Knowledge-based Execution Configuration for Adaptive Behavior Trees

Oriol Ruiz-Celada*
Institute of Industrial and Control Eng.
Universitat Politècnica de Catalunya
Barcelona, Spain
oriol.ruiz.celada@upc.edu
ORCID: 0000-0002-6171-7270

Jan Rosell
Institute of Industrial and Control Eng.
Universitat Politècnica de Catalunya
Barcelona, Spain
jan.rosell@upc.edu
ORCID: 0000-0003-4854-2370

Raúl Suárez
Institute of Industrial and Control Eng.
Universitat Politècnica de Catalunya
Barcelona, Spain
raul.suarez@upc.edu
ORCID: 0000-0002-3853-7095

***Abstract*—Automated planning is commonly used to obtain plans to solve particular tasks. To execute these plans, Behavior Trees have emerged as a popular execution architecture due to their reactivity and modularity. Configuring the execution of a plan into a Behavior Tree requires expanding the high level actions into the proper Behavior Tree structure. Typically, this is achieved using of pre-defined rigid templates. In this work, we propose a novel approach to generating the Behavior Trees using ontologies. The generated Behavior Trees are tailored to the specific requirements of a task and the world by using modifiers to a base template that provides a general solution to the task. These modifiers and their properties are formally defined using ontologies. A proof of concept is developed, illustrating how the Behavior Trees for the execution of manipulation tasks in a kitchen scenario can be adapted to varying conditions by applying different modifiers.**

***Index Terms*—Robotic Manipulation, Behavior Trees, Automated Planning, Knowledge Representation and Reasoning, Ontologies.**

I. Introduction

To enable robot manipulation programs to adapt to unstructured environments, they must be both general and adaptable to various situations. In this context, research into automated planning has extensively focused on developing planners to address a wide range of tasks, from manipulating small objects to managing large-scale logistics in Industry 4.0 warehouses. In this work, in particular, we focus on the field of task planning, which deals with obtaining the set of actions that the system must execute in order to reach a certain goal state. While planners capable of generating correct action sequences have existed for decades, there remains strong interest in exploring how to effectively integrate acting with planning. This is because existing planning algorithms often use action models that describe what the actions do, but do not describe how these actions will be carried out [1]. For instance, in dynamic environments, under partial observability and subject to interferences, obtaining a robust plan remains a significant challenge because the robot must couple observation actions with actuation actions to adapt to disruptions and uncertainties [2]. In particular, for true robustness, the robot requires observing its environment and be capable of monitoring, refining, extending, updating, changing, and repairing its plan at runtime [3]. To achieve this adaptability to uncertainty, it is necessary to integrate both functional properties (those that affect the world by moving the robot towards achieving the goals) and non-functional properties (those that support the execution, such as monitoring and replanning [4] [5]).

* Corresponding author

The plans generated by the existing planners are usually not directly executable by the robot, needing to be configured into a desired execution structure, such as Finite State Machines [6], Petri Nets [7] or Behavior Trees (BTs) [8]. We call this Execution Configuration. These structures are able to control the flow of the actions in the plan and manage the transitions between the states of the system. BTs, in particular, have gained popularity in robotics due to their modularity and reactivity properties, but it is important to highlight that they also allow configuring and restructuring the execution of a program at run-time instead of at compile time [9]. Different behaviors are represented by their own modular trees, and functional and non-functional properties can be integrated in the same BT as different nodes and subtrees.

We present here an Execution Configuration proposal that translates a high level plan obtained by a task planner into an execution structure, BTs in this work, that will robustly execute the plan. Many current solutions for Execution Configuration rely on implicit representations of knowledge, leading to a series of *if-else* statements in the code, which restricts the implementation to a specific task and limits interoperability and interchangeability [10]. Additionally, the use of templates or action recipes that fix the execution structure for each robot action is also common [11] [12]. These templates are predefined and need to be designed when creating the problem domain, but they can be left undetermined, allowing for parameters to be filled in during planning or during execution, for instance, to select which arm must perform a Pick operation. In these approaches, Execution Configuration consists in joining the templates in the order established by the plan, with specific knowledge queries used to complete them.

While the use of templates is functional, our approach goes one step beyond in how the execution can be configured, by making full use of the knowledge that is available to the robot when orchestrating a plan. In other works, the templates are rigid and do not allow variations based on the knowledge of the world, and therefore may not be specialized to the

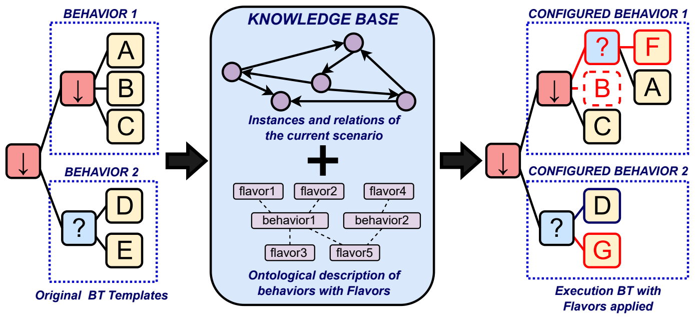


Fig. 1: Illustration of the proposed approach to configure BT templates based on ontological knowledge and Flavors.

specific situation at hand [13], instead, our approach uses this knowledge to tailor and optimize the resulting execution structure to the specific circumstances under which it will be used.

Automating the creation of new robot programs without having to repeat the design process is desirable [14]. To achieve this, robots must have relevant knowledge to perform their tasks and understand the problem domain [15] [16]. The way this knowledge is represented is of key importance and, therefore, the field of Knowledge Representation and Reasoning (KR&R) [17] has gained significant interest as a core capability for cognitive autonomous robotics. A common approach involves the use of ontologies, which represent the concepts in the problem domain and how they are related to one another in a machine-understandable way [18]. Reasoning is then used in robotics to automatically fill the gaps between the pieces of information that have been asserted, inferring new knowledge. This ability to update conclusions when getting new information is necessary for dynamic adaptation in non-deterministic environments [19]. Additionally, the robot must be aware of its own capabilities and skills, and how they can be executed.

In this work, we propose an ontological representation of robotic Behaviors that enables richer descriptions of their skills and supports adaptive execution based on task-specific and environmental conditions. Central to this approach is the introduction of **Flavors**, modifiers applied to Behaviors through ontological reasoning, which allow for the dynamic configuration of BTs. Figure 1 illustrates a generic example where two BTs are modified by applying different Flavors, guided by environmental knowledge. We refer to this process as Execution Configuration using Flavors, and detail its implementation in the following sections.

The contributions of this work are:

- A novel characterization of domain and execution knowledge to define robot Behaviors within Behavior Trees.
- An enhanced ontology-based description of robot Behaviors that enables variations and intricate Behavior Tree structures, while avoiding reliance on predefined templates.
- A knowledge-driven dynamic procedure to generate adaptive Behavior Trees, with reasoned incorporation of both functional and non-functional actions.

After this introduction, the paper is structured as follows. Section II provides an overview of existing literature on Execution Configuration and the tools used in this work. Section III introduces the framework of this work, setting the context for implementing the proposal. Section IV presents the proposed knowledge-based Execution Configuration. Section V offers a proof of concept of the proposed approach in a real-world setting through an example. Section VI presents a discussion of new challenges and future work. Finally, definitions of key concepts used in this paper are summarized in a glossary in an appendix.

II. Background and Previous Works

This section is divided into three subsections. The first two cover the basics of ontologies and Behavior Trees (BTs), the two main tools used in this work, and the third one contains an overview of previous Execution Configuration approaches.

A. *Ontologies*

Ontologies are widely used formalisms in the field of robotics as a method to represent knowledge [20]. They make explicit all the relevant information of a domain and present it in a machine-readable format, enabling software to engage in logical reasoning over the knowledge, facilitating the inference of additional information. The use of semantics is crucial for providing a comprehensive depiction of the environment, the available actions, and the state variables, which ensures an efficient and effective execution of robot actions [19].

With the aim of facilitating integration of different ontologies through the web, the shared Ontology Web Language (OWL) [21], specified by the World Wide Web Consortium [22], was created. OWL integrates different formats (*.xml*, RDF, RDF-Schema) to enable multiple systems to share semantic information. OWL is based on Description Logic (DL) [23], allowing the use of DL reasoners such as Pellet [24] or HermiT [25]. OWL also allows formalizing a domain into *Classes*, *Individuals* (members of those classes), *Object Properties* (relating two individuals to each other), and *Data Properties* (relating an individual to a data value). Ontologies can be created using editors such as Protégé [26].

Ontologies are categorized into three levels of abstraction: Upper, Domain, and Application ontologies [27]. Upper ontologies, such as the Suggested Upper Merged Ontology (SUMO) [28], define universally applicable concepts like space, time, matter, object, event, and action, independent of specific domains. Domain ontologies define terms specific to a particular field, with the Core Ontology for Robotics and Automation (CORA) [29] and the more recent Autonomous Robotics Ontology (AuR) [30] being standards for the domain of autonomous robots within the robotics field. Application ontologies encapsulate all the knowledge necessary to describe a particular scenario or task. The use of Upper and Domain ontologies is encouraged to facilitate the integration of works from different research groups working in the same field [31].

DL reasoners can be used to infer new knowledge, by taking into account the features that the properties in the ontology may have, like being transitive or symmetric. For instance, when describing a scene with objects, spatial relationships can be defined with properties like `isNextTo`, which has the symmetry feature [32], allowing the reasoner to automatically infer that if A `isNextTo` B then B `isNextTo` A. For more complex reasonings, there is the Semantic Web Rule Language (SWRL) [33], whose rules are formed by an antecedent and a consequent, each of them formed by multiple predicates that refer to knowledge in an OWL format: classes, properties, individuals and data values [19].

