertainty in robotic missions specifications, our approach envisions the specification of robotic missions through an extended version of BTs, namely *Adaptable BT*. Using Adaptable BTs, mission designers can model missions while accounting for uncertainties by specifying a set of mission alternatives, modeled as separate BTs, that describe all possible behaviors to be executed when facing different conditions during mission execution. The conditions leading to the execution of a given alternative are specified using a FRETISH-based syntax. These three inputs (adaptable BT, alternatives, and conditions) are

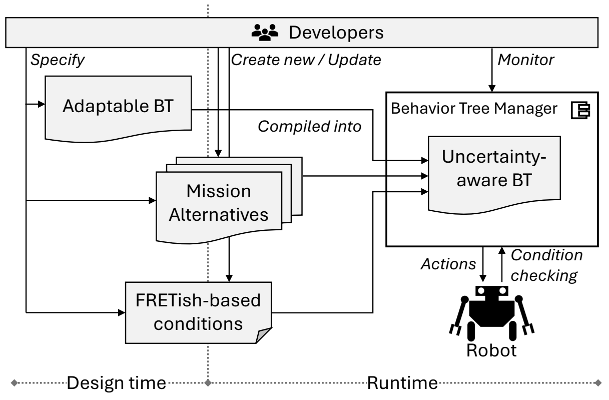

Fig. 2: Overview of our approach

then automatically compiled into a full BT (*uncertainty-aware BT*), which (i) integrates all the alternatives, (ii) owns the logic required for their selection, (iii) can be deployed on the robot and executed. In the following, we specify each of the elements that build our solution.

A. Mission specification through Adaptable BT

Adaptable BTs extend BTs with a novel leaf node, called *adaptable node* that allows the specification of an uncertainty-affected portion of the mission. Adaptable nodes model a point of uncertainty in the mission for which the actions performed by the robots cannot (or are inconvenient to) be fully specified during the mission design. In other words, adaptable nodes are used as a "placeholder" for a set of alternatives that specify the possible robot's behaviors at runtime, in response to the (uncertain) runtime conditions. Adaptable nodes are useful to model and manage the *known unknowns*. In fact, before mission execution, the mission specifier should know that a specific part of the mission is subject to uncertainty and can specify such behavior through adaptable nodes.

Adaptable nodes are abstract, meaning that unlike action and condition leaf nodes, adaptable nodes are not executable, i.e., they cannot be *ticked*, since their behavior is not defined (do not return *Failure*, *Running*, or *Success* like other leaf nodes). Graphically, adaptable nodes are represented with dashed boxes. Figure 3 shows the adaptable BT built to model the running example described in Section II. The BT features a condition node (the ellipses with label *low_battery*), an action node (the solid box with label *nav_to_base*), and the novel adaptable node (the dashed box with label *exploration*). The adaptable node specifies the mission portion affected by uncertainty, for which a unique behavior cannot be modeled beforehand; rather, multiple alternatives, describing the behavior required whenever the robot is moving on the sand or on the rocks, are required. It is important to note that some alternative behaviors can be defined before mission execution. However, additional alternative behaviors can be specified and added during mission execution. See Section III-D for a description on how the addition of mission alternatives at runtime works.

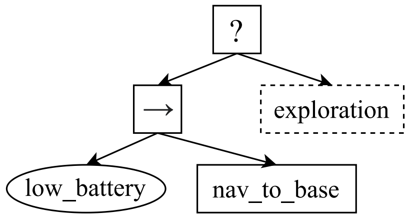

Fig. 3: Graphical representation of an adaptable task

The alternative behaviors for an adaptable node are defined as fully specified BTs, externally provided as separate models, that encode the different strategies or actions to be executed according to the possibly different and changing runtime conditions. Each alternative represents a specific approach to achieve a specific task, hence providing the robot with multiple options that can be chosen at runtime.

To make the mission executable, adaptable BTs need to be compiled into a full BT that embeds the alternatives defined for the uncertain mission portions and their selection logic. As we will detail later, the full BT is generated by replacing the adaptable BT's adaptable nodes with a suitably built subtree containing the set of specified alternatives and the proper selection logic.

B. FRETISH-based conditions specification

The specification of the conditions leading to the execution of the mission alternative is provided through the FRETISH syntax. FRETISH is a structured natural language developed by NASA to elicit system requirements, allowing their unambiguous specification and formalization [26], [32]. Our approach leverages FRETISH due to its structured yet simple syntax, which facilitates machine parsing and processing, hence making it well-suited for unambiguously specifying the conditions for mission alternatives. A FRETISH requirement is composed of six fields: *scope*, *condition*, *component*, *"shall"*, *timing*, and *response*. The *scope* field specifies the interval in which the requirement holds; the field *condition* is a Boolean expression that triggers the need for a response; the *component* defines the component targeted by the requirement; *shall* is a keyword stating that the component behavior must conform with the requirement; *timing* is an optional field that specifies when the response is expected; *response* is a Boolean expression that specifies the conditions that must be satisfied by the component.

