ReBeT: Architecture-based Self-adaptation of Robotic Systems through Behavior Trees

Elvin Alberts
Vrije Universiteit Amsterdam & Delft University of Technology
The Netherlands
e.g.alberts@{vu.nl, tudelft.nl}

Ilias Gerostathopoulos, Vincenzo Stoico, Patricia Lago
Vrije Universiteit Amsterdam
The Netherlands
{i.g.gerostathopoulos,v.stoico,p.lago}@vu.nl

Abstract—**Robotics software needs to be self-adaptive. Self-adaptation in robotics can, among others, take the form of changing a robot's task plan or its software architecture at runtime. The latter has shown to be effective in satisfying quality requirements such as minimizing energy consumption and operating safely. However, most self-adaptive robotic systems perform architecture-based self-adaptation to meet the functional goal of completing an assigned mission. Additionally, the mechanisms to accomplish architectural adaptations are mostly ad-hoc and not oriented towards reuse. We in turn investigate how quality requirements and architecture-based self-adaptation can be facilitated in robotics software while integrating into existing practices to promote practitioners' adoption and reuse. To this end, we design and implement an extension to the Behavior Trees (BTs) task plan formalism which introduces an explicit consideration of quality requirements. Additionally, we implement a general architectural adaptation layer for ROS2 systems and an extension to BTs which showcases its utilization. Finally, we perform quantitative experiments to evaluate the effectiveness of our approach in satisfying quality requirements via architectural adaptation on a mobile terrestrial robot. We find our approach to be an effective means to address a variety of self-adaptation scenarios within the mission of the system.**

Index Terms—**robotics, self-adaptation, behavior trees, quality**

I. Introduction

The world is seeing a consistent increase in the use of robotics across industries [1]. This entails an increase in demand for the capabilities of said robotic systems and their software. At the same time, robotics applications need to cope with the runtime uncertainty they face during their missions. This need comes both from the increasingly dynamic environments in which robots operate (*e.g.,* different terrains/lighting conditions/obstacles), from potentially unreliable software and hardware, and from the dynamic missions they undertake (*e.g.,* mission goals which change at runtime). A promising solution is to incorporate the ability to dynamically change their operation at runtime *i.e.,* self-adaptation [2], [3].

There is an established pattern for the use of self-adaptation in robotics when it comes to changing the task plan of a robot [4]–[7]. Particularly, plans are made *reactive* to enable autonomous behavior. For example, if the camera component in a robotics system fails at runtime, it adjusts its plan to navigate based on a saved map of the environment rather than the real-time visual feed. Crucially, these adaptations tend to be made for the sake of completing the overall 'mission' of the robot.

In contrast, research into self-adaptive systems outside of robotics has seen a focus on adaptation for meeting quality requirements (QRs) [8], [9]. For example, through the use of self-adaptation to ensure the availability of a website by scaling the number of servers providing it. Additionally, meeting QRs is often done by modifying the software architecture of the system (*e.g.,* by adding, removing, binding, re-configuring components). Since the seminal work by Garlan *et al.* [10] *architecture-based self-adaptation* has become a well-known approach in which architectural models are used as the basis for both the reasoning about and execution of self-adaptation. Among other advantages, this allows engineers to develop self-adaptation solutions that abstract away from particular applications or missions and can be potentially reused [9].

In our quest to apply architecture-based self-adaptation to robotics, we phrase two main research questions **RQ1: How can we integrate the consideration of quality requirements into existing practices of robotics software?** and **RQ2: How can we integrate architecture-based self-adaptation into existing practices of robotics software?** By integrating into existing practices we emphasize our aim of relying on existing tools and libraries (*e.g.,* Behavior Trees and ROS2) as opposed to requiring extensive re-modeling or re-implementation.

In this paper we propose a multi-faceted approach, **ReBeT**[1](**Re**-configuration with **Be**havior **T**rees), for building reusable architecture-based self-adaptation solutions in robotics software. We introduce a set of abstractions to allow (1) specifying QRs and monitoring their satisfaction to allow motivating either task plan or architectural adaptations (2) adapting the architecture of a ROS2 system irrespective of the planning paradigm used (3) integrating the adaptation of a system's software architecture within the task plan specification. While (1) addresses RQ1, (2) and (3) address RQ2.

We implement the above by relying on ROS2 [11] and extending Behavior Trees (BTs) [12], a formalism for planning the actions of an agent. BTs are an established task planning approach in robotics and are similar to hierarchical state machines [13]. One of the benefits of BTs is that their hierarchical structure allows us to capture the fact that multiple tasks

[1]Code available at: https://github.com/EGAlberts/ReBeT

may be subject to the same QR. Additionally, their popularity makes our approach accessible to robotics practitioners. Our approach is evaluated on FROG, a terrestrial mobile robot tasked with a realistic mission of locating and detecting a particular object. We apply ReBeT to FROG, implementing specific adaptations to meet its QRs. We also quantitatively evaluate the increase in overall utility brought by our self-adaptation solutions compared to a non-adaptive baseline.

The target audience of this work is robotics software developers and researchers. With ReBeT we aspire to support developers and researchers of robotics software in considering quality of service explicitly, and ensuring it at runtime through self-adaptation. We also aim to standardize architecture-based adaptations and make them accessible.

II. Background and Related Work

A. Background

1) ROS2: Robot Operating System 2 (ROS2) is a set of open-source libraries and tools that has become the de facto standard for building robotic applications [11]. From an architectural perspective, an application built with ROS2 consists of a number of software components – *nodes* – deployed to in part command a given robot. To communicate, nodes can act as publishers and subscribers to *topics*, clients and servers to *services*, or clients and servers to *actions* – the latter being intended to be used only by long-running tasks. Important for our work is that ROS2 includes 'managed' nodes called *lifecycle nodes*. These can be made to transition between a set of pre-defined states in their 'lifecycle' in accordance with the progress of their execution (*Create*, *Activate*, *Destroy*, etc.). For more details, we invite the reader to look at the official ROS2 documentation[2].

2) Behavior Trees: Behavior trees (BTs) are a popular formalism for specifying action plans in robotics [12], [13]. BTs are an alternative to hierarchical state machines where the main building block is an action instead of a state. A BT is graphically represented by a directed tree of nodes where outgoing nodes are parents and incoming nodes are children. Leaf nodes are *action* or *condition* nodes, *i.e.,* nodes that either execute tasks or evaluate expressions, while non-leaf nodes are either *control* nodes (can have many children) or *decorator* nodes (can have a single child only). A control node can be either a *sequence*, *parallel* or a *fallback* node. A sequence node prescribes the sequential execution of its children and a parallel their parallel execution, whereas a fallback selects only one of its children to be executed at any point in time, dependent on the results of the children. Finally, a decorator node may transform the result received from its child, terminate it, or repeat its processing, to mention a few common uses. The execution of a tree starts from the root node that sends an execution signal – *tick* – to its child. Ticks are propagated through the tree according to the control nodes. When an action node is ticked its execution begins – if not already running. Every node sends to its parent one of the following three state indicators: *running* (operation ongoing), *success* (goal reached), or *failure* (goal not reached). This combination of downward propagation of ticks and upward propagation of states allows for specifying and executing complex yet compactly represented plans.