B. Behavior Trees

Behavior Trees [8] allow for a natural way of defining Behaviors [9]. Their use in robotics, specially in skill-based systems, has risen due to their modular, interpretable and reactive capabilities [34].

Behavior Trees (BTs) are directed rooted trees formed by nodes. A tick signal propagates at a given frequency from the root of the tree to the nodes, executing them when ticked. Nodes may return either `SUCCESS`, `FAILURE` or `RUNNING` (meaning that the execution is still taking place). Nodes can be classified as:

Control Nodes: they have a number of children nodes, and are used to control their execution flow by regulating the tick signal. In turn, they can be:

- *Sequence Nodes* (indicated by "$\rightarrow$") that return `SUCCESS` only if all the children nodes return `SUCCESS`.
- *Fallback Nodes:* (indicated by "?") that return `SUCCESS` as soon as one of the children nodes returns `SUCCESS`.

Leaf (or execution) Nodes: they have no children and are used to represent single functionalities. In turn, they can be:

- *Condition Nodes:* that are used to represent simple checks (e.g., "is the door open?"), and can return either `SUCCESS` or `FAILURE`.
- *Action Nodes:* that are used to represent actions (e.g., "move the arm to a pose"), and can return either `SUCCESS`, `FAILURE` or `RUNNING`. Their execution can be preempted.

Nodes have ports to access a key/value storage, called the BT blackboard, where input/output data is read/written. Grouping the nodes into subtrees allows reusing whole branches of BTs.

C. Execution Configuration

The integration of knowledge in Execution Configuration has not been fully realized yet. Some approaches use knowledge to enhance the planning capabilities, but then directly execute the actions of the planner [19] [20]. In these approaches, there is a significant limitation regarding the need for re-planning when the robot actions fail due to uncertainties. These uncertainties may stem from unexpected outcomes or disparities in the dynamic state of the world, leading to frequent plan failures. Re-planning is both time-consuming and resource intensive, especially for complex tasks involving numerous objects and actions.

In contrast, BTs inherently offer reactivity by continuously checking conditions and switching actions based on the perceived world state. However, scaling up BTs for tasks with many objects and subtasks is challenging, as larger BTs become difficult to be manually designed. The topic of automatic BT generation has been growing in popularity [35], with two main strategies: learning-based and planning-based [36]. Learning-based approaches are able to learn how to generate BTs through reinforcement learning [37], evolution-inspired learning [38], genetic programming [14] and the use of Large Language Models [39] [40]. While these methods are promising, they suffer from lack of explainability regarding the final BT structure obtained.

Planning-based approaches, on the other hand, use planning algorithms applying reasoning with specific knowledge to refine BTs, combining the reactivity of BTs with the goal-directed nature of planners. The typical process uses a planner to first compute a task-solving plan and, then, converts the actions of this plan into a BT [36]. This has been used in several works [3] [9] [10] [41] [42], including also our previous work [12], where a planner based on the Planning Domain Definition Language (PDDL) [43] was used.

A notable contribution in this area is the work of Colledanchise et al. [44], who proposed an automatic planning approach for Behavior Trees (BTs) based on backchaining. Starting from an initial atomic BT, the system dynamically expands the tree whenever a condition fails, replacing it with a subtree that achieves the failed condition. This enables the BT to adapt to environmental changes without the need for complete replanning. By associating actions with preconditions and postconditions, the approach supports reactivity and modularity. Additionally, conflicts in action sequences are addressed by dynamically reordering the tree to maintain feasibility. While the method is highly reactive, it has limitations in the types of adaptations it can handle.

Extended Behavior Trees [9] (eBTs) add an additional step in the refining of the planned BT by using an optimizing algorithm that merges and parallelizes BT structures into a more compact and efficient BT. The applications of eBTs shown in [45] are represented by parallelizable motion primitives coordinated through BTs and superposed via motion generators. This method enables reactive adaptation to environmental changes by dynamically blending motion contributions. While it improves modularity and robustness, especially for low-level motion behaviors, it focuses mainly on local reactive strategies rather than complex high-level task reasoning.

In partially observable environments, the combination of both sensing an acting becomes vital for a robust task execution. The Adjoint Sensing an Acting (ASA) model [46] starts from a Partially Observable Markov Decision Process-planned BT and automatically extends the actions into two types of custom-made BT structures, using sensing for acting and using acting for sensing. This approach leads to more robust plans for simple actions, but with limited complexity of BT structures that can be used to represent robotic skills.

Despite the advantages of these approaches in generating reactive and adaptable BTs, they typically do not incorporate ontological reasoning in the configuration of BTs. This limits their scalability to more complex domains where task understanding and skill composition require deeper semantic grounding. Our approach allows for reasoning to adapt the refining of Behavior Trees with ontological knowledge.

III. Framework

The work presented in this paper is framed in the ontology-based adaptive robotic manipulation framework BE-AWARE [47], which is intended to be used in robotic manipulation scenarios with dynamic environments. The BE-AWARE framework uses Knowledge Representation and Reasoning capabilities to allow awareness of the world to plan the actions to evolve towards the goal, monitor the execution of the resulting plan, and adapt it in case of disruptions.

The proposed approach for Execution Configuration is responsible for generating a BT adapted to the current scenario, which can in turn react and adapt the execution to unexpected changes. For this, BTs of planned Behaviors are configured to include monitoring and recovery branches.

A. The Execution Configuration Module

The Execution Configuration approach presented in this paper constitutes the Execution Configuration Module of the BE-AWARE framework. The pipeline illustrated in Figure 2 shows the steps necessary to go from the introduction of a new goal in the framework, to the execution of the BT generated to reach it. When a new goal is introduced, the new problem is configured in the (Re)Planning Design module, which converts the knowledge in the Knowledge Base into the PDDL files needed for planning. The procedure to generate PDDL files from ontological knowledge is detailed in our previous work [48]. The PDDL files are sent to an off-the-shelf task planner, such as Fast Forward (FF) [49], to obtain a high-level task plan to change the current state into the goal state. The Execution Configuration serves as the bridge between high-level Task Planning and low-level Execution, by obtaining the *Output Behavior Tree*, which is the BT that will actually execute the plan. The proposed approach for Execution Configuration relies on the Knowledge Manager, explained below, to retrieve the available knowledge. Finally, the Output Behavior Tree is executed by the BT Executor Module.

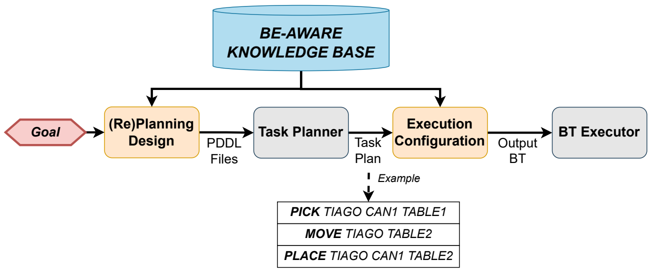


Fig. 2: Pipeline schema between Planning and Execution.

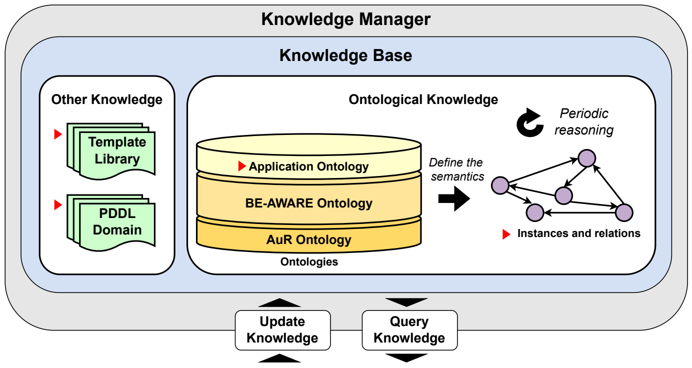


Fig. 3: Knowledge Manager of the BE-AWARE framework and its components. The red triangle symbol indicates elements with knowledge that the user has to provide for their own application.

B. Knowledge Manager

The Knowledge Manager, detailed in Figure 3, manages the Knowledge Base that contains all the knowledge that the functions in the framework require. The Knowledge Manager opens an entry point for queries to retrieve existing knowledge and to introduce updated knowledge into the Knowledge base. The Knowledge Base has two categories of knowledge:

- **Ontological Knowledge:** It is the knowledge defined in an ontology. There are three ontologies in the Knowledge Base. The first one, the Base Ontology, is the *IEEE Standard for Autonomous Robotics Ontology* (AuR Ontology) [50]. Recent surveys on ontologies in robotics [31] show the necessity for standardization and compatibility between projects related to the use of ontologies in autonomous robotics. Basing the other ontologies in the framework on this standard facilitates compatibility of the BE-AWARE framework with the broader research community. The second ontology is the BE-AWARE *Ontology*, developed to define all the concepts (Classes, Properties, SWRL rules) necessary for an integrated implementation of the different functions of the framework, including the concepts defined below in Section IV for the Execution Configuration. The BE-AWARE Ontology is compliant with the AuR Ontology and is domain neutral within the field of autonomous robots. For a specific domain, concepts such as types of objects, robots and their capabilities, behaviors and properties of the world need to be defined. To do so, the user has to define these concepts into a third ontology, the *Application Ontology*, which needs to be compliant with the BE-AWARE Ontology, and, therefore, with the AuR Ontology.
  When initializing the framework, the Knowledge Base is populated with instances of the concepts defined in the ontologies. These instances have their own properties and relations with one another, following the semantics defined in the ontologies. As an example, the BE-AWARE Ontology includes the class `Object`. The user, when creating their own Application Ontology, can create the subclasses `DangerousObject` and `Knife`, and define,

for their own domain, that all knives are dangerous. When the perception module detects a knife, an instance of the class `Knife` is created, and reasoning will automatically classify it as a `DangerousObject`, information that then can be used by the other modules in the framework.