In our approach, we consider the conditions for the selection of the mission alternatives as requirements that must hold when their related adaptable node should be executed.[1] Thus, developers can leverage the FRETISH structured syntax to specify these requirements, without the need for directly coding the subtree that checks the conditions and realizes the alternative's choice within the mission BTs. In particular, we employ a subset of the FRETISH grammar to specify

[1]Although the adaptable node is not executable, we write "the execution of the adaptable node" to refer to the execution of the portion of the mission represented by this node, for which alternatives are defined.

these conditions, as described in the following. For each adaptation alternative provided within the mission, a FRETISH requirement must be specified. By leveraging the semantics of the FRETISH grammar, each requirement describes, through the *scope* field, the name of the adaptable node during whose execution the requirement must hold. The *condition* field specifies the conditions enabling the related alternative as a Boolean expression. The *component* field defines the system component targeted by the requirement; in our approach, this component is always the mission. Finally, the *response* field specifies which is the alternative that should be executed when the defined conditions hold. The response is specified as a Boolean expression comparing the name of the alternative with the *selected_alt* variable. In summary, the specified conditions always have the following structure: *"in <adaptable_node> if <condition> mission shall satisfy selected_alt=<alt_name>"*. The definition of the grammar of the FRETISH language is available online within the replication package of this work.[2] Table I shows the FRETISH requirement within the FRET tool[3] related to our running example. By following the example description, the requirement states that in the scope of the *exploration* adaptable node, if the robot is on a sandy terrain, the alternative to be executed should be the one named "sand_alternative", which involves the maximum speed movement of the robot.

[SCOPE]	[CONDITIONS]	[COMPONENTS*]	SHALL*	[TIMING]	[RESPONSE*]
In Exploration	IF sand_terrain	Mission	SHALL		SATISFY selected_alt=sand_alternative

TABLE I: Requirement in FRETISH syntax.

C. Uncertainty-aware BT generation

Since adaptable nodes cannot be executed (i.e., they do not have a defined returned value when ticked), the mission specified through the adaptable BT cannot be executed as well. To allow the execution of the mission and the runtime selection of the alternatives, a complete BT is required (*uncertainty-aware BT* in Figure 2). This tree must own the defined alternatives and it must encode the conditions needed for their runtime selection. To generate the uncertainty-aware BT, the following two steps are needed: (i) FRETISH requirements have to be translated into fragments of BT that perform the condition checking (from hereon, we will refer to these fragments as "condition-checking subtrees"); (ii) all the condition-checking subtrees and their corresponding alternatives need to be suitably integrated into the adaptable BT and put in place of the adaptable node.

Concerning the creation of the condition-checking subtrees, they are generated for each FRETISH requirement, i.e., a condition-checking subtree is generated for each alternative. These subtrees are generated according to the Boolean expression in the *condition* field of the requirement (see Section III-B). When ticked, they must return *Success* if the condition expressed through the Boolean expression holds and *Failure* otherwise. To build a condition-checking subtree, the Boolean expression of the related FRETISH requirement is first parsed into a tree where: (i) inner nodes are the AND, OR, and NOT operators and (ii) leaves are the checked conditions. Then, the subtree is built according to Algorithm 1, which is run from the root of the Boolean expression tree, as follows:

- If the current node is a leaf node (i.e., it is the label of the checked condition), return a *condition node* with the same label (lines 2-5);
- If the current node is a NOT operator, create a new *inverter node*, add the result of the algorithm execution on the operator's node child as a child of the new node, and return it (lines 6-9);
- If the current node is an OR operator (resp. AND), create a new *fallback node* (resp. *sequence node*), add the result of the algorithm execution on the operator's left and right children as children of the new node, and return it (lines 10-16 and 17-23).