[2]https://docs.ros.org/en/humble/

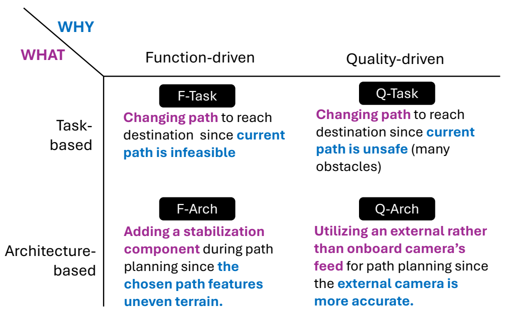
WHY
WHAT
Function-driven
Quality-driven
Task-based
Architecture-based
F-Task
Changing path to reach destination since current path is infeasible
Q-Task
Changing path to reach destination since current path is unsafe (many obstacles)
F-Arch
Adding a stabilization component during path planning since the chosen path features uneven terrain.
Q-Arch
Utilizing an external rather than onboard camera's feed for path planning since the external camera is more accurate.

Fig. 1: Analysis and Examples of Runtime Adaptations in Robotics

In our implementation, we extend BehaviorTree.CPP (BT.CPP)[3], a C++ BT library. Specifically, BT.CPP implements BT nodes as port-based objects. The ports of objects can be constrained to that specific object or specified as a reference to a value stored in a `Blackboard` conforming to the blackboard design pattern. BT.CPP uses XML to declare BTs. Each node is an XML element; the element's attributes are the nodes's input and output ports. To represent hierarchy, nodes below each other in the tree are enclosed by their parent element. A ROS2 library acts as a bridge between BT.CPP and ROS2[4], wherein BT nodes play the role of 'clients' for ROS2 nodes. In particular, a BT action node can enclose a service client, action client, or publisher/subscriber to a ROS2 topic.

3) Self-adaptation in Robotics: Self-adaptation applied in robotics is not novel in its own right. Approaches stemming from both research and practice exist to make a robotics system robust to runtime uncertainty via runtime change. Based on our prior analysis of the state of the art in the field [3], we categorize runtime adaptations that typically occur in robotic systems according to two criteria: (i) the reason for runtime adaptation *i.e., why* an adaptation takes place, and (ii) the nature of such adaptation *i.e., what* is being adapted at runtime (Fig. 1). For the former, we distinguish between *function-driven* and *quality-driven* adaptations. A function-driven adaptation is triggered in order to fulfil the goal of completing the assigned mission (reach a destination, complete a task). A quality-driven adaptation happens in order to keep satisfying a QR of the robotics system, *e.g.,* minimize energy consumption or navigate with maximum velocity while remaining safe. As to the nature of adaptations, we distinguish between *task-based* and *architecture-based* adaptations. In task-based adaptations, the task plan is modified at runtime, *e.g.,* in case of navigation, this might entail replanning to find an alternative route to a destination. In architecture-based adaptation, the architecture of the robotics system is being

[3]https://www.behaviortree.dev/
[4]https://github.com/BehaviorTree/BehaviorTree.ROS2/tree/humble

modified at runtime, *e.g.,* via adding or removing a component to the system, creating or dissolving connections between components, or re-configuring a component at runtime.

The combination of options for each criterion in our analysis form the four quadrants depicted in Fig. 1. A runtime adaptation in robotics typically falls into one of the quadrants. At the same time, an approach such as ours can support adaptations that span across quadrants. In particular, we rely on BTs, which already support F-Task adaptations primarily via incorporating fallback nodes in the design of the task plan. By explicitly modeling QRs in task plans, we support Q-Task adaptations. We also provide an interface to perform architectural adaptations. By using this interface in both function-driven and quality-driven adaptations we also effectively support F-Arch and Q-Arch, respectively.

B. Related Work

1) Relevance of BTs and architectural adaptations: The adoption of Behavior Trees by robotics practitioners is increasing, as confirmed by Ghzouli *et al.* [13]. They study five DSLs in the context of ROS-based software – two state-machine and three BT ones. The study aims at understanding how DSLs are engineered and used by practitioners in open-source ROS projects. Ghzouli *et al.* find that existing implementations of BTs are tightly coupled with the surrounding software system and their structure should be simple for enhancing understandability and reuse. Our extension of Behavior Trees keeps this simplicity, as the BTs of ReBeT involve only two new kinds of nodes. The work by Ghzouli *et al.* evidences that BTs are in widespread use and of interest for research, motivating our own choice for this particular planning formalism.

Peldszus *et al.* [14] perform a systematic literature review on software reconfiguration in robotics and the frameworks they utilize to enable this. This is relevant in that it represents the actuation of runtime adaptations. On the basis of their review (98 papers, 48 software artifacts) they determine among others the prevalence of certain types of adaptations. They find that, exceedingly, work in robotics focuses primarily on reparameterization of components and not other kinds of architectural adaptation. Through our own work, we hope to facilitate the increased usage of more types of adaptations, such adding or removing nodes and changing their connections.

2) Self-adaptation with BTs: Romero-Garcés *et al.* [15] provide a sophisticated model-based approach which employs quality attributes in robotics system through behavior trees for the purpose of self-adaptation. Their approach relies on a combination of neural networks and probabilistic models to transform perceived information from a variety of sources (the robot sensors, online data) to gauge the satisfaction of QRs, and then use this information for self-adaptation. The goal of their work is very close to our RQ1 *i.e.,* facilitating Q-Task adaptations. We still find the following key differences to our work. First, our work focuses much more heavily on architectural adaptation. In their work, they consider architectural adaptation only via parameter change, while our work also includes and demonstrates the removal and addition of components as well as the change in connections between components. Second, their work does not rely on standards within existing practices as they do not use ROS but a collection of disparate tooling with little detail as to their specific usage. Third, their work has a pervasive reliance on model-driven engineering, which entails the specification and modeling of the system, the adaptation scenario, and the context through which quality attributes are estimated. As mentioned in the introduction, while having merit, such requirements raise the barrier of entry for the adoption of their approach. With ReBeT, all necessary specification is contained within the confines of the BT. Finally, they accomplish their adaptations through modification of the structure of existing BTs, while our additions as their name suggest merely decorate existing BTs, leaving existing mission task plan specifications intact and not disrupting the evolution of BTs over time.