- **Other Knowledge:** It is the Knowledge that is not defined in the ontologies. For Execution Configuration, this knowledge includes the PDDL files for planning and the *Template Library*, which includes the different BT Base Templates for the different Behaviors needed.

For the implementation of the Knowledge Manager, the Owlready2 library [51] is used, which loads the ontologies and instances into an optimized quadstore based on SQLite3. Owlready2 permits the use of SPARQL [52] queries to interact with the Knowledge Base, retrieving and updating information. The benefit of the framework structure is that all the modules can access this knowledge using these queries. New knowledge can be asserted, such as creating a new instance of the `Can` class when a new can is seen by the sensors, or retrieved, such as asking what kind of end effector `Tool` a robot has. A reasoner, currently Pellet [24], allows inferring and deducing new facts and relations inside the Knowledge Base. The reasoner is run periodically, so the framework has an up-to-date awareness of the state of the world.

IV. Execution Configuration

In the Execution Configuration presented here, reasoning is used so that the same Behavior, such as a `Pick`, is configured into different BT structures depending on the conclusions reached using the current knowledge. For instance, the Execution Configuration can come to the conclusion that in order to perform a `Pick` of a book that is on a table, there is the need to check if the book can be grasped in its current position, and, in case it is not, trigger a contingency branch that pushes the book slightly beyond the edge of the table before attempting a lateral grasp. Many types of changes can be made to the basic template of a BT to better suit a specific instance of the problem, but they need to be formally defined and implemented. In this proposal, these modifications are referred to as *Flavors*, which are further discussed in Subsection IV-C.

In addition to Flavors, the approach developed in this work allows for full awareness and control of the resulting BT structure. Monitoring and replanning capabilities, in the form of BT branches, can also be integrated into the Output Behavior Tree thanks to its modular structure.

The main aspects of the Execution Configuration are detailed in the next subsections, as follows:

- Subsection IV-A describes the concepts of Primitive, Behavior and Task Plan, and how they are related to Execution Configuration.
- Subsection IV-B explains the pipeline of the Execution Configuration and how it is related to the rest of the modules in the framework.
- Subsection IV-C explains in detail the concept of Flavor, the types of Flavors and the effects they have on the generation of BTs.
- Subsection IV-D details the main function in the Execution Configuration (function `grow_subtree`), which generates the BT to be executed.
- Subsection IV-E discusses how reasoning can help to automate the process of designing Behaviors and applying Flavors to them.
- Subsection IV-F provides information on the implementation of the proposal.

A. Main Concepts and Terminology

There is an array of terminology used in the literature to refer to *what* is being planned, e.g., actions [11] [53], skills [20] [53] [54], behaviors [14], tasks [39], processes, action primitives [55], and commands [53], but there are some overlaps that need clarification. A hierarchical structure formed by Tasks, Skills and Primitives was defined in [56], where: *Tasks* represent high-level goals, e.g., *set the table*; *Skills* are the building blocks for the robot to complete tasks, representing its capabilities, e.g., *place object* and *open door*; and *Primitives* are the basic commands that cannot be divided further, e.g., *arm motion* and *close gripper*. While this hierarchy fits with the purposes of the Be-Aware framework, a slight change was made to be more in line with common BT terminology. The terminology used in this work is:

- **Primitive:** An indivisible part of the execution, represented as a BT execution node (Action Nodes represent *Action Primitives* and Condition Nodes represent *Condition Primitives*).
- **Behavior:** A composition of more than one Primitive, forming a BT, that enables the execution of more complex actions. Behaviors consist of Primitives and other Behaviors connected by Control Nodes. The term "Behavior" is used instead of "Skill" because, in the literature, skills are typically associated with a robot's capabilities. In contrast, this work uses "Behavior" to refer to any combination of Primitives with meaning, regardless of whether it qualifies as a robot capability.
  A subset of the Behaviors is also represented as PDDL Actions in the PDDL domain file; these are referred to as *Plan-Level Behaviors*. Consequently, the high-level task plan generated by the task planner is composed of *Plan-Level Behaviors*. The information from the PDDL domain file (including the PDDL predicates, preconditions, and effects of the actions) is utilized by the framework when handling the *Plan-Level Behaviors*.
- **Task Plan:** Sequence of Plan-Level Behaviors that solves a particular problem. The example in Figure 2 shows a basic Task Plan formed by a sequence of the Plan-Level Behaviors `Pick`$\rightarrow$`Move`$\rightarrow$`Place`.

`Primitive`, `Behavior` and `TaskPlan` are all classes defined in the Be-Aware Ontology, as well as the class `PlanLevelBehavior`, defined as a subclass of `Behavior`. The user can create subclasses of these (to have categories of Behavior or Primitive that share properties), as well as define those that will have an associated PDDL Action as being a subclass of `PlanLevelBehavior` and therefore part of a `TaskPlan`. For instance, the user can define the classes `ManipulationBehavior` and `NavigationBehavior` (to differentiate between two groups of Behaviors in their domain, each with some common properties), and define class `Pick` as a subclass of both `ManipulationBehavior` and `PlanLevelBehavior`.

In the Knowledge Base, regular Behaviors have two representations, while the subset of Plan-Level Behaviors have three. They are the following:

- **Ontology representation:** An instance of the class `Behavior` with some semantic properties. When creating the Application Ontology, the user has to populate it with instances of the Behaviors that exist in their domain, and assert their properties, such as asserting the Flavors that each Behavior has. Figure 4a shows a Behavior Instance in the Knowledge Base and some of its properties.
- **Behavior Tree representation**: A Base Template of a BT that is the basis over which the modifications described in the Flavors will be applied. Base Templates may include subtrees which are Base Templates of other Behaviors, as shown in Figure 4b. Base Templates are stored in the *Template Library* in the Knowledge Base using a *.xml* format. The `hasBTId` property in the Knowledge Base is used to link a Behavior instance with the ID of the corresponding Base Template in the library.
- **PDDL representation** (only for Plan-Level Behaviors): A PDDL Action with its parameters, preconditions and effects, stored in the PDDL domain file. Each Plan-Level Behavior has its own associated entry. Figure 4c shows a PDDL Action definition for the `Pick` Plan-Level Behavior.

B. Execution Configuration Pipeline

The Execution Configuration Pipeline starts with the input of a Task Plan and ends with the Output Behavior Tree to be executed, and has several steps detailed below. It is illustrated in Figure 5 (top) and exemplified in Figure 5 (bottom), showing how the Output Behavior Tree is grown after each step.

First, Step 1 creates a BT for the Task Plan, which is composed of a sequence node with a child subtree per Plan-Level Behavior in the Task Plan. Then, iteratively, Steps 2 and 3 are executed to parse the Behaviors and grow the subtrees. Step 2 parses each Behavior into a *Seed*, which is used to grow a BT in Step 3. A Seed includes the Behavior and a set of parameters to adjust the BT to the current situation. In particular, the Seed directly points to the instances of both the Behaviors and the parameters in the Knowledge Base, which will allow the Execution Configuration to query any other information related to them and needed to grow the tree. As an example Figure 6 shows a step of the plan given by the task planner, *PICK TIAGO CAN1 TABLE1*, which is parsed into the Plan-Level Behavior `pick`, the robot `tiago`, the target object `can1` and the location `table1`.

The Seeds are parsed by finding a match between the Behavior and the parameter names in the Task Plan to existing instances in the Knowledge Base. In the example above, after parsing, the system can identify the robot carrying out the action (`tiago`) and retrieve all its properties and relationships from the Knowledge Base, such as: it is a dual-armed mobile robot, it has a front-facing camera, it has a prismatic torso that can be lowered or raised to reach different heights, it has planar grippers, it has a battery level of 50%, and it is currently located facing `table2`. This knowledge is used later to determine how the resulting BT will be structured when applying the Flavors.

In Step 3, each Seed is used as the input for the `grow_subtree` function, detailed below, which loads the Base Template for the Behavior indicated in the Seed, configures it by applying the Flavors, and adds it to the Output Behavior Tree. If at the end of the call to the `grow_subtree` function there are calls to other subtrees, representing sub-Behaviors of the current Behavior, steps 2 and 3 are repeated, creating a new Seed that inherits the parameters of the previous one, but with the new Behavior, and `grow_subtree` is executed again. This is done recursively until all leaves remaining in the Output Behavior Tree are Primitives.

C. Flavors

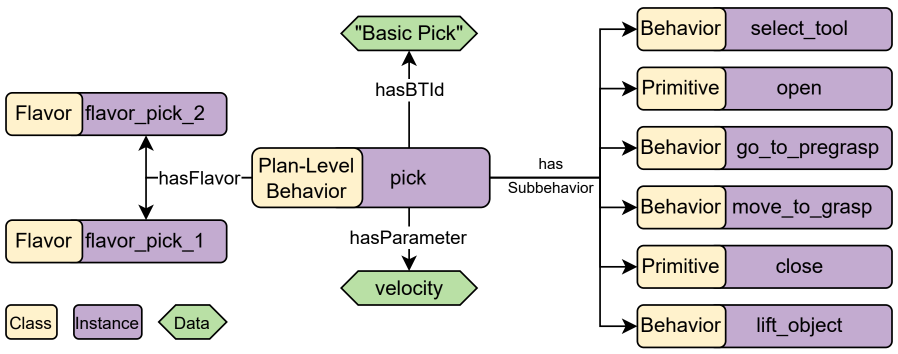
A Flavor is a modifier used to adapt a Behavior, modifying its Base Template if certain circumstances are met. This modified BT is introduced in the Output Behavior Tree to be executed. The modifications that Flavors apply to the Base Templates include:

- reordering branches,
- adding or modifying control nodes by adjusting parameter values,
- modifying Primitives,
- and replacing, inserting or deleting sub-Behaviors.