Algorithm 1 Condition-checking subtree generation algorithm

```
 1: function GENSUBTREE($node$)
 2:     if $node$ is a leaf then
 3:         create a new $condition\_node$
 4:         $condition\_node.label \leftarrow node.label$
 5:         return $condition\_node$
 6:     else if $node$ is "NOT" then
 7:         create a new $inverter\_node$
 8:         $inverter\_node \leftarrow$ GENSUBTREE($node.child$)
 9:         return $inverter\_node$
10:     else if $node$ is "OR" then
11:         create a new $fallback\_node$
12:         $children \leftarrow$ [ ]
13:         APPEND($children$, GENSUBTREE($node.left$))
14:         APPEND($children$, GENSUBTREE($node.right$))
15:         $fallback\_node.children \leftarrow children$
16:         return $fallback\_node$
17:     else if $node$ is "AND" then
18:         create a new $sequence\_node$
19:         $children \leftarrow$ [ ]
20:         APPEND($children$, GENSUBTREE($node.left$))
21:         APPEND($children$, GENSUBTREE($node.right$))
22:         $sequence\_node.children \leftarrow children$
23:         return $sequence\_node$
```

After the condition-checking subtree is generated, it is connected with the related alternative using a *sequence* node by keeping the condition-checking subtree as the first child. This structure ensures that, when the tree is ticked, the tick is propagated to the alternative only if the condition-checking subtree returns success (i.e., only if the conditions enabling the alternative execution hold). The subtrees resulting from this process are connected through a *fallback* node; then, only one alternative is executed. Note that, for the potential overlap in conditions among multiple alternatives, this structure prioritizes the alternatives according to their order (i.e., the first left alternative with successful condition-checking is selected). Figure 4 shows the structure of the tree resulting from the

[2]https://github.com/RoboChor/towards-uncertainty-aware-bts
[3]https://github.com/NASA-SW-VnV/fret

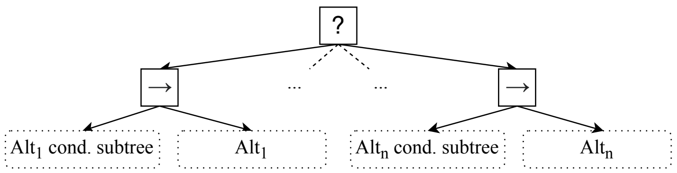

Fig. 4: Structure of the alternative selection and execution subtree

described process. This tree is substituted to the adaptable node in the adaptable mission BT.

The BT resulting from this process encodes the whole mission and owns the decision logic required for selecting the right alternative to be executed according to the FRETISH requirements specified by the developer. Differently from the adaptable BT described in Section III-A, this tree can be executed since it only contains *condition* and *action* node leaves, whose behavior is defined [25].

D. *Adaptable BT execution*

The uncertainty-aware BT is deployed and controlled by a BT management component at runtime (*Behavior Tree Manager* in Figure 2). Upon the discovery of new conditions requiring a different behavior, a mission alternative can be defined. A new version of the uncertainty-aware BT is then generated and provided to the behavior tree manager. Thanks to the reactive nature of BTs, whose nodes are continuously ticked, the new alternative will be available as soon as a new tick is sent from the tree's root without stopping the mission: once the new specified conditions are evaluated, the newly associated alternative subtree is executed, if required by the mission's runtime state. This allows the continuous refinement of the BT in order to cope with unexpected situations that require a different and alternative strategy, like the case of the NASA motivating example.

IV. Adaptable and Uncertainty-aware Behavior Trees in Practice

In this section, we demonstrate our approach in the space domain using the MSL Curiosity rover example from Section II. The rover collects Mars soil rock samples by moving to a designated location, deploying its tool arm to drill the rock, and depositing the powderized sample in a safe case. It also takes pictures before drilling and after completing the collection to document the process. As discussed in Section II, the described mission is affected by uncertainty due to the lack of knowledge about the actual conditions of the terrain upon which the rover has to move. In fact, different terrains like dust or rocks require different navigation strategies in order to avoid wheel damage. The specification of the mission has to consider this uncertainty and has to allow the definition of multiple navigation strategies. In the following, we detail how the mission is modeled while accounting for this uncertainty according to our approach.