The approach of Segura-Muros *et al.* [6] sees the combination of AI planning languages and BTs to allow for runtime task plan adaptation. As in our approach, the BTs they use are integrated to work with ROS-based systems. The authors provide an architectural blueprint made of three macro-components: a Planner, a Blackboard, and an Executor. The Planner has a pre-fixed set of tasks and creates a plan out of them, while the Executor takes a plan and creates a set of actions to execute the plan represented by a BT. Crucially, they focus on F-Task adaptations *i.e.,* replanning tasks, without consideration of modifying software architecture. Additionally, there is no explicit consideration of QRs.

The work by Behery *et al.* sees BTs combined with mixed initiative planning to make them self-adaptive. To do so, similar to our own work, the authors introduce a new type of node which determines the sequence of execution of predefined subtrees at runtime. This is applied to a scenario where a robot arm is used for manufacturing and has to do so while working alongside a human safely yet quickly. This is another example of F-Task adaptation, as the self-adaptation that takes place is confined entirely to the BT itself. This is in contrast to our approach which allows for both task-based and architecture-based runtime adaptations. Additionally, while their approach does tackle quality concerns, these are not made explicit. Rather, these are encoded into models representing tasks.

3) Self-adaptation with other formalisms: In addition to BTs, there are approaches which consider other formalisms to realize self-adaptation in robotics. The approach of Jamshidi *et al.* [16] and its follow-up by Cámara *et al.* [17] use planning as model checking using PRISM. Through their approaches the two sets of authors realize Q-Arch within a ROS-based system as we do. Particularly in the work by Cámara *et al.* there is a focus on the co-adaptation of tasks and software architecture. They first adapt the architecture (Q-Arch) and then generate valid task plans for it (F-Task). In contrast, while our own work also supports Q-Arch and F-Task, there is no set sequence or hierarchy between the two. Additionally, as we introduce QRs independently of architectural adaptation, our approach also allows for Q-Task and F-Arch. Lastly, our approach has a lower adoption barrier as it does not rely on design-time models (in this case the input to PRISM).

Lastly, our previous work SUAVE [18], also uses ROS2 and employs self-adaptation to re-configure the software architecture of the robot through mode switching of components for fault handling, and reparameterization of a search component. The approach is applied to an underwater robot, which needs to locate an oil pipeline despite the uncertainty caused by components failing and poor visibility in the water. A fundamental difference is that SUAVE's adaptation logic is based on ontology-based reasoning, and that there is no explicit task planning formalism used. This work builds on the self-adaptation mechanisms of SUAVE through the addition of explicit task plans (BTs) and more types of architectural adaptation.

III. Approach - ReBeT

Our proposed approach aims to support building self-adaptation solutions in robotics software which integrate easily into existing practices. We have named our approach **ReBeT** which stands for **Re**-configuration with **Be**havior **T**rees. It consists of three contributions:

1) Quality Requirement Decorators (`QRDecorators`) – an extension to BTs allowing for quality-driven self-adaptation.
2) Architecture Adaptation Layer (`AAL`) – a ROS2 package for architecture-based self-adaptation.
3) Adaptation Decorators (`AdaptDecorators`) – an extension to BTs allowing for architecture-based self-adaptation through `AAL`.

`QRDecorators` make it so that QRs are treated as a first-class citizen when specifying a robotic task plan. Similar to goal-oriented approaches to self-adaptation [19], QRs provide a way to reason about *why* certain runtime adaptations should take place or be chosen over alternatives. The current degree of satisfaction of a QR can then be used to motivate adaptation decisions at runtime. We detail `QRDecorators` further in Section III-B.

`AAL` consolidates the built-in architectural adaptation capabilities of ROS2, allowing any ROS2-based system to easily have its software architecture manipulated at runtime. `AAL` is used alongside BTs in this paper but is fully functional independent of the task plan formalism used. We detail it further in Section III-C1.

To utilize `AAL`, we introduce `AdaptDecorators` which allows for baking architectural adaptations into the specification of a BT's task plan. `AdaptDecorators` can be provided an initial specification, which characterizes the architectural adaptations it performs – or have their specification provided at runtime through BT logic. We further detail `AdaptDecorators` in Section III-C2.

A. Overview

To provide an overview of our approach, we describe its operation as it fits the architecture model of a self-adaptive system. Here, we use a combination of the three-layer model by Kramer and Magee [20] in combination with the MAPE-K model [21] which is situated in the Change Management layer of the three-layer model as described by Weyns [9]. We illustrate how ReBeT fits within these two models in Fig. 2.

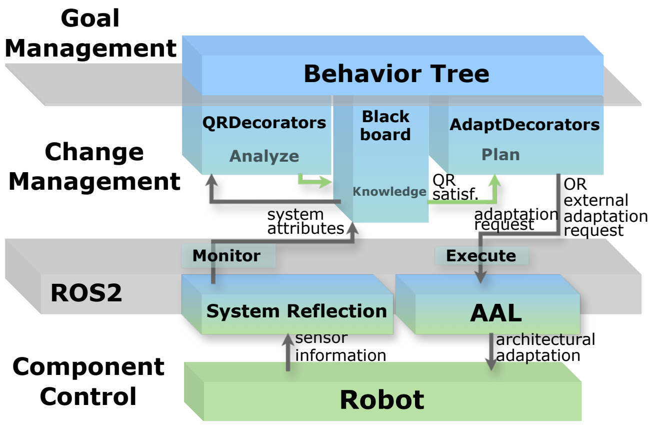

Fig. 2: Architecture Model of ReBeT

To aid our explanation we use a running example, FROG, which is also used in Section IV. FROG is a mobile terrestrial robot, specifically a Turtlebot3 Waffle, equipped with a camera and LIDAR sensor. FROG has been assigned the mission to map an unknown room with four obstacles in it, one of these obstacles being the target object that it has to detect and determine the location of.

The three-layer architecture model consists from the bottom up of a Component Control, Change Management and Goal Management layer.

The bottom layer, the Component Control Layer controls the system being made self-adaptive, *i.e.,* the managed system. With ReBeT this can be any system running ROS2. For the purposes of this paper, the lowest level, the robot's hardware, is simulated. Component Control has two interactions with the Change Management Layer above it. Firstly, the Component Layer provides the facility for the manipulation of the managed system. In our case, it manipulates the managed system's software architecture. We extend the facilities provided by ROS2 for architectural change with `AAL`. Secondly, Component Control forwards information regarding the status of the robot to the Change Management layer. To connect that information with the knowledge of the BT used for Change and Goal Management we provide a `SystemReflection` ROS2 component. For example, in FROG `SystemReflection` subscribes to the output of its odometry and LIDAR sensors, and places these in the blackboard of the BT.