With Flavors, the Behaviors are enriched, enlarging the number of BT structures that can be generated from a given Behavior. For instance, the same `Pick`, defined by a single Base Template, may have different Output Behavior Trees depending on what Flavors have been applied to it. Flavors are represented as instances of the class `Flavor`, and are linked to instances of `Behavior` through the property `hasFlavor`. The example in Figure 4a shows the `Pick` Behavior with two Flavors.

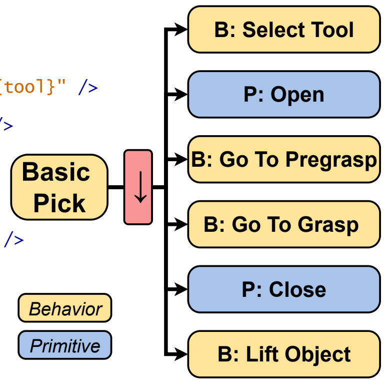
Flavors are formed by three main components:

- **FlavorTarget(s)**: Part, or parts, in the Base Template being affected by the Flavor, like Primitives, BT control nodes and other Behaviors, as well as the parameters in the nodes of the BT.
- **Trigger**: Condition that determines if the Flavor is applied or not. They serve a similar function as the applicability rules used in [13] to reason on when to transform the original robot Behavior at runtime. Triggers are checked through SPARQL queries to the Knowledge Base that return either True or False. The queries are stored in a generic way, with parameters that represent information that needs to be filled out. As an example, using the Seed in Figure 6, the query of the Trigger in Figure 7 will be set by substituting `<obj>` with the name of the `obj` parameter in the seed



(a) `Pick` Behavior as an instance of the Behavior class in the Knowledge Base, defined in the Application ontology.

```
<BehaviorTree ID="Basic Pick">
  <Sequence name="Pick Sequence">
    <SubTree ID="Select Tool" tool="{tool}" />
    <Action ID="Open" tool="{tool}" />
    <SubTree ID="Go To PreGrasp" />
    <SubTree ID="Go To Grasp" />
    <Action ID="Close" tool="{tool}" />
    <SubTree ID="Lift Object" />
  </Sequence>
</BehaviorTree>
```



(b) Base Template of the `Pick` Behavior in *.xml* and its corresponding BT.

```
(:action pick
:parameters (?rob - robot
    ?obj - object ?surface -
    location )
:precondition (and (not
    (clear ?obj)) (toolempty
    ?rob) (on ?surface ?obj))
:effect (and (holding ?rob
    ?obj) (not (on ?obj
    ?surface)))
)
```

(c) `Pick` Behavior as a PDDL Action in the PDDL domain file.

Fig. 4: Representations of the Behavior `Pick` in the BE-AWARE framework.

(`can1`). In this way, the generic query is specified according to the current Seed. If the picked object is a `Can` (which does not belong to the `DangerousObject` class), the trigger returns False, and the Flavor will not be applied. In the case that the picked object was a `Knife` (which is a Subclass of `DangerousObject`), the trigger would return True, and the Flavor would be applied. The same Behavior can result in different BTs depending on the results of the Trigger queries, which in turn depend on the properties and relations of the parameters in the Seed.

- **Keyword**: Type of modifier that the Flavor is. In the ontology, the Flavor class has several subclasses identified by these keywords, and each Flavor belongs to one of them. For instance, the keyword "Ignore A" indicates that sub-Behavior A in the Base Template needs to be eliminated; and the keyword "Delay A Until After B" indicates that A needs to be moved after B in the sequence of the Base Template.

Table I shows examples of two Flavors, and Figure 8 shows their visualization in the Knowledge Base (their trigger queries have been omitted for space). Thanks to the ontology structure, a Flavor can also be applied to an entire subclass of Behaviors, such as Flavor 2 that applies a velocity modifier Flavor to all `ManipulationBehaviors`.

	Flavor 1	**Flavor 2**
Flavor Description	*If the robot has a hand, don't open it until GoTo-Pregrasp has been done*	*If the object manipulated is dangerous, reduce velocity by 30%*
Behavior	Pick	Manipulation Behavior
Keyword	**Delay** A until after B	**ModifyParam** P [-30%]
Trigger	{if `robot` hasEndEffect of class `RobotHand`}	{if `object` is of class `DangerousObject`}
Flavor Targets	A=open, B=go_to_pregrasp	P=velocity, Modifier=-30

TABLE I: Two examples of Behavior Flavors.

Table II shows the currently implemented library of Flavors and the effect they have on a BT. Flavors are organized into subcategories depending on what is modified in the Base Template. This eases the implementation of the Flavors. For instance, the `SequenceModifier` class of Flavors indicates that the Flavor refers to sub-Behaviors that are children of the same sequence control node. Sub-Behaviors in the Base Template are referred with A, B, C, etc. in Table II.

This library of Flavors serves as a proof of concept and illustrates the variety of possibilities of the proposed approach, but this list can be extended. The approach allows for new Flavor to be added with relative ease, expanding the adaptation possibilities.

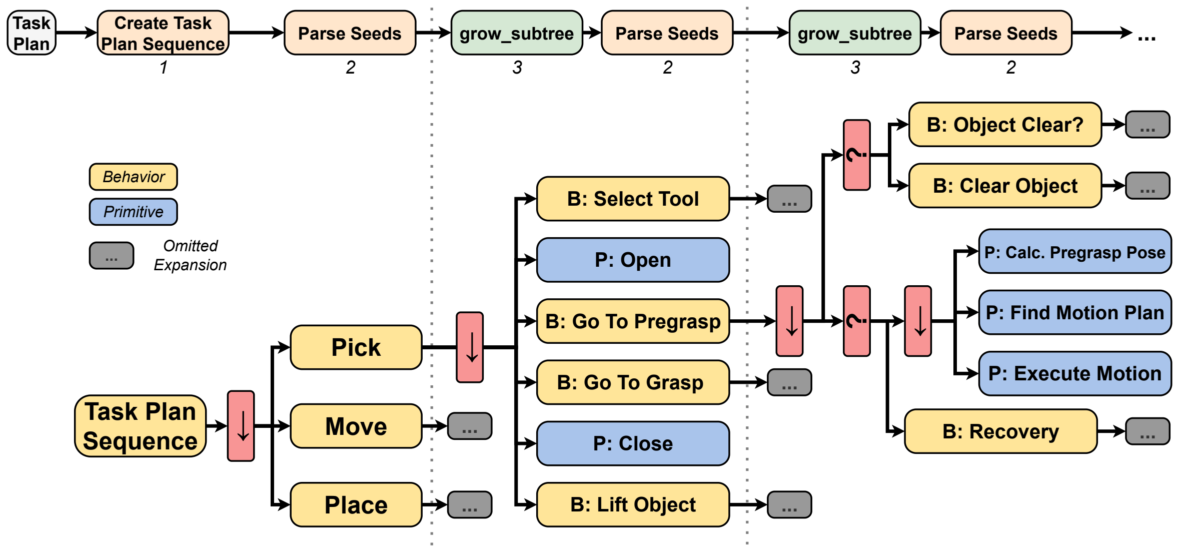


Fig. 5: Execution Configuration pipeline (top), accompanied by an example (bottom) of the Output Behavior Tree of a `Pick` part of a Task Plan (the Behavior Tree has been simplified and compressed due to space constraints).

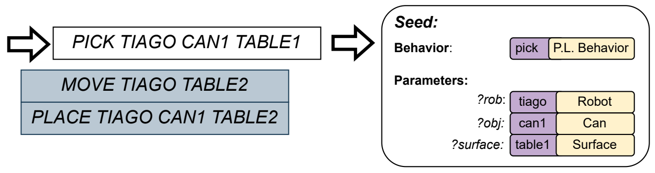


Fig. 6: Parsing the Task Plan into Seeds. Process repeats for each Plan-Level Behavior in the plan.

```
ASK {
  <obj> rdf:type/rdfs:subClassOf* apponto:DangerousObject
}
```

Fig. 7: Generic form of the SPARQL query for the trigger "if the object is Dangerous". `<obj>` is a parameter that will be substituted by the `obj` parameter in the Seed (parameters that represent information that needs to be filled out are indicated by < >).

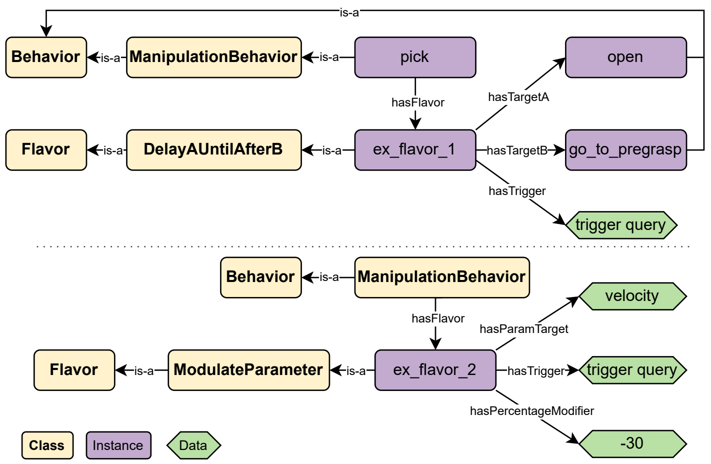


Fig. 8: Representation of the Flavors in Table I in the Knowledge Base.