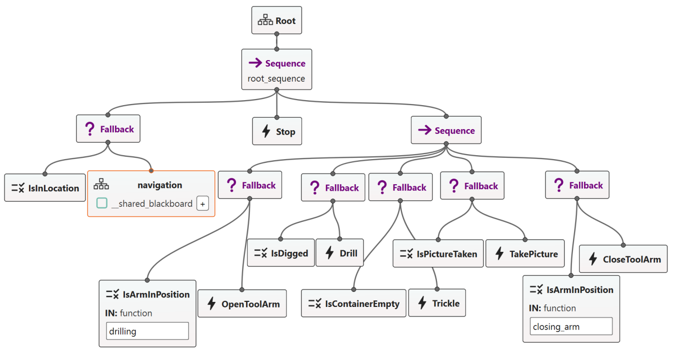

Fig. 5: Adaptable BT for the MSL Curiosity scenario

A. *Mission specification through Adaptable BT*

Being affected by uncertainty on the navigation, the mission is modeled through an Adaptable BT that uses an Adaptable node to model the navigation. Figure 5 shows the adaptable BT modeled for the mission using Groot2[4]. Alongside the actions nodes (e.g., *Stop*, *OpenToolArm*, *Drill*, etc.) and the condition nodes (e.g., *IsInLocation*, *IsArmInPosition*, *IsDigged*, etc.) modeling the "fixed" portions of the mission, the adaptable node is employed to model the uncertainty-related portion of the mission (see the highlighted *navigation* node). By leveraging the BehaviorTree.CPP XML syntax, we model this node by using the <Adaptable> tag, which we reserve for the specification of adaptable nodes.

According to the guidelines for the robot's navigation mentioned in Section II, we developed two alternative behaviors to realize the navigation on different terrain conditions. Figure 6 and Figure 7 show the BTs realized to model the mission alternatives. When on dust terrain, the robot moves forward at its "normal" speed: the robot has to turn toward the target location and drive forward (Figure 6). When on rocky terrain, as prescribed by the guidelines, the robot has to turn around (after checking the safest direction) and then move backward at a low speed toward the target location (Figure 7).

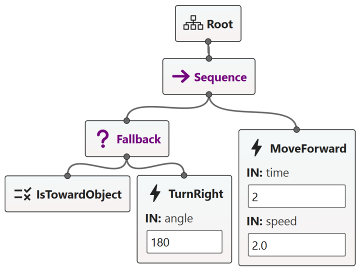

Fig. 6: *on_dust* mission alternative

To allow the execution of the correct alternative, we specified, for each of the two alternatives shown above, the requirements using the FRETISH language, as in Table II. The first requirement (**Req**$_1$) states that, if the condition *OnDust* holds, the alternative to be executed is the one named *on_dust*

[4]https://www.behaviortree.dev/groot

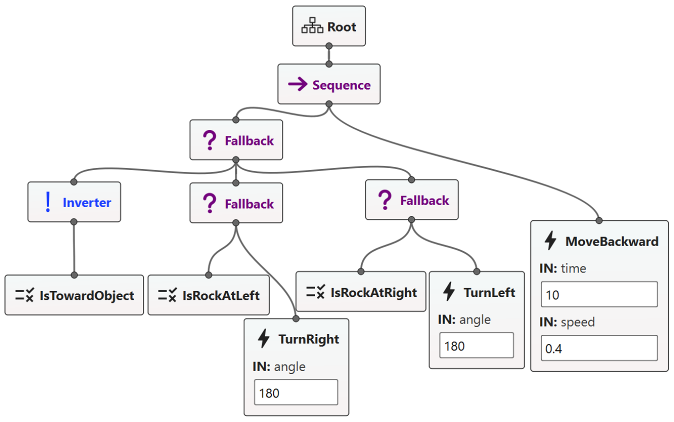

Fig. 7: *on_rocks* mission alternative

(i.e., the navigation on dusty terrain). The second requirement (**Req**$_2$) demands that the alternative *on_rocks* (navigation on rocks) has to be executed when *OnRocks* holds.

We modeled the two alternatives based on our design-time knowledge of possible terrain conditions. As discussed, additional BTs for robot navigation can be introduced during mission execution, if new terrain characteristics are discovered. This does not require updating the adaptable BT.

[SCOPE] [CONDITIONS] [COMPONENTS*] SHALL* [TIMING] [RESPONSE*]

(**Req**$_1$) In navigation IF OnDust Mission SHALL SATISFY selected_alt=on_dust
(**Req**$_2$) In navigation IF OnRock Mission SHALL SATISFY selected_alt=on_rocks

TABLE II: FRETISH requirements for *navigation* node.

B. Uncertainty-aware BT generation

As described in Section III-C, the mission's adaptable BT, the alternatives, and their associated requirements are compiled into a full BT that can be executed by the robot. We implemented the BT generator in such a way that the requirements are translated into subtrees that check the conditions required for the execution of the alternatives, according to Algorithm 1. The uncertainty-aware BT is then completed by substituting the adaptable node to realize the structure shown in Figure 4. In our scenario, the two condition-checking subtrees are built of the only conditions nodes *OnDust* and *OnRocks*, respectively. Each of these subtrees is then connected through a <Sequence> node with its related alternative. Finally, the *navigation* subtree is composed by connecting the resulting subtrees through a <Fallback> node. This is substituted to the <Adaptable> node to form the uncertainty-aware BT that can be executed by the Behavior Tree Manager.