Within the Change Management layer, the entirety of MAPE-K (*i.e.,* the managing system) loop takes place. This layer makes use of `AAL` to adapt the software architecture of the robot. `AAL` is usable with information analyzed by way of the `QRDecorators` (Q-Arch), or as specified by the task plan of the BT (F-Arch). As an example of Q-Arch, as FROG performs its mapping task, the size of the room is uncertain. Simultaneously, there is a QR in effect for FROG to keep its battery above a certain percentage. A self-adaptation plan can ensure this QR is satisfied at runtime, by enacting adaptations such as changing the robot's behavior to visit a charging dock or lowering its velocity to use less power while driving. As an example of F-Arch, consider the case of FROG transitioning from its mapping task to its detection task, FROG removes the ROS2 nodes it uses for mapping and adds those it requires for detecting objects from its camera feed. Irrespective of what the adaptations are driven by, both are realized by `AdaptDecorators`, the simple difference being that quality-driven `AdaptDecorators` contain plans conditional on the satisfaction of QRs, and function-driven by their placement within the task plan logic of the BT.

We now briefly describe how ReBeT follows the MAPE-K loop within the Change Management layer.

- ***Monitoring*** is performed by `SystemReflection` which monitors the status of the robot and how it perceives its environment stemming from Component Control.
- ***Analysis*** is performed by `QRDecorators`. Upon ticking the node(s) below it, each `QRDecorator` first retrieves the data necessary for determining its satisfaction and then places its output in the `Blackboard` through its metric and status ports. For example, when a `QRDecorator` that requires power consumption below 400W retrieves a consumption of 300W, it outputs a metric value of 0.25 (3/4 of the way to 0W) and status 'Power OK'.
- ***Planning*** happens in the `AdaptDecorators`. `AdaptDecorators` read information from the `Blackboard` placed there by `QRDecorators`. This information is provided as arguments to either (i) an internal adaptation plan or (ii) a utility function used for an external adaptation strategy. When using (i), the user provides adaptation logic directly in the implementation of an `AdaptDecorator` (*e.g.,* Listing 1). When using (ii), adaptation requests provide values of the utility function to a separate module encompassing the adaptation logic – we use a library of reinforcement learning algorithms [22] as such a module for our evaluation.
- ***Execution*** is performed by `AAL`. `AAL` simplifies the architectural adaptation of ROS2 nodes as it centralizes their manipulation. Without it, each `AdaptDecorator` would require the prior definition of a unique ROS2 client to every node it may wish to adapt at runtime. Instead, `AAL` provides two ROS2 services, one for executing the adaptations of internal adaptation plans and another for external adaptation strategies.
- ***Knowledge*** is stored in the `Blackboard` within the BT. The information shared between each of the four phases is stored in the `Blackboard` through output ports of nodes in the BT, or by the `SystemReflection` ROS2 node. Information is read from the BT through input ports of BT nodes.

The Goal Management layer is responsible for changing the MAPE loop that is active in the Change Management layer. It is important to realize that a task plan encoded in a BT typically has several paths ending in distinct tasks (action nodes). The execution of the task plan entails progressing through several paths, potentially in nonlinear fashion. Each path potentially sees the realization of a MAPE-K loop which uses the `QRDecorators` and the `AdaptDecorators` for analysis and planning, respectively. As the BT is traversed during execution its logic controls both which MAPE loop is currently active (concern of the Goal Management) and the execution of the MAPE loop itself (concern of the Change Management) – that is why it cuts across the two layers in Fig. 2.

B. Quality-driven Self-adaptation with Behavior Trees

To address RQ1 which concerns the integration of QRs into existing practices, we propose `QRDecorators`. They are pure extensions of BTs, with no inherent dependency on ROS2. The role of each `QRDecorator` is to continually determine the degree of satisfaction of a QR of the task it decorates. For example, the QR 'The robot should not consume more than 150 Watts per second' could have a metric which calculates the ratio between current power consumption in Watts and 150 Watts. Optionally it can also output a description of the 'status' of the QR, such as 'in violation' or 'partially fulfilled' to provide a semantic description. To follow the previous example, if power consumption were to exceed 150 Watts the status could indicate 'in violation'.

We provide two subclasses of `QRDecorators`, the `TaskLevelQR` for localized QRs and the `SystemLevelQR` for system-wide QRs. A `TaskLevelQR` is intended to represent requirements pertinent to a specific task a robot performs. For instance, it could represent the QR 'Proximity to objects must exceed 0.5m during movement'. A `SystemLevelQR` is intended for representing requirements pertinent to the whole system. For instance, it could represent our earlier example QR of not consuming more than 150 Watts per second for the entire robot. To establish a semantic and programmatic relationship between task-level and system-level QRs, each `QRDecorator` indicates which quality attribute (QA) it pertains to *e.g.,* safety or energy consumption. When a `SystemLevelQR` calculates its own degree of satisfaction, it can then optionally gather the satisfaction of each `TaskLevelQR` of the same QA active below it in the BT. For example, each task (navigation, mapping, detection) could have its own QR with the QA 'power', and all their metrics can then be combined to determine whether how many Watts per seconds are being consumed.

In order to calculate a metric, a `QRDecorator` requires information from the robot to be present in the `Blackboard`. We implement this as illustrated in Fig. 3, the `SystemReflection` ROS2 node subscribes to and consolidates information streams from the robot *e.g.,* LIDAR sensor data, diagnostics. It then places this information into the `Blackboard`, allowing a `QRDecorator` can simply access it through an input port.

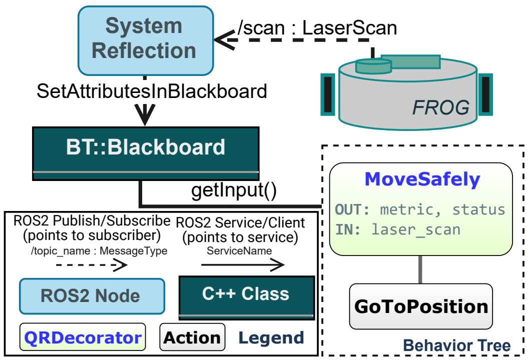

Fig. 3: Overview and Usage Example of `SystemReflection`

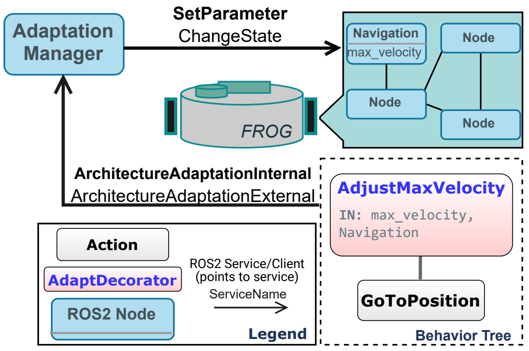

Fig. 4: Overview and Usage Example of `AAL`

C. Architecture-based Self-adaptation with Behavior Trees and ROS2

To address RQ2, which concerns integrating architecture-based self-adaptation, we propose two contributions: an Architectural Adaptation Layer and Adaptation Decorators.

1) Architectural Adaptation Layer: `AAL` is a ROS2 package we created in our effort to standardize the **E**xecution phase of MAPE-K for ROS2-based systems. Note that it is possible to use `AAL` without the rest of ReBeT; we refer the reader to the repository[5] that hosts several demos demonstrating as much.