D. Growing a Behavior Tree

The process of growing a BT is done in the `grow_subtree` function, shown in Algorithm 1. The input to the function is the Seed, which includes the Behavior and the current related parameters, as well as the root of the BT where the output will be inserted. The Behavior is extracted from the Seed (Line 2), and its corresponding Base Template is loaded from the Template Library (Line 3), generating a first verison of the Output Behavior Tree. The Behavior has a number of Flavors tied to it through the `hasFlavor` property, which are applied one by one (Lines 4 to 8). The function `query_trigger` (Line 5) uses the parameters in the Seed to specify the Trigger (which initially is in a generic form as that shown in Figure 7), and a SPARQL query is sent to the Knowledge Base. If the return is True, then the Flavor is applied. The function `apply_flavor` (Line 6) applies the modification to the `output_bt`, following the implementation of each Keyword.

After all the Flavors have been applied, the resulting `output_bt` is added to the root (Line 9). In this new tree, there may be calls to other Behaviors, in the form of `Subtree` calls. For each of them (Lines 10-13), a new Seed is parsed in the `parse_seed` function, that takes the ID of the `Subtree` call, and the new Seed will inherit the parameters of the parent Seed. With this new Seed, the `grow_subtree` function is called again. Growing the resulting BT is a recursive process that is repeated until all the leaves in the final tree are Primitives.

A Behavior can have multiple Flavors affecting the same part of its Base Template. This can lead to some scenarios with

Flavor Keyword	**Effect Description**	***Example***
Sub-BehaviorModifier	Flavors that modify the call to a sub-Behavior directly	
Ignore A	The sub-Behavior is eliminated from the tree	*Skip going to the pregrasp position if the object is small, go directly to grasp it*
Replace A by B	The call to A is replaced by a call to B	*If the object is categorized as heavy, replace the Pick by a dual-armed Pick.*
A has Precondition B	A is replaced by a sequence B→A. Optionally, a recovery Behavior can be selected to executed if B returns `FAILURE`	*If the region where the Behavior will take place is a drawer, add the precondition to check if it is open before manipulating in there. If not, do the Behavior to open it.*
A has Postcondition B	A is replaced by a sequence A→B. Optionally, a recovery Behavior can be selected to be executed if B returns `FAILURE`	*If the container being interacted with is an Electrical Device, check that it has been turned OFF at the end of the Behavior.*
A has Holdcondition B	A is replaced by a parallel execution B||A. Optionally, a recovery Behavior can be selected to be executed if B returns `FAILURE`	*Hold a bowl with one hand while the other stirs it. If the holding Behavior fails, stop stirring and trigger the recovery Behavior.*
B alternative to A	A is replaced by a fallback A?B, meaning that if A fails, B is done as an alternative	*If a grasping done with the left arm fails, try to do it with the right arm*
SequenceModifier	Flavors that modify the elements that are in a sequence.	
Delay A Until After B	Move the call to A to be done right after B	*If the robot has a hand, don't open it until GoToPregrasp has been done*
A Redundant with B	A and B must be together in a sequence A→B. This is replaced by A?B, indicating that if A returns `SUCCESS`, B does not need to be done.	*Try to locate an object with the robot camera and with the room camera. If you only need to locate it with one, the two Behaviors are redundant*
Switch A and B	A and B switch places in the sequence	*Switch the pouring of the cereal and the pouring of the milk*
SymbolicStateFlavors	Flavors that relate to the symbolic state in the knowledge base	
A sets property Q	A call to a SetProperty primitive is added after the execution of A to assert the value of Q	*After a button has been pressed (A), assert that the oven has the property `on` (Q)*
A requires property Q	A call to the GetProperty condition primitive is done before A, leading to a recovery if the condition Q is not met	*Before pouring a drink to a glass (A), check that the glass is `empty` (Q)*
ParameterModifier	Flavors that modify the parameters in primitives, sub-Behaviors and the BT blackboard	
Fix Parameter	Set the value of the parameter P in the blackboard to a specific amount	*Set the torso joint to 30 cm.*
Modulate Parameter	Modulate the parameter P in the blackboard by a percentage from its base value.	*Increase velocity by 50%*
Replace A by Parameter	Replace the Behavior A by setting the parameter P in the blackboard.	*Replace the Behavior `select_robot` for setting the parameter (robot="tiago") if tiago is known as the only robot available*
Add Kinematic Restriction to A	Add a kinematic restriction to the Behavior A.	*If the target object is a container filled with a liquid, such as a glass, add the Kinematic restriction that it must be held upright*

TABLE II: Currently implemented Flavor Library.

Algorithm 1 grow_subtree

```
1:  procedure GROW_BT($seed$,$root$)
2:      $behavior \leftarrow seed.behavior$
3:      $output\_bt \leftarrow load\_template(behavior)$
4:      for each $flavor$ in $behavior.flavors$ do
5:          if query_trigger($flavor.trigger, seed$) then
6:              $apply\_flavor(output\_bt, seed, flavor)$
7:          end if
8:      end for
9:      $root.append(output\_bt)$
10:     for each $Subtree$ in $output\_bt$ do
11:         $new\_seed \leftarrow parse\_seed(Subtree.ID, seed)$
12:         grow_subtree($new\_seed$,$root$)
13:     end for
14: end procedure
```

overlapping effects, such as having a Flavor deleting one sub-Behavior and another modifying it. The current implementation applies the Flavors in the order that they are retrieved from the ontology when querying for the `hasFlavor` property of the Behavior, optimizing this in some way is as open point for future work.

E. Flavor Reasoning

Flavor Reasoning refers to the use of reasoning techniques to automatically assign a Flavor to a Behavior. While Flavors introduce richer representations for Behaviors, there is nonetheless a cost in the complexity of the knowledge that needs to be asserted by the user when designing the problem domain. Other than the design needed for the PDDL domain file and the Base Templates, the user also needs to assert the Behaviors and the Flavors in the Application Ontology, and, in this contetx, Flavor Reasoning is proposed to reduce this effort.

Thanks to the use of ontologies and Knowledge Representation and Reasoning techniques, not all the domain needs to be explicitly asserted. Using reasoners and a proper initialization of simple rules, Flavors can be assigned to the Behaviors through inference. This can be done in two ways:

- **Flavor Inheritance:** A Flavor can be assigned to a Behavior subclass instead of to a specific Behavior instance, so that any Behavior belonging to the subclass will inherit this Flavor. Flavor 2 in Table I serves as an example of this; the Flavor to reduce the speed if the manipulated object is classified as `DangerousObject` is asserted for the subclass `ManipulationBehavior`, then the Flavor is

```
Rule: Behavior(?parent),
    hasParentBehavior(p_open, ?parent),
    hasParentBehavior(b_go_to_pregrasp,
    ?parent) -> hasFlavor(?parent,
    flavor_swrl_example)
```

Fig. 9: SWRL rule to assign a Flavor to any Behavior that has both `open` and `go_to_pregrasp` as sub-Behaviors.

automatically inferred for the Behavior `pick`, as well as for any other `ManipulationBehavior`.

- **SWRL rules:** SWRL rules are used to automatically assign Flavors to Behaviors through reasoning. The user can add custom SWRL rules to their application ontology to automatically establish when certain Behaviors will have a Flavor. As an example, Flavor 1 of Table I, delaying opening the end effector until the robot is at the pregrasp position, can be applied to any Behavior that has both `open` and `go_to_pregrasp` as sub-Behaviors. Figure 9 shows this SWRL rule.

F. Implementation

The proposed approach was implemented as a ROS 2 package part of the BE-AWARE [47] framework, containing the Knowledge Manager and the Execution Configuration Module described in this paper. The modules use the Owlready2 library to manipulate and query the ontologies. A smart perception module is integrated to create instances of classes defined in the Application Ontology of the objects perceived by the sensors [57]. The BehaviorTree.CPP (https://www.behaviortree.dev/) and BehaviorTree.ROS2 (https://github.com/BehaviorTree/BehaviorTree.ROS2) libraries are used to execute the Behavior Tree. BehaviorTree.ROS2 provides the wrappers to implement the Behavior Tree nodes as ROS 2 actions and service clients. This allows to integrate executions by representing each Primitive of the user as a call to a ROS 2 action or service server. Motion Planning queries are done using the Kautham Project (TKP, [58]), a motion planning tool based on the Open Motion Planning Library (OMPL, [59]) that offers motion planning and geometric reasoning ROS services.

The BE-AWARE Execution Configuration Module implemented provides the following non-functional Primitives:

- Setting and modifying execution parameters.
- Querying or updating properties from the Knowledge base.
- Querying a Flavor Trigger.
- Generating PDDL planning files from ontological knowledge [48].
- Calling the Task Planner, both at the start and when replanning is needed.
- Calling for the Execution Configuration pipeline to be initiated after a new Task Plan is obtained.

V. EXPERIMENTAL EVALUATION

A. Scenario description

To demonstrate the different capabilities of the proposed Execution Configuration approach, an illustrative example has been defined. The Tiago robot is considered in a kitchen scenario, a domain that has a wide variety of objects and manipulation actions and where disruptions are common. It is an ideal scenario to show how Flavors can define more expressive Behaviors as well as the rest of the features described previously. The kitchen scenario is simulated using the Gazebo simulator (Figure 10).

The general task goal is to clear the objects on the table, putting them in the sink if they are dirty or on the shelf if not. Modifying the types of objects, their starting and ending locations, and the features of the robot (number of arms, available tools) requires adaptations to the Behavior Tree.