C. Mission execution

We ran our scenario on the Mars Rover demo provided within the Space ROS demos Github repository[5]. We implemented the Behavior Tree Manager using the BehaviorTree.CPP library[6] to parse the BT and control the robot accordingly. The Behavior Tree Manager implements the action nodes specified in the mission model by sending the appropriate commands to the robot to control its behavior. Moreover, to evaluate the condition nodes, we integrated a condition manager to both keep track of the robot's status and simulate different terrains types.

Figure 8 shows some screenshots from the mission execution in the Gazebo simulation environment[7]. The screenshots show the main phases of the mission: the robot starts the mission (Figure 8a) and approaches the location according to the terrain conditions (screenshots refer to the dusty terrain, Figure 8b). When received at the location the robot takes a picture, drills the rock, and trickles the sample inside (Figure 8c). Finally, the mission is finished (Figure 8d).

The adaptable BT of the mission with alternatives and FRETISH requirements, the uncertainty-aware BT generator, its output, and the Behavior Tree Manager developed for this work are publicly available on Github[8].

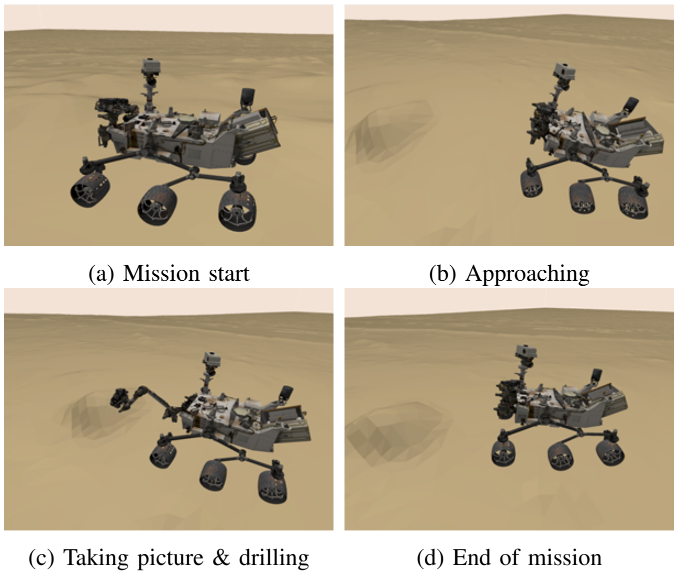
(a) Mission start (b) Approaching (c) Taking picture & drilling (d) End of mission

Fig. 8: Screenshots from the Mars Rover mission execution

V. Related work

In this section, we compare our uncertainty-aware BTs to relatable approaches that leverage uncertainty representation in robotics, focusing on dynamicity in BTs, specification relaxation techniques for runtime adaptation, and formalisms for uncertainty-aware planning in robotics.

A. Dynamicity in Behavior Trees

BTs gained traction as a formalism to specifying robotics behavior due to their flexibility and modularity. Robotics behavior asks for rich reactions to environmental changes

[5]https://github.com/space-ros/demos

[6]https://www.behaviortree.dev/
[7]https://gazebosim.org/home/
[8]https://github.com/RoboChor/towards-uncertainty-aware-bts

and BTs facilitate such modeling in comparison with the traditionally used state machines [15], [33]. A recent study on that mined behavior-tree Domain Specific Languages (DSLs) from GitHub found that dynamicity is typically achieved through runtime model modification, e.g. changing nodes at runtime, offering more flexibility to runtime modifications in comparison with activity diagrams and state diagrams [34]. Both BehaviorTree.CPP and py_trees[9] expose aspects of dynamic languages; they are interpreted and allow for creating new node types and modifying the shape of the syntax tree at runtime. We rely on BehaviorTree.CPP to implement runtime adaptation, though, differently from the native language we enhance BT modeling with a new type of leaf node.

B. Relaxing Specifications for Runtime Adaptation

The lack of information about runtime conditions presents a significant challenge in robotics software engineering [3]. We draw on self-adaptive systems literature to compare approaches for handling uncertainty [35]. Rigid behavioral specifications often hinder runtime adaptation by forcing developers to fully specify behaviors at design time. In response, some works introduce flexible constructs that enable underspecified behaviors, allowing decisions to be made at runtime.