In our previous work [3], we identified four types of architectural adaptations present in the state of the art of architecture-based self-adaptation in robotics:

1) Reparameterization of component(s)
2) Addition and/or removal of component(s)
3) Change in connection(s) between component(s)
4) Component(s) redeployment

We implement the first three in `AAL` through two ROS2 services `AdaptArchitecture` and `AdaptArchitectureExternal`. We relegate the fourth to future work. We now briefly cover the design and implementation of `AAL` as a whole, and then each of the three architectural adaptations.

a) Overview: Fig. 4 provides an overview of how `AAL` operates. Within the figure, an example `AdaptDecorator` 'AdjustMaxVelocity' uses `AAL`'s `AdaptationManager` ROS2 node to change the maximum velocity parameter of the `Navigation` ROS2 node. In this particular example, as highlighted in bold, it makes use of the `ArchitectureAdaptationInternal` service provided by the `AdaptationManager`, which then uses the `SetParameter` service (provided by all ROS2 nodes) to modify the maximum velocity. `AAL` also offers the possibility for `AdaptDecorators` to use the `ArchitectureAdaptationExternal` service to make use of external adaptation strategies as we detail in Sec. III-C2. For changes in connection between components the `SetParameter` service is also used. As for the `ChangeState` service, this is a ROS2 service provided by every ROS2 lifecycle node. This allows switching a lifecycle node between its seven states, including shutting down and activating the node, which are analogous to their removal and addition, respectively.

b) Reparameterization of component(s): As mentioned, ROS2 nodes are reparameterized through the `SetParameter` service they provide. On the side of the node, this poses the small requirement which is that the execution logic of the node is actively updating the values of its local variables to match those of the parameters. For example, each time the navigation node checks if a velocity is below the maximum, it should compare this to an up to date maximum velocity variable instead of its initial value. This update can be manually encoded, or done through an event handler.

c) Addition and/or removal of component(s): As stated, the addition and or removal of components requires the adapted component to be a ROS2 lifecycle node. Turning a regular ROS2 node into a lifecycle node is as simple as changing one line in the import statement. However, typically one wants to define behavior through event handlers upon transition of states. For example, when the node transitions from inactive to active, the sub-components such as publisher/subscribers or clients/services should also be activated, rather than perpetually being active irrespective of the node's current state. However, for the purposes of adapting the node, we place no requirements on the implementation of these event handlers. The only caveat is that to be able to activate a node, it must already be 'launched' (the process must be running). This means that if one wants to have a system operate without any user intervention, every node which will be added during runtime should be launched at the beginning of execution.

d) Change in connection(s) between component(s): Changing the connection between components in ROS2 constitutes modifying the publishers and subscribers to ROS2 topics. For example, FROG typically uses its own camera feed for detecting objects by having the object detection node subscribe to the same topic its camera publishes its feed to. If FROG's camera were to fail, one could change the connections such that the object detection node is now subscribed to the camera feed of an alternative camera in the room. Our implementation of the mechanism to accomplish is inspired by de Leng and Heintz [23]. To allow a ROS2 node's connections to be modified, we reserve parameters for

[5]https://github.com/EGAlberts/AAL-ROS2

```
virtual bool evaluate_condition() override {
  double curr_safety, curr_power;
  getInput(SAFE_IN, curr_safety);
  getInput(POW_IN, curr_power);
  if(curr_safety < 0.09 || curr_power > 5.0) {
    return decrease_velocity();
  }
  if(curr_safety > 0.15 || curr_power < 4.0) {
     return increase_velocity();
  }
  return false;
}
```

Listing 1: Example of Internal Adaptation Plan

that purpose which correspond to the topics the node either subscribes or publishes to. Then, the node implements an event handler contingent on the change in value of that parameter. Within the event handler, the existing subscriber/publisher is destroyed and replaced with one corresponding to the new topic (indicated by the parameter's new value), but the state of the node (*e.g.,* a running average of all messages received so far) is kept intact. An alternative implementation such as removing/adding the node with a differing initial topic would instead destroy the existing state.

2) Adaptation Decorators: `AdaptDecorators` are used to specify the adaptations of the subset of software architecture supporting performing tasks at runtime. When integrated with ROS2, the leaf nodes of BTs are clients to ROS2 services, which then actuate behavior in the robot. These services have a surrounding infrastructure to support the functionality they provide. For example, Nav2, a navigation library for ROS2, provides a 'NavigateToPose' server which is connected to myriad other ROS2 nodes which eventually actuate the motors driving the robot's wheels. Each of these nodes provides an opportunity for runtime architectural adaptation pertinent to the task of navigation, as we exploit in Section IV to adjust the maximum velocity of a robot. The placement of an `AdaptDecorator` in the BT is a hook into that architectural adaptation, a platform from which self-adaptation can be planned and executed. Every `AdaptDecorator` requires the specification of an `adaptation_location` which provides the name(s) of the ROS2 node(s) that will be adapted. When the adaptation reparameterizes, we require the additional specification of an `adaptation_subject` to identify the parameter(s). Lastly, when using an external adaptation strategy, two more details need to be specified, the name of the adaptation strategy, and the adaptation options that strategy should choose from. For example, the strategy could be 'ϵ-greedy' [24] and the options $\{0.1, 0.5, 1.0\}$ representing velocities in m/s. All these details are specified through input ports into the `AdaptDecorator` BT node, entailing that their values can also be modified by BT logic during execution (see Goal Management in Section III-A).

Each `AdaptDecorator` encapsulates ROS2 clients to the two architectural adaptation services provided by `AAL`, the internal and the external one. The internal service requires adaptation plans be implemented by overriding the `evaluate_condition` method which guards the use of the `AAL` service, *i.e.,* it is invoked when the function returns true. In Listing 1 we give an example of such a strategy for adapting the velocity of the robot at runtime. In contrast, the external service requires a utility function to be provided. This function assigns a utility value to the adaptation chosen by the external strategy within `AAL`.

To aid the definition of both internal and external adaptation strategies we provide four predefined conditions based on the state of the BT node decorated by the `AdaptDecorator`. These can be combined with overridden conditions through inheritance. The first of the four `AdaptOnConditionStart` calls `evaluate_condition` should it be the first time its child node is ticked (see Section II-A2 for a reminder on this mechanism). The remaining three evaluate their condition on return of the state of the child node after being ticked by the decorator. That is, `AdaptOnConditionRunning`, `AdaptOnConditionSuccess` and `AdaptOnConditionFailure` request an adaptation if the child indicates a state of Running, Success, or Failure, respectively. For the earlier example of modifying the maximum velocity, the `AdaptDecorator` would be a subclass of `AdaptOnConditionRunning` so that it modulates the velocity while the robot moves rather than before or afterwards.