A specific Application Ontology for this example was created, including the classes in the kitchen environment (`Glass`, `Dish`, `Knife`, etc.) and their properties (`Glass is a LiquidContainer`, `Knife is a DangerousObject`, etc.). A sample of the object classes and subclasses featured in the scenario can be seen in Figure 11a. Locations are also assigned specific properties, such as categorizing them into those that require the Tiago torso to be lowered (e.g., `Dishwasher`) or raised (e.g., `Shelf`) to access. Ontological reasoning is used to infer additional properties, such as establishing that all the individuals with `hasMaterial glass` are classified as `FragileObject`. This Application Ontology is constant in the different tested scenarios, but the individuals of those classes and their properties are changed in each. The Application Ontology used for these examples is formed of around 1000 axioms, with some additions depending on the scenario.

A Template Library was created with the Plan-Level Behaviors of `Pick`, `Place` and `Move` as starting points, and with several sub-Behaviors and primitives under them. All these sub-Behaviors and primitives are represented in the Knowledge Base, as discussed in Section IV-A. The integration within the BE-AWARE framework of the PDDL domain and problem files with the Ontological Knowledge falls under the (Re)Planning Design module [48]. In this example, the PDDL Domain file has been designed manually, ensuring compatibility with the Application Ontology. The knowledge files for this example have been made available to the public [60].

A `Pick` Behavior is used as a representative example to illustrate the Execution Configuration module applying Flavors and analyze the overhead required. The original Base Template is shown in Figure 12a. For this example, the Behavior has been simplified, and recovery or alternative sub-Behaviors that would be executed in case the Condition Nodes return `FAILURE` have not been included.

A representative sample of Flavors of different Flavor Types in the Flavor Library has been chosen to be applied to the Behaviors available in the Base Template, gathered in Table III (the Flavors have been numerated to refer to them). Figure 11b

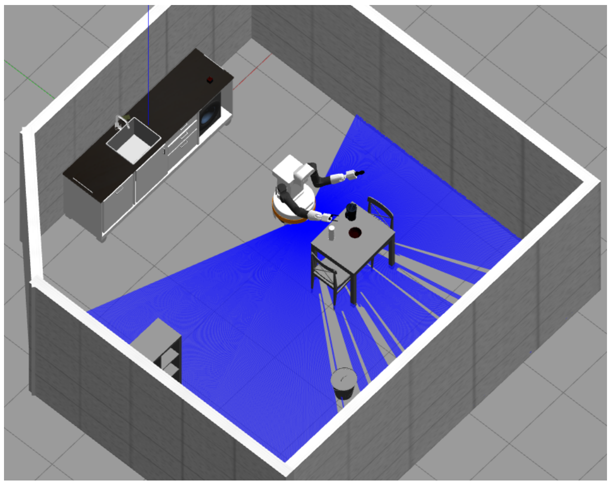
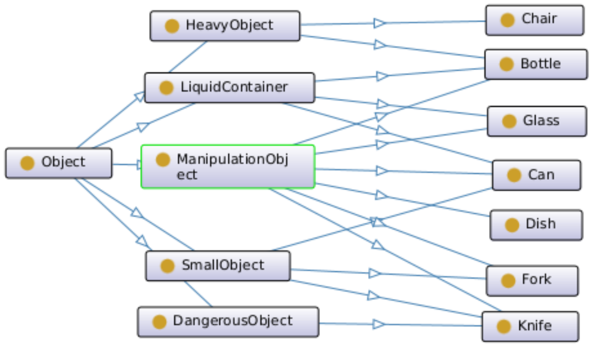


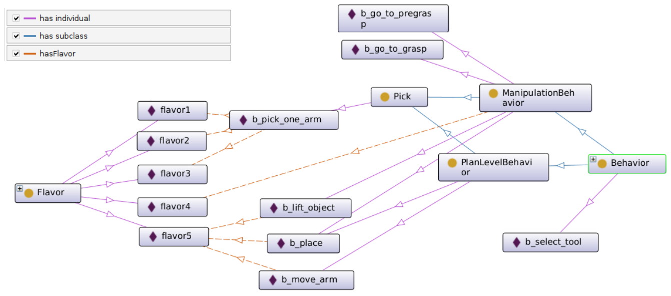
Fig. 10: Tiago robot in a simulated kitchen scenario in Gazebo.

#	Behavior	Flavor Keyword	Flavor Description
1	Pick	Delay A until after B	*If the robot has a hand, do not open it until GoToPregrasp has been done*
2	Pick	Ignore A	*Skip going to the pregrasp position if the object is small; go directly to grasp it*
3	Pick	Replace By Param	*If the robot only has one arm and end-effector, replace the Select Tool behavior by directly setting the only tool available in the BT*
4	Manipulation Behavior	Modulate Param	*If the object is dangerous, reduce velocity by 30%*
5	Lift, Move, Place	Add Kinematic Restriction to A	*If the target object is a container filled with a liquid, such as a glass, add the Kinematic restriction that it must be held upright*

TABLE III: Flavors applied to the Behaviors in the example.



(a) Object classes and subclasses in the Application Ontology.



(b) Behaviors and Flavors in the Application Ontology.

Fig. 11: Representative sample of classes and individuals featured in the Application Ontology for the kitchen scenario.

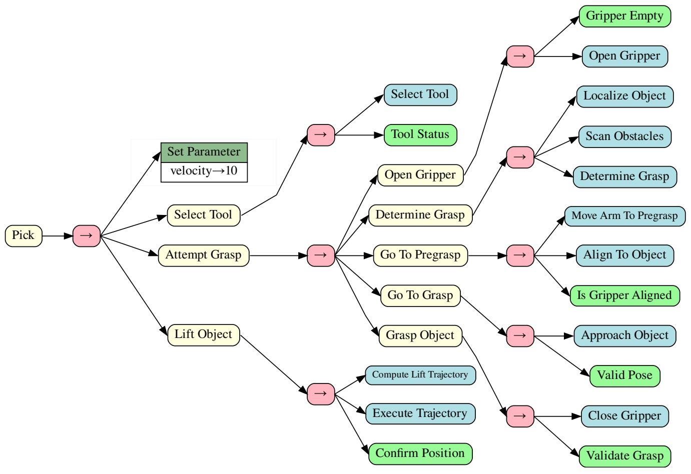
illustrates the ontological representation of these Flavors and the Behaviors they are applied to.

The fully expanded `Pick` Base Template shown in Figure 12a is formed of 8 sub-Behaviors: `Select Tool`, `Attempt Grasp`, `Lift Object`, `Open Gripper`, `Determine Grasp`, `Go To Pregrasp`, `Go to Grasp`, and `Grasp Object`. Each has its own template and ontological representation, following Figure 4, but as they are not Plan-Level Behaviors, they lack a PDDL representation. Figure 12b shows Flavors 1 to 5 applied to this Base Template, indicating in red the modifications that have been made. It is important to remark that these Flavors would not necessarily be active at once; it depends on the results of the triggers, which will depend on the Seed. Hard-coding the equivalent of these five Flavors into the generation of this `Pick` Behavior Tree is possible, as done in our previous work [12], but it would involve having a corresponding amount of templates for the same skill, incorporating these modifications with different combinations, and having a selection layer that depends on the triggers. The design of such trees with several variations gets more and more complex with each required modification. For illustrative purposes, a video showing the execution of these two BTs is provided at https://youtu.be/wofRBYOpXsw and in the repository [60].

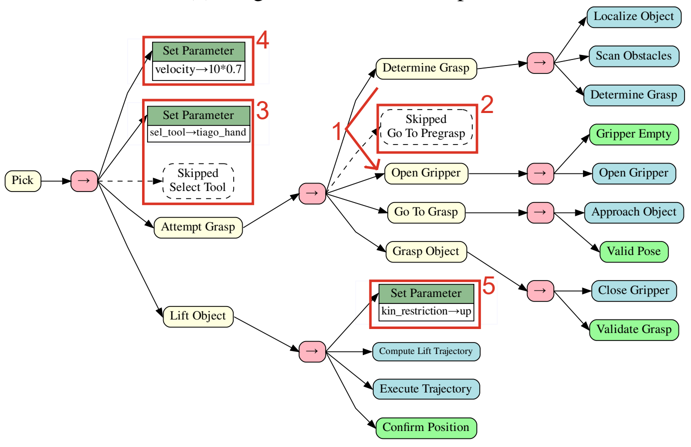
B. Performance Analysis

From the execution time perspective, the application of Flavors does not necessarily result in a direct, quantifiable improvement in Behavior execution. While certain adaptations encoded within Flavors may reduce execution time (for instance, by eliminating redundant actions), many of their potential advantages are qualitative in nature. These include facilitating safer Behavior execution or restructuring the sequence of actions according to non-performance-related criteria. Accordingly, the extent to which Flavors provide measurable benefits is dependent on the specific application domain and the evaluation metrics employed.

The integration of the Execution Configuration Module introduces additional planning overhead that has to be analyzed. Figure 13 presents the measured times required to perform Execution Configuration for the `Pick` Behavior. The baseline corresponds to loading the Base Template without any modifications, as in previous work [12]. Six additional scenarios were considered: one for each of the five Flavors and a final scenario combining all the Flavors. The Execution Configuration Pipeline described in Figure 5 and Algorithm 1 was applied in these cases. For each scenario, two Seeds were tested: one resulting in trigger values that return False, the



(a) Original Pick Base Template



(b) Pick configured with Flavors

Fig. 12: BT representations of an unconfigured and configured Pick. Yellow nodes represent Behaviors, blue nodes represent Action Nodes, and green nodes represent Condition Nodes.