For instance, in [36] the authors discuss relaxing specifications by using flexible operators (e.g., MAY, AS CLOSE AS POSSIBLE TO) and uncertainty annotations (e.g., environmental, behavioral) in modeling. These operators, based on fuzzy logic, have shown promise in adaptive robotic systems (aerial and terrestrial) implemented in the Anunnaki framework, which uses uncertainty-aware requirements specification to enhance flexibility under uncertainty [37]. Similarly, Solano et al. [38] apply uncertainty annotations in goal models, categorizing uncertainties related to the system, goals, and environment, including non-deterministic behaviors. This approach embeds uncertainty through probabilistic quality attributes and introduces a dedicated node for runtime decision-making, enabling models to adapt to environmental changes. Filippone et al. [39] introduce *adaptable tasks* within Hierarchical Task Networks (HTNs) to specify mission segments affected by uncertainty. Similar to our approach, alternatives are defined as separate models selected at runtime through *trigger functions* executed by the robot. However, this requires an ad hoc mission controller outside of the main mission execution.

We use BTs for mission specification, which, in comparison with goal models or HTNs, is closer to the execution semantics. Our approach uses FRETISH specifications to define conditions for executing alternatives, generating an uncertainty-aware BT that embeds both condition-checking and selection, without requiring extra functionality in the behavior tree executor.

C. Other Uncertainty-Aware Formalisms in Robotics

Another line of work addresses runtime uncertainty through automated learning for motion and task planning, where uncertainty is a core consideration. Robot actions are modeled using Markov Decision Processes (MDPs), which allow non-deterministic actions in partially observable environments [40]. These MDPs enable researchers to train for uncertainty-aware behavior using reinforcement learning [41], [42] or deep learning [43]. However, MDPs represent uncertainty as probabilistic annotations in state-machine-like models, which lack flexibility and modularity when compared to BTs and lack adaptable task richness leveraged in our work.

[9]https://py-trees.readthedocs.io/en/devel/

VI. Conclusion and Future Work

We presented an approach to systematically handle uncertainty in robotic mission specifications using BTs. We extended BT constructs with an adaptable node, which enables runtime modification of subtrees that define alternatives and conditions, and use FRETISH to specify requirements. We showed our approach in a scenario inspired by NASA's Mars Science Laboratory mission, addressing a real issue with the Curiosity rover's wheels. In our example, we assumed that the robot is able to recognize the type of terrain. In practice, this can be achieved through ML-based approaches that classify the terrain images coming from the robot's cameras. Furthermore, we assumed that finding the mission portion where adaptable nodes are required is done manually by designers according to their knowledge. However, in the future, this can be aided by integrating ML-based approaches or a feedback loop to find and provide feedback about critical mission portions subject to uncertainties.

Future work includes validating our framework in diverse contexts with more complex missions involving multiple uncertainty sources. We also aim to optimize the alternative selection process to improve resource usage and implement smart monitoring for detecting changes efficiently at runtime.

Acknowledgment

The authors of this paper would like to acknowledge Matei Schiopu, from Chalmers University and Gothenburg University, for his valuable contribution in running the simulation. This work has been (partially) funded by (i) the MUR (Italy) – PRIN PNRR 2022 project "RoboChor: Robot Choreography" (grant P2022RSW5W), (ii) the European Union - NextGenerationEU under the Italian Ministry of University and Research (MUR) National Innovation Ecosystem grant ECS00000041 - VITALITY – CUP: D13C21000430001, (iii) The MUR (PNRR) and ASI "Space it up" project, and (iv) the European HORIZON-KDT-JU-2023-2-RIA research project MATISSE "Model-based engineering of Digital Twins for early verification and validation of Industrial Systems" (grant 101140216-2, KDT232RIA_00017). This work is also supported by the Wallenberg AI, Autonomous Systems and Software Program (WASP) funded by the Knut and Alice Wallenberg Foundation.

References

[1] S. Guenat, P. Purnell, Z. G. Davies, M. Nawrath, L. C. Stringer, G. R. Babu, M. Balasubramanian, E. E. Ballantyne, B. K. Bylappa, B. Chen *et al.*, "Meeting sustainable development goals via robotics and autonomous systems," *Nature communications*, vol. 13, no. 1, p. 3559, 2022.

[2] D. Brugali and E. Prassler, "Software engineering for robotics [from the guest editors]," *IEEE Robotics & Automation Magazine*, vol. 16, no. 1, 2009.

[3] S. García, D. Strüber, D. Brugali, T. Berger, and P. Pelliccione, "Robotics software engineering: A perspective from the service robotics domain," in *Proceedings of the 28th ACM Joint Meeting on European Software Engineering Conference and Symposium on the Foundations of Software Engineering*, 2020, pp. 593–604.

[4] S. García, D. Strüber, D. Brugali, A. Di Fava, P. Pelliccione, and T. Berger, "Software variability in service robotics," *Empirical Softw. Engg.*, vol. 28, no. 2, Dec. 2022.