IV. Evaluation

In this section we evaluate the two research questions previously stated in Section I separately. Our first evaluation is a showcase encompassing our first contribution of `QRDecorators`. The second consists of a set of experiments performed with FROG. In each evaluation, we evaluate the effect of self-adaptation on the driver of adaptation, which in both cases is quality-driven and therefore we measure the effect on fulfilling QRs. All the material to reproduce the experiments and the results can be found in the replication package[6].

A. Quality-driven Task-based Self-adaptation

In this section we evaluate RQ1, 'How can we integrate the consideration of quality requirements into existing practices of robotics software?'. As we do this in the absence of RQ2 which considers architecture-based self-adaptation, we apply `QRDecorators` to task-based self-adaptation. This demonstrates their functionality independent of the basis of adaptation, given that `QRDecorators` will also be used in the evaluation of RQ2 to follow. In particular, we hypothesize: **The presence of `QRDecorators` along with task-based adaptation in FROG will lead to QRs being continually satisfied throughout operation**. This hypothesis, should it be proven true, would demonstrate integration of explicit QR consideration in a meaningful scenario.

We report on a small experiment in which `QRDecorators` determine the task plan of FROG as it is mapping a room. As it maps the room, it visits the frontiers of its currently known area in attempt to efficiently map the space *i.e.,* performs

[6]https://github.com/EGAlberts/ACSOS-2024-ReBeT-FROG-rep-pkg

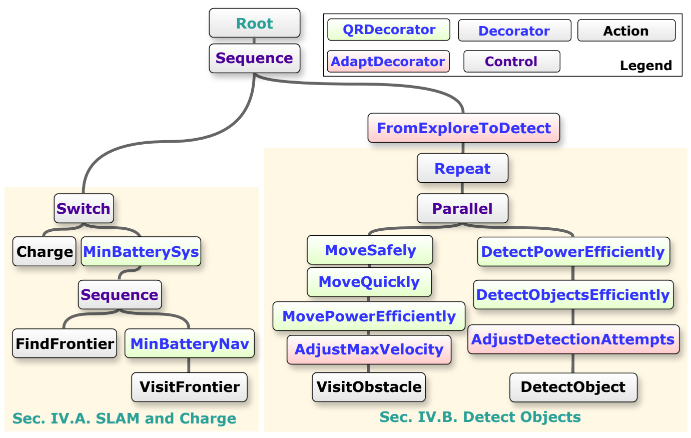

Fig. 5: Complete View of the Behavior Tree Specified for the Two Evaluations

frontier exploration. However, once frontier exploration is complete, the robot must complete a time-sensitive task which may or may not last longer than the remaining charge. We express this constraint through the following QR and its sub-requirement:

> **QR A1** `MinBatterySystem`: The robot must keep its battery percentage above 90% during operation.
> **QR A1.1** `MinBatteryNavigation`: The above requirement, during navigation.

The QR `MinBatterySystem` is a system-level QR with a corresponding task-level sub-requirement `MinBattery-Navigation`. As explained in Section III-B this entails that during operation `MinBatterySystem` collects the metric of `MinBatteryNavigation`. In this case, that metric is the power consumed by the robot's movement. `MinBat-terySystem` combines this with system-level power consumption by the LIDAR sensor and idle consumption. The power consumption values used are based on work by Jaiem *et al.* [25] with a similar robot.

The BT corresponding to this small mission can be found on the left side of Fig. 5. As can be seen, each QR is represented alongside the action nodes which command the robot to charge or perform frontier exploration. To maintain the QR, we implement a straightforward task plan adaptation, which commands the robot to charge when `MinBatterySystem` is in violation. We realize this in the BT through the `Switch` control node which acts on the current status outputted by `MinBatterySystem` and switches between frontier exploration and charging.

In Fig. 6 we plot the satisfaction status of `MinBat-terySystem` over time, alongside the current task the robot is performing during a single run. As the room is static, the execution is deterministic meaning one run is representative of the mechanism. As can be seen, each time `MinBat-terySystem` is not satisfied the current action of the robot changes to charging, until frontier exploration is completed (which in this case, saw three frontiers visited). **This confirms**

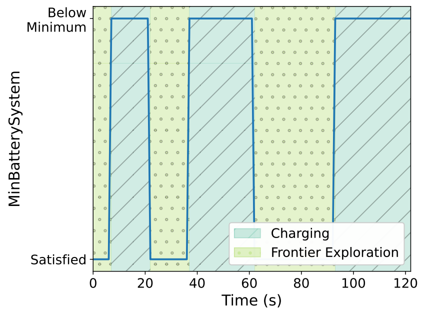

Fig. 6: Task-based Adaptations to Satisfy a Quality Requirement

our hypothesis and shows that `QRDecorators` can facilitate QRs being satisfied through integration into a regular BT.

B. *Quality-driven Architecture-based Self-adaptation*

We restate our RQ2 from Section I which is as follows: 'How can we integrate architecture-based self-adaptation into existing practices of robotics software?'. Particularly, in answering this RQ we hypothesize: **The utility of FROG with architecture-based adaptation through ReBeT is higher than without self-adaptation**. This hypothesis, should it be proven true, would indicate that our designed approach when integrated is effective in its purpose of architecture-based adaptation.

FROG, having completed mapping during in the previous evaluation (Section IV-A), is now tasked with using its onboard camera to detect a target object (in our case a fire hydrant) among a set of 4. Additionally, an external camera facing a fixed angle, can also potentially be used for detection. The BT which represents the mission can be seen in the right half of Fig. 5. When transitioning from mapping to detecting, `FromExploreToDetect` removes the nodes exclusive to mapping and adds those exclusive to detection. Specifically, the frontier exploration service is removed and the object detection service is added. Further, as can be seen, detecting consists of two subtasks, `VisitObstacle` and `Detec-tObject`, each with their own set of QRs in effect and `AdaptDecorators` to try and satisfy the set. Three QRs are imposed on VisitObstacle:

> **QR B1-1** `MoveSafely`: Try to keep a velocity of below 0.18m/s while an object is within the first 10% of the range of the LIDAR.
> **QR B1-2** `MoveQuickly`: Navigate as close to the maximum velocity of 0.26m/s as possible at all times.
> **QR B1-3** `MovePowerEfficiently`: While navigating, conserve as much power as possible.