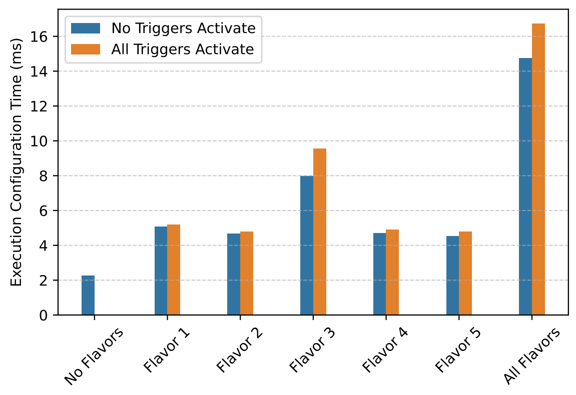


Fig. 13: Execution Configuration times for the `Pick` example. Flavor 3 exhibits the largest difference between the two cases, as it includes a SPARQL query within the Flavor itself.

other with trigger values that return True. The difference in execution configuration time between these two cases reflects the time required to apply the Flavor and modify the *.xml* template using the Flavor Library.

From the results, several conclusions can be drawn. First, even when a Flavor is not applied for a given Seed, a full traversal of the BT is still required, leading to an increase in execution time relative to the baseline. Second, executing SPARQL queries contributes noticeably to the overall time. Flavor 3 exemplifies this: the first SPARQL query used for the trigger counts the number of arms on a robot, increasing evaluation time compared to the other triggers. Then, the Flavor itself requires a second query to retrieve the tool name for parameter replacement, explaining why Flavor 3 exhibits the largest difference between the False and True trigger cases. Although SPARQL execution in Python is slower than in some other languages [61], the capabilities provided by the Owlready2 library for ontology management make it an acceptable choice for this module.

Importantly, the execution configuration time in this representative example remains very low, around 16 ms for a moderately sized Base Template. While more complex templates or Flavors with additional SPARQL queries would increase this time, the current results indicate there is ample margin to increase complexity before execution time would become problematic. This demonstrates that the pipeline can efficiently handle moderate template and query complexity without significant performance concerns. In the context of high-level task planning, this small overhead is an acceptable trade-off.

The main performance limitation observed in these experiments arises from the loading and reasoning of the ontology. Loading and reasoning over the Application Ontology used in these examples requires an average of 750 ms. Since Execution Configuration depends on the most up-to-date reasoned knowledge, this reasoning time is non-negligible. The module must either wait for a reasoning cycle to be completed or rely on a previously stored snapshot of reasoned knowledge, which introduces slightly outdated information. This snapshot is currently taken periodically after reasoning. This ontological overhead is inherent to the BE-AWARE framework itself and affects all the modules, not just Execution Configuration, the focus of this paper. Nonetheless, reducing this overhead will be addressed in future work, as it is a key bottleneck that affects the whole framework.

VI. Conclusions and Future Work

This work introduced a knowledge-based approach to configure the execution of plans in Behavior Trees. The traditional planning-based approach for Behavior Tree generation, limited by the use of hard-coded templates, has been improved by introducing the concept of Flavors. Flavors enrich the description of Behaviors, allowing Behaviors that can have their Behavior Tree structure adapted to the circumstances in the domain. Flavors can also be used to allow the same Base Templates to be applicable in different domains. The process of configuring the Behavior Tree and applying the Flavors is doable at runtime, with minimal additional time required. At runtime, a new Behavior Tree can be rebuilt when the execution of the original tree encounters an unforeseen disturbance that requires replanning.

The modularity of Behavior Trees and the ontological description allow for flexibility in the kinds of Behaviors that can work with one another. The proposed approach can configure the execution of a plan involving manipulation, perception, navigation, monitoring and recovery into a Behavior Tree, adapted to the scenario through the use of reasoning and applying the Flavors. We presented a proof of concept of different kinds of Flavors that can be applied to typical manipulation scenarios in robotics. The implementation is intended to facilitate the expansion to new Flavors.

The use of Flavors has the following advantages:

- Consistency in applying the modifications to the Base Templates. In the proposed approach, all the Flavors of the same type are applied equally. This allows for better Execution Awareness as the resulting execution structure is fully configured by the framework.
- Scalability in the amount of modifications applied to the Base Templates. Pre-designing all the scenarios for the desired modifications and their resulting BT structure is manageable when dealing with a few cases, but can quickly grow to cumbersome amounts. The proposed approach allows for quicker and easier creation and assigning of Flavors to Behaviors.
- Generality in Behavior descriptions. The proposed approach allows the same Base Templates to be used across different domains, and the only modifications necessary to adapt to them would be to design a different assortment of Flavors.

A limitation of the current method is the reliance on available knowledge for checking Triggers, when it is possible that parts of this knowledge is only known at execution time of a Behavior. To address this, future work will explore the use of reasoning to detect the Triggers that need to be checked as part of the Behavior Tree, and how Flavors can be applied in those cases. Also, future work will include expanding the Flavor keywords available, and developing improvements in how they are applied, such as improving how multiple Flavors interact with each other. The application of optimization algorithms [9] [42] after applying the Flavors is also a promising direction to further refine the Execution Configuration capabilities.

In the context of the BE-AWARE framework, future work involves developing specific Behaviors and Flavors for smart monitoring and recovery, integrating them into Behavior Trees to enable robust task execution in partially observable environments, which require integrating both sensing and acting into the BT structure [46]. Finally, improvements over ontology reasoning and querying will be investigated to improve performance of the overall framework.

Building on the current proposal and future developments, the BE-AWARE framework aims to significantly enhance the adaptive and robust execution of robotic manipulation tasks, further increasing the autonomy of robotic systems.

GLOSSARY

This glossary summarizes the key concepts used throughout this paper. It is intended to provide concise definitions for reference and to ensure consistent understanding of terminology.

- **Execution Configuration:** The process of transforming a high-level task plan into an executable Behavior Tree adapted to the current scenario.
- **Behavior:** A structured composition of Primitives and possibly other Behaviors that represents a meaningful unit of execution beyond an indivisible action.
- **Base Template:** A generic Behavior Tree structure associated with a Behavior, used as the starting point for Execution Configuration.
- **Flavor:** A modifier that adapts a Behavior by changing its Base Template when its trigger condition is satisfied. Different types of Flavors exist in the Flavor library.
- **Seed:** A representation that specifies which concrete entities in the Knowledge Base relate to the Behavior instance being configured.
- **Trigger:** A knowledge SPARQL query that determines whether a Flavor is applied to a Behavior in a given context.

DECLARATIONS

Author Contributions All authors contributed to the study conception and design. The first draft of the manuscript was written by Oriol Ruiz-Celada and all authors commented on previous versions of the manuscript. All authors read and approved the final manuscript.

Funding This work was supported by the European Commission's Horizon Europe Framework Programme with the project AI-Powered Manipulation System for Advanced Robotic Service, Manufacturing and Prosthetics (IntelliMan) under Grant Agreement 101070136.

Availability of data and materials All data generated or analyzed during this work are included in this published article and the complementary repository [60]. The source codes used during the research are available from the corresponding author on reasonable request.

Competing Interests The authors have no conflicts of interest to declare that are relevant to the content of this manuscript.

Ethics approval This research did not involve human participants or animals and did not require ethical approval.

Consent to participate Not applicable.

Consent for publication Not applicable.

REFERENCES

[1] D. S. Nau, M. Ghallab, and P. Traverso, "Blended planning and acting: Preliminary approach, research challenges," *Proceedings of the AAAI Conference on Artificial Intelligence*, vol. 29, pp. 4047–4051, 3 2015. [Online]. Available: https://ojs.aaai.org/index.php/AAAI/article/view/9768

[2] S. Yang, X. Mao, S. Wang, H. Xiao, and Y. Xue, "Towards adjoint sensing and acting schemes and interleaving task planning for robust robot plan," *Proceedings - IEEE International Conference on Robotics and Automation*, vol. 2021-May, pp. 398–404, 2021.

[3] A. Kattepur and B. Purushotaman, "RoboPlanner: a pragmatic task planning framework for autonomous robots," *Cognitive Computation and Systems*, vol. 2, pp. 12–22, 3 2020.

[4] M. Ghallab, D. Nau, and P. Traverso, "The actor's view of automated planning and acting: A position paper," *Artificial Intelligence*, vol. 208, pp. 1–17, 3 2014.

[5] M. Hölzl and T. Gabor, "Reasoning and learning for awareness and adaptation," *Lecture Notes in Computer Science*, vol. 8998, pp. 249–290, 2015.

[6] M. Ben-Ari and F. Mondada, *Finite State Machines*. Springer International Publishing, 2018, pp. 55–61. [Online]. Available: https://doi.org/10.1007/978-3-319-62533-1_4

[7] T. Murata, "Petri nets: Properties, analysis and applications," *Proceedings of the IEEE*, vol. 77, no. 4, pp. 541–580, 1989.

[8] M. Colledanchise and P. Ögren, *Behavior Trees in Robotics and AI* . CRC Press, 7 2018.

[9] F. Rovida, B. Grossmann, and V. Kruger, "Extended behavior trees for quick definition of flexible robotic tasks," *IEEE International Conference on Intelligent Robots and Systems*, vol. 2017-September, pp. 6793–6800, 12 2017.

[10] M. Mayr, F. Rovida, and V. Krueger, "SkiROS2: A Skill-Based Robot Control Platform for ROS," in *2023 IEEE/RSJ International Conference on Intelligent Robots and Systems (IROS)*, 2023, pp. 6273–6280.