[5] L. Flückiger and H. Utz, "Service oriented robotic architecture for space robotics: Design, testing, and lessons learned," *Journal of Field Robotics*, vol. 31, no. 1, pp. 176–191, 2014.

[6] J. Cámara, B. Schmerl *et al.*, "Software architecture and task plan co-adaptation for mobile service robots," in *Proceedings of the IEEE/ACM 15th International Symposium on Software Engineering for Adaptive and Self-Managing Systems*, 2020.

[7] M. Askarpour, C. Tsigkanos *et al.*, "Robomax: Robotic mission adaptation exemplars," in *2021 International Symposium on Software Engineering for Adaptive and Self-Managing Systems (SEAMS)*. IEEE, 2021.

[8] S. Dragule, S. G. Gonzalo *et al.*, "Languages for specifying missions of robotic applications," *Software Engineering for Robotics*, 2021.

[9] C. Menghi, C. Tsigkanos *et al.*, "Mission specification patterns for mobile robots: Providing support for quantitative properties," *IEEE Transactions on Software Engineering*, 2022.

[10] C. Menghi, C. Tsigkanos, P. Pelliccione, C. Ghezzi, and T. Berger, "Specification patterns for robotic missions," *IEEE Transactions on Software Engineering*, vol. 47, no. 10, pp. 2208–2224, 2021.

[11] D. Kortenkamp, R. Simmons *et al.*, "Robotic systems architectures and programming," *Springer handbook of robotics*, 2016.

[12] G. Rodrigues, R. Caldas *et al.*, "An architecture for mission coordination of heterogeneous robots," *Journal of Systems and Software*, vol. 191, 2022.

[13] D. Brugali and P. Scandurra, "Component-based robotic engineering (part i) [tutorial]," *IEEE Robotics & Automation Magazine*, vol. 16, no. 4, 2009.

[14] D. Brugali and A. Shakhimardanov, "Component-based robotic engineering (part ii)," *IEEE Robotics & Automation Magazine*, vol. 17, no. 1, 2010.

[15] R. Ghzouli, T. Berger, E. B. Johnsen, A. Wasowski, and S. Dragule, "Behavior trees and state machines in robotics applications," *IEEE Transactions on Software Engineering*, vol. 49, no. 9, pp. 4243–4267, 2023.

[16] A. Gotlieb, D. Marijan *et al.*, "Testing industrial robotic systems: A new battlefield!" *Software Engineering for Robotics*, 2021.

[17] A. Cavalcanti, W. Barnett *et al.*, "Robostar technology: A roboticist's toolbox for combined proof, simulation, and testing," *Software Engineering for Robotics*, 2021.

[18] F. Ingrand, "Verification of autonomous robots: A roboticist's bottom-up approach," *Software engineering for robotics*, 2021.

[19] R. Caldas, J. A. P. García, M. Schiopu, P. Pelliccione, G. Rodrigues, and T. Berger, "Runtime verification and field-based testing for ros-based robotic systems," *IEEE Transactions on Software Engineering*, 2024.

[20] D. Bozhinoski, D. Di Ruscio, I. Malavolta, P. Pelliccione, and I. Crnkovic, "Safety for mobile robotic systems: A systematic mapping study from a software engineering perspective," *Journal of Systems and Software*, vol. 151, pp. 150–179, 2019.

[21] S. Peldszus, D. Brugali, D. Struber, P. Pelliccione, and T. Berger, "Software reconfiguration in robotics." *Empirical Software Engineering journal, To appear*, 2024.

[22] E. B. Gil, G. N. Rodrigues *et al.*, "Mission specification and decomposition for multi-robot systems," *Robotics and Autonomous Systems*, vol. 163, 2023.

[23] S. García, P. Pelliccione, C. Menghi, T. Berger, and T. Bures, "High-level mission specification for multiple robots," in *Proceedings of the 12th ACM SIGPLAN international conference on software language engineering*, 2019, pp. 127–140.

[24] S. Garcia, P. Pelliccione, C. Menghi, T. Berger, and T. Bures, "Promise: High-level mission specification for multiple robots," in *Proceedings of the ACM/IEEE 42nd International Conference on Software Engineering: Companion Proceedings*, 2020, pp. 5–8.

[25] M. Colledanchise and P. Ögren, *Behavior trees in robotics and AI: An introduction*. CRC Press, 2018.

[26] D. Giannakopoulou, T. Pressburger, A. Mavridou, and J. Schumann, "Automated formalization of structured natural language requirements," *Information and Software Technology*, vol. 137, p. 106590, 2021.