Clearly, these QRs are partially in conflict with one another *e.g.,* B1-2 requires maximum velocity which uses the most power, in contrast with B1-3 which requires conserving power. Therefore we use self-adaptation to dynamically satisfy these three QRs in accordance with the current operating state. Particularly, we implement both an internal adaptation plan and an external adaptation strategy to change FROG's maximum velocity. The internal plan was previously shown in Listing

System	$p_{\text{succ}}/p_{\text{total}}$		v/v_{max}		$1/W^{\text{mov}}$		U_{vis}		U_{det}	
	mean	std	mean	std	mean	std	mean	std	mean	std
Baseline	0.09	0.03	**0.03**	0.004	**0.78**	0.06	0.001	0.0004	0.09	0.03
ReBeT-Internal	**1.27**	0.10	0.02	0.003	0.49	0.03	**0.004**	0.0005	**1.20**	0.11
ReBeT-External	**1.33**	0.14	**0.03**	0.003	0.56	0.07	**0.005**	0.0013	**1.27**	0.13

TABLE I: FROG Results. Detection Picture Ratio ($p_{\text{succ}}/p_{\text{total}}$); Velocity Ratio ($v/v_{\text{max}}$); Inverted Watts Moving ($1/W^{\text{mov}}$); VisitObs Utility (U_{vis}); DetectObj Utility (U_{det});

1 and decreases the maximum velocity when B1-1 and B1-3 are in violation, while it increases the velocity to satisfy B1-2 if both B1-1 and B1-3 are satisfied. The external adaptation strategy makes use of the UCB multi-armed bandit algorithm [24] which operates by trying to maximize the utility in response to adaptations. The utility represents the simultaneous satisfaction of B1-1 through B1-3, the exact calculation of which we explain later in the section. Two QRs are imposed on DetectObject:

QR B1-4 `DetectPowerEfficiently`: Do not have FROG consume more than 7113 Watts by detecting objects.
QR B1-5 `DetectEffectively`: Perform as many detections of the target object as possible.

The figure of 7113 Watts in B1-4 is a calculation based on the measured power consumed by a singular object detection attempt. We multiply this by the number of obstacles mapped, and parameters which specify how many pictures should be taken per obstacle. The exact calculation can be found in the code specific to FROG's QRs. The two QRs are in conflict as each attempt at a successful detections consumes power. Additionally, during the evaluation we introduce noise into the camera feed of the robot by changing the lighting conditions. In poor lighting conditions it is difficult to successfully detect an object. We implement an internal adaptation plan in `AdjustDetectionAttempts` which depending on the lighting changes a parameter controlling the number of attempts per execution of `DetectObject`. Additionally, when the power budget imposed by B1-4 is depleted, we adapt to change the connection pattern of the ROS2 nodes. Instead of using the camera feed from the camera onboard FROG, the adaptation logic changes the feed to that of the external camera. While the external camera does not consume power on the robot, any obstruction makes the target object impossible to detect. For the sake of demonstration, we ensure the external camera's view is unobstructed.

We define the utility of QRs imposed on `VisitObstacle` as:

$$U_{\text{vis}} = \mathbb{I}\{\text{isSatisfied}(\texttt{MoveSafely})\}\frac{v}{v_{\text{max}}}\frac{1}{W^{\text{mov}}}$$

`MoveSafely` is satisfied when $d^{\text{laser}}_{\text{min}} > 0.10 \vee (d^{\text{laser}}_{\text{min}} < 0.10 \wedge v_{\text{max}} < 0.18)$ is true, where v represents the velocity of the robot, W^{mov} the Watts consumed by the robot moving and d^{laser} the distances of objects around the robot provided by the LIDAR's laser scan.

We define the utility of QRs imposed on `DetectObject` as:

$$U_{\text{det}} = \mathbb{I}\{\text{isSatisfied}(\texttt{DetectPowerEfficiently})\}\frac{p_{\text{succ}}}{p_{\text{total}}}$$

`DetectPowerEfficiently` is satisfied when $W^{\text{det}} > 7113$ where W^{det} is the Watts expended using the object detection service and p represents the pictures used to detect.

From both utility calculation a prioritization in the QRs is clear. There should be no effort in optimizing for QRs B1-2, B-3, and B1-5 if `DetectPowerEfficiently` or `MoveSafely` are being violated. Meanwhile, a tradeoff between the QRs within each task exists between while these two are satisfied.

Each run of the experiment lasts 300 seconds and sees FROG continuously visit each obstacle in the hopes detecting the target object. Table I shows the mean and standard deviation of the utility of each task and their components across 25 runs of the three configurations of FROG we consider (internal, external, baseline). As can be seen, for both utility values ReBeT-Internal and ReBeT-External achieve higher values by a margin exceeding the variance. Noticeable is that for $1/W^{\text{mov}}$, the baseline achieves higher values. However, despite this technically satisfying `MovePowerEfficiently` to a greater degree, this is not translated into U_{vis} due to `MoveSafely` being violated.

To compare the U_{vis} and U_{det} values of the internal adaptation plan and external strategy against the baseline further we perform a Mann-Whitney U rank test with significance level 0.05 to see whether in each case the utilities are greater than that of the baseline. For the case of the ReBeT-Internal we find $7e^{-10}$ for both U_{vis} and U_{det} which indicates that it indeed has a significantly greater utility than the baseline. For the case of the ReBeT-External, we find $6.5e^{-9}$ and $7e^{-10}$ for U_{vis} and U_{det} respectively, which indicates that it is also has a significantly greater utility than the baseline. We can therefore conclude that **in both cases our hypothesis is confirmed**.

V. Discussion and Future Work

A. Threats to Validity

Construct validity: While our RQs mention integration, we do not quantify the integration effort of ReBeT in our evaluation, posing a threat to construct validity. Due to our familiarity with ReBeT as its authors, we abstained from quantifying our own effort in integration as this would introduce unavoidable bias. To counter this threat, our evaluation is based on utility, which is commonplace in self-adaptation research. As future work, we would like to evaluate the integration of ReBeT more aptly *e.g.,* through a case study. **Internal validity**: In our evaluation we assume the changes in utility are caused by self-adaptation. We verify this with the use of a baseline without it. This poses a threat to internal validity, as such a baseline is not a realistic implementation of a system without self-adaptation. Rather, a real system may have other measures to satisfy quality requirements. We counter this as our aim concerns the integration of QRs and self-adaptation rather than the utility of our adaptation mechanisms. **External validity**: We have only applied ReBeT to a singular system. Therefore, a threat to validity is that our conclusions may not hold across applications. While we partially counter this by ensuring a lack of dependencies on FROG in our design and implementation, we recognize future work is necessary in broadening both the range and complexity of systems we evaluate ReBeT on.

B. Added Value of ReBeT Abstractions

While our evaluation has revolved around the utility in satisfying quality requirements, our approach also eases the development of self-adaptive robotics applications. Our approach allows for the decomposition of self-adaptation into its essential elements (*when* and *what* to adapt) as manifested through `QRDecorators` and `AdaptDecorator`, respectively. We believe this to be beneficial to developers of self-adaptive robotics applications. `QRDecorators` and `AdaptDecorators` promote separation of concerns and maintainability as QR satisfaction calculations and adaptation plans can evolve independently of tasks. Their introduction as BT nodes affords them the BT's inherent hierarchy, making it easy to specify the tasks to they apply to. Finally, the introduction of new ROS2 nodes for monitoring (`SystemReflection`) and executing (`AdaptationManager`) centralizes and homogenizes their respective processes. Without them, each task would require ad-hoc realizations for each of the two.