[11] M. Beetz, G. Kazhoyan, and D. Vernon, "Robot manipulation in everyday activities with the CRAM 2.0 cognitive architecture and generalized action plans," *Cognitive Systems Research*, vol. 92, p. 101375, 2025. [Online]. Available: https://www.sciencedirect.com/science/article/pii/S1389041725000555

[12] O. Ruiz-Celada, P. Verma, M. Diab, and J. Rosell, "Automating adaptive execution behaviors for robot manipulation," *IEEE Access*, vol. 10, pp. 123 489–123 497, 2022.

[13] G. Kazhoyan, A. Niedzwiecki, and M. Beetz, "Towards plan transformations for real-world pick and place tasks," 2018. [Online]. Available: https://doi.org/10.48550/arXiv.1812.08226

[14] J. Styrud, M. Iovino, M. Norrlöf, M. Björkman, and C. Smith, "Combining planning and learning of behavior trees for robotic assembly," in *2022 International Conference on Robotics and Automation (ICRA)*, 2022, pp. 11 511–11 517.

[15] A. Al-Moadhen, R. Qiu, M. Packianather, Z. Ji, and R. Setchi, "Integrating robot task planner with common-sense knowledge base to improve the efficiency of planning," *Procedia Computer Science*, vol. 22, pp. 211–220, 1 2013.

[16] S. Wang, Y. Zhang, and Z. Liao, "Review on the knowledge graph in robotics domain," pp. 424–431, 7 2019. [Online]. Available: https://www.atlantis-press.com/proceedings/iccia-19/125913157

[17] P. Pirnay-Dummer, D. Ifenthaler, and N. M. Seel, *Knowledge Representation*. Boston, MA: Springer US, 2012, pp. 1689–1692. [Online]. Available: https://doi.org/10.1007/978-1-4419-1428-6_875

[18] M. Beetz, "Knowledge Representation and Reasoning," in *Cognitive Robotics*. The MIT Press, 05 2022. [Online]. Available: https://doi.org/10.7551/mitpress/13780.003.0027

[19] R. Bernardo, J. M. Sousa, and P. J. Gonçalves, "A novel framework to improve motion planning of robotic systems through semantic knowledge-based reasoning," *Computers & Industrial Engineering*, vol. 182, p. 109345, 8 2023.

[20] T. Höbert, W. Lepuschitz, M. Vincze, and M. Merdan, "Knowledge-driven framework for industrial robotic systems," *Journal of Intelligent Manufacturing*, vol. 34, 3 2021.

[21] S. Staab and R. Studer, Eds., *Handbook on Ontologies*. Springer Berlin Heidelberg, 2009.

[22] W3C, "OWL 2 Web Ontology Language Document Overview (Second Edition)." [Online]. Available: https://www.w3.org/TR/owl2-overview/

[23] I. Horrocks, "OWL: A description logic based ontology language," *Lecture Notes in Computer Science*, vol. 3709 LNCS, pp. 5–8, 2005.

[24] E. Sirin, B. Parsia, B. C. Grau, A. Kalyanpur, and Y. Katz, "Pellet: A practical OWL-DL reasoner," *Journal of Web Semantics*, vol. 5, no. 2, pp. 51–53, 2007, software Engineering and the Semantic Web. [Online]. Available: https://www.sciencedirect.com/science/article/pii/S1570826807000169

[25] B. Glimm, I. Horrocks, B. Motik, G. Stoilos, and Z. Wang, "HermiT: An OWL 2 Reasoner," *Journal of Automated Reasoning*, vol. 53, no. 3, pp. 245–269, Oct 2014. [Online]. Available: https://doi.org/10.1007/s10817-014-9305-1

[26] M. A. Musen and Protégé Team, "The Protégé project: A look back and a look forward," *AI Matters*, vol. 1, no. 4, pp. 4–12, Jun. 2015.

[27] N. Guarino, "Formal Ontology in Information Systems: Proceedings of the 1st International Conference June 6-8, 1998, Trento, Italy." NLD: IOS Press, 1998.

[28] I. Niles and A. Pease, "Towards a standard upper ontology," in *Proceedings of the International Conference on Formal Ontology in Information Systems - Volume 2001*, ser. FOIS '01. New York, NY, USA: Association for Computing Machinery, 2001, p. 2–9.

[29] "IEEE Standard Ontologies for Robotics and Automation," *IEEE Std 1872-2015*, pp. 1–60, 2015.

[30] "IEEE Standard for Autonomous Robotics (AuR) Ontology," *IEEE Std 1872.2-2021*, pp. 1–49, 2022.

[31] A. Olivares-Alarcos, D. Beßler, A. Khamis, P. Goncalves, M. Habib, J. Bermejo-Alonso, M. Barreto, M. Diab, J. Rosell, J. Quintas, J. Olszewska, H. Nakawala, E. Pignaton, A. Gyrard, S. Borgo, G. Alenyà, M. Beetz, and H. Li, "A review and comparison of ontology-based approaches to robot autonomy," *The Knowledge Engineering Review*, vol. 34, 2019.

[32] S. H. Joo, S. Manzoor, Y. G. Rocha, S. H. Bae, K. H. Lee, T. Y. Kuc, and M. Kim, "Autonomous Navigation Framework for Intelligent Robots Based on a Semantic Environment Modeling," *Applied Sciences 2020, Vol. 10, Page 3219*, vol. 10, p. 3219, 5 2020.

[33] I. Horrocks, P. F. Patel-Schneider, H. Boley, S. Tabed, B. Grosof, and M. Dean, "SWRL: A Semantic Web Rule Language Combining OWL and RuleML," 2004. [Online]. Available: https://www.w3.org/Submission/SWRL/

[34] M. Mayr, K. Chatzilygeroudis, F. Ahmad, L. Nardi, and V. Krueger, "Learning of parameters in behavior trees for movement skills," in *2021 IEEE/RSJ International Conference on Intelligent Robots and Systems (IROS)*. IEEE, Sep. 2021. [Online]. Available: http://dx.doi.org/10.1109/IROS51168.2021.9636292

[35] Y. Wu, J. Li, H. Jin, J. Zhang, and Y. Wang, "RBT-HCI: A Reliable Behavior Tree Planning Method with Human-Computer Interaction," *2022 IEEE International Conference on Robotics and Biomimetics (ROBIO)*, pp. 1637–1642, 12 2022.

[36] 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, 8 2022.

[37] B. Banerjee, "Autonomous acquisition of behavior trees for robot control," *IEEE International Conference on Intelligent Robots and Systems*, pp. 3460–3467, 12 2018.

[38] M. Colledanchise, R. Parasuraman, and P. Ögren, "Learning of Behavior Trees for Autonomous Agents," 4 2015.

[39] Y. Cao and C. S. G. Lee, "Robot Behavior-Tree-Based Task Generation with Large Language Models," in *Proceedings of the AAAI 2023 Spring Symposium on Challenges Requiring the Combination of Machine Learning and Knowledge Engineering (AAAI-MAKE 2023)*, 2023. [Online]. Available: https://ceur-ws.org/Vol-3433/paper4.pdf

[40] J. Styrud, M. Iovino, M. Norrlöf, M. Björkman, and C. Smith, "Automatic Behavior Tree Expansion with LLMs for Robotic Manipulation," in *2025 IEEE International Conference on Robotics and Automation (ICRA)*, 2025, pp. 1225–1232.

[41] M. Mayr, F. Ahmad, K. Chatzilygeroudis, L. Nardi, and V. Krueger, "Combining planning, reasoning and reinforcement learning to solve industrial robot tasks," 12 2022.

[42] F. Martín Rico, M. Morelli, H. Espinoza, F. J. Rodríguez-Lera, and V. Matellán Olivera, "Optimized execution of PDDL plans using behavior trees," in *Proceedings of the 20th International Conference on Autonomous Agents and MultiAgent Systems*, ser. AAMAS '21. Richland, SC: International Foundation for Autonomous Agents and Multiagent Systems, 2021, p. 1596–1598.

[43] M. Ghallab, C. Knoblock, D. Wilkins, A. Barrett, D. Christianson, M. Friedman, C. Kwok, K. Golden, S. Penberthy, D. Smith, Y. Sun, and D. Weld, "PDDL - The Planning Domain Definition Language," 08 1998.

[44] M. Colledanchise, D. Almeida, and P. Ögren, "Towards Blended Reactive Planning and Acting using Behavior Trees," 11.

[45] F. Rovida, D. Wuthier, B. Grossmann, M. Fumagalli, and V. Krüger, "Motion generators combined with behavior trees: A novel approach to skill modelling," in *2018 IEEE/RSJ International Conference on Intelligent Robots and Systems (IROS)*, 2018, pp. 5964–5971.

[46] S. Yang, X. Mao, S. Wang, H. Xiao, and Y. Xue, "Towards adjoint sensing and acting schemes and interleaving task planning for robust robot plan," in *2021 IEEE International Conference on Robotics and Automation (ICRA)*, 2021, pp. 13 791–13 797.

[47] O. Ruiz-Celada, A. Dalmases, R. Suárez, and J. Rosell, "BE-AWARE: an ontology-based adaptive robotic manipulation framework," in *2023 IEEE 28th International Conference on Emerging Technologies and Factory Automation (ETFA)*, 2023, pp. 1–4.

[48] V. Molina, O. Ruiz-Celada, R. Suarez, J. Rosell, and I. Zaplana, "Robot Situation and Task Awareness Using Large Language Models and Ontologies," in *2025 55th Annual IEEE/IFIP International Conference on Dependable Systems and Networks Workshops (DSN-W)*, 2025, pp. 96–103.

[49] J. Hoffmann and B. Nebel, "The FF Planning System: Fast Plan Generation Through Heuristic Search," *Journal of Artificial Intelligence