[27] NASA, "Premature wear of the msl wheels," https://llis.nasa.gov/lesson/22401, 2017, accessed: (November 8th, 2024).

[28] A. Rankin, N. Patel, E. Graser, J.-K. F. Wang, and K. Rink, "Assessing mars curiosity rover wheel damage," in *2022 IEEE Aerospace Conference (AERO)*, 2022, pp. 1–19.

[29] L. Baresi and C. Ghezzi, "The disappearing boundary between development-time and run-time," in *Proceedings of the FSE/SDP Workshop on Future of Software Engineering Research*, ser. FoSER '10. New York, NY, USA: Association for Computing Machinery, 2010, p. 17–22.

[30] S. Mahdavi-Hezavehi, P. Avgeriou, and D. Weyns, "Chapter 3 - a classification framework of uncertainty in architecture-based self-adaptive systems with multiple quality requirements," in *Managing Trade-Offs in Adaptable Software Architectures*, I. Mistrik, N. Ali, R. Kazman, J. Grundy, and B. Schmerl, Eds. Boston: Morgan Kaufmann, 2017, pp. 45–77.

[31] J. Andersson, L. Baresi, N. Bencomo, R. de Lemos, A. Gorla, P. Inverardi, and T. Vogel, *Software Engineering Processes for Self-Adaptive Systems*. Springer Berlin Heidelberg, 2013, pp. 51–75.

[32] E. Conrad, L. Titolo, D. Giannakopoulou, T. Pressburger, and A. Dutle, "A compositional proof framework for fretish requirements," in *Proceedings of the 11th ACM SIGPLAN International Conference on Certified Programs and Proofs*, 2022, pp. 68–81.

[33] M. Iovino, E. Scukins, J. Styrud, P. Ögren, and C. Smith, "A survey of behavior trees in robotics and ai," *Robotics and Autonomous Systems*, vol. 154, p. 104096, 2022.

[34] 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*, 2020, pp. 196–209.

[35] S. Mahdavi-Hezavehi, P. Avgeriou, and D. Weyns, "A classification framework of uncertainty in architecture-based self-adaptive systems with multiple quality requirements," in *Managing Trade-Offs in Adaptable Software Architectures*. Elsevier, 2017, pp. 45–77.

[36] J. Whittle, P. Sawyer, N. Bencomo, B. H. Cheng, and J.-M. Bruel, "Relax: Incorporating uncertainty into the specification of self-adaptive systems," in *2009 17th IEEE International Requirements Engineering Conference*. IEEE, 2009, pp. 79–88.

[37] M. A. Langford, S. Zilberman, and B. Cheng, "Anunnaki: A modular framework for developing trusted artificial intelligence," *ACM Transactions on Autonomous and Adaptive Systems*, vol. 19, no. 3, pp. 1–34, 2024.

[38] G. Félix Solano, R. Diniz Caldas, G. Nunes Rodrigues, T. Vogel, and P. Pelliccione, "Taming uncertainty in the assurance process of self-adaptive systems: a goal-oriented approach," in *2019 IEEE/ACM 14th International Symposium on Software Engineering for Adaptive and Self-Managing Systems (SEAMS)*, 2019, pp. 89–99.

[39] G. Filippone, J. A. Piñera García, M. Autili, and P. Pelliccione, "Handling uncertainty in the specification of autonomous multi-robot systems through mission adaptation," in *Proceedings of the 19th International Symposium on Software Engineering for Adaptive and Self-Managing Systems*, 2024, pp. 25–36.

[40] J. Jiang, Y. Zhao, and S. Coogan, "Safe learning for uncertainty-aware planning via interval mdp abstraction," *IEEE Control Systems Letters*, vol. 6, pp. 2641–2646, 2022.

[41] C. Diehl, T. S. Sievernich, M. Krüger, F. Hoffmann, and T. Bertram, "Uncertainty-aware model-based offline reinforcement learning for automated driving," *IEEE Robotics and Automation Letters*, vol. 8, no. 2, pp. 1167–1174, 2023.

[42] C.-Y. Kuo, A. Schaarschmidt, Y. Cui, T. Asfour, and T. Matsubara, "Uncertainty-aware contact-safe model-based reinforcement learning," *IEEE Robotics and Automation Letters*, vol. 6, no. 2, pp. 3918–3925, 2021.

[43] L. González-Rodríguez and A. Plasencia-Salgueiro, "Uncertainty-aware autonomous mobile robot navigation with deep reinforcement learning," *Deep learning for unmanned systems*, pp. 225–257, 2021.