C. Ascribing Architecture-based Self-Adaptation to Existing ROS2 systems

Our work facilitates three distinct types of architectural adaptations through `AAL`. However, only reparameterization is supported by default by all ROS2 nodes. Addition/removal of nodes requires that the ROS2 node is a lifecycle node and change in connection the implementation of an event handler. This makes it more difficult to use existing ROS2 packages. For example, the library we use for SLAM [26] does not use lifecycle nodes. Therefore, a secondary set of mechanisms constrained to the functionality applicable to *all* ROS2 nodes should be made in future work. A potential solution could be extending ROS2's *launch* system which is used for starting groups of ROS2 nodes. For example, we could introduce new launch 'actions' which cover architectural adaptations. As the launch files manage nodes externally, this would remove the need for any changes to the nodes themselves.

VI. Conclusion

In this paper, we describe ReBeT, **Re**-configuration with **Be**havior **T**rees, a framework for building self-adaptive robotics applications through integration into behavior trees. ReBeT strives to treat QRs as first-class citizens when creating robotic mission plans and to make modification of the software architecture of a robotic system accessible at the task plan level. To do this, we extended the BT formalism with two new types of nodes, one for modeling QRs and one for modeling architecture-based adaptations. Our approach comes with a comprehensive runtime adaptation scheme that directly works with the provided variability points and QRs. We implemented ReBeT on top of a state-of-the-art BT library and ROS2, the de facto standard in robotics. Overall, we demonstrate that ReBeT is useful for a range of different adaptation scenarios while leveraging three distinct types of architectural adaptation in robotics applications. We believe it has the potential to be easily adopted and extended to accommodate more forms of architectural adaptation and QRs.

References

[1] International Federation of Robotics, "World robotics report 2022," https://ifr.org/downloads/press2018/2022_WR_extended_version.pdf.

[2] S. Garcia, D. Strüber, D. Brugali, A. Di Fava, P. Pelliccione, and T. Berger, "Software variability in service robotics," *Empirical Software Engineering*, vol. 28, no. 2, p. 24, 2023.

[3] E. Alberts, I. Gerostathopoulos, I. Malavolta, C. Hernández Corbato, and P. Lago, "Software architecture-based self-adaptation in robotics," *Available at SSRN 4805883*.

[4] M. Beetz, L. Mösenlechner, and M. Tenorth, "Cram—a cognitive robot abstract machine for everyday manipulation in human environments," in *Proceedings of IROS 2010*. IEEE, 2010, pp. 1012–1017.

[5] S. Macenski, F. Martín, R. White, and J. G. Clavero, "The marathon 2: A navigation system," in *2020 IEEE/RSJ International Conference on Intelligent Robots and Systems (IROS)*, 2020, pp. 2718–2725.

[6] J. A. Segura-Muros and J. Fernández-Olivares, "Integration of an automated hierarchical task planner in ros using behaviour trees," in *2017 6th International Conference on SMC-IT*, 2017, pp. 20–25.

[7] F. Martín, J. G. Clavero, V. Matellán, and F. J. Rodríguez, "Plansys2: A planning system framework for ros2," in *Proc. of IROS*. IEEE, 2021.

[8] R. Edwards and N. Bencomo, "DeSiRE: further understanding nuances of degrees of satisfaction of non-functional requirements trade-off," in *Proceedings of SEAMS*, Gothenburg, Sweden, May 2018.

[9] D. Weyns, *An introduction to self-adaptive systems: A contemporary software engineering perspective*. John Wiley & Sons, 2020.

[10] D. Garlan, S.-W. Cheng, A.-C. Huang, B. Schmerl, and P. Steenkiste, "Rainbow: Architecture-based self-adaptation with reusable infrastructure," *Computer*, vol. 37, no. 10, pp. 46–54, 2004.

[11] 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, p. eabm6074, 2022.

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

[13] 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*, 2023.

[14] S. Peldszus, D. Brugali, D. Strüber, P. Pelliccione, and T. Berger, "Software reconfiguration in robotics," *arXiv preprint arXiv:2310.01039*, 2023.

[15] A. Romero-Garcés, R. Salles De Freitas, R. Marfil, C. Vicente-Chicote, J. Martínez, J. F. Inglés-Romero, and A. Bandera, "Qos metrics-in-the-loop for endowing runtime self-adaptation to robotic software architectures," *Multimedia Tools and Applications*, 2022.

[16] P. Jamshidi, J. Cámara, B. Schmerl, C. Käestner, and D. Garlan, "Machine learning meets quantitative planning: Enabling self-adaptation in autonomous robots," in *SEAMS 2019*. IEEE, 2019, pp. 39–50.

[17] J. Cámara, B. Schmerl, and D. Garlan, "Software architecture and task plan co-adaptation for mobile service robots," in *Proceedings of SEAMS 2020*, 2020, pp. 125–136.

[18] G. R. Silva, J. Päßler, J. Zwanepol, E. Alberts, S. L. T. Tarifa, I. Gerostathopoulos, E. B. Johnsen, and C. H. Corbato, "Suave: An exemplar for self-adaptive underwater vehicles," in *SEAMS*, 2023.

[19] V. E. Souza, A. Lapouchnian, K. Angelopoulos, and J. Mylopoulos, "Requirements-driven software evolution," *Comput. Sci.*, vol. 28, no. 4, p. 311–329, nov 2013.

[20] J. Kramer and J. Magee, "Self-managed systems: an architectural challenge," in *Future of Software Engineering (FOSE'07)*. IEEE, 2007.

[21] J. O. Kephart and D. M. Chess, "The vision of autonomic computing," *Computer*, vol. 36, no. 1, pp. 41–50, 2003.

[22] E. Alberts, "Multi-armed bandits library." [Online]. Available: https://github.com/egalberts/masced_bandits

[23] D. De Leng and F. Heintz, "Dyknow: A dynamically reconfigurable stream reasoning framework as an extension to the robot operating system," in *Proceedings of SIMPAR 2016*. IEEE, 2016, pp. 55–60.

[24] P. Auer, "Using Confidence Bounds for Exploitation-Exploration Trade-offs," *Journal of Machine Learning Research*, pp. 397–422, 2002.

[25] L. Jaiem, S. Druon, L. Lapierre, and D. Crestani, "A step toward mobile robots autonomy: Energy estimation models," in *Towards Autonomous Robotic Systems*, ser. Lecture Notes in Computer Science. Springer International Publishing, pp. 177–188.

[26] S. Macenski and I. Jambrecic, "Slam toolbox: Slam for the dynamic world," *Journal of Open Source Software*, vol. 6, no. 61, p. 2783, 2021. [Online]. Available: https://doi.org/10.21105/joss.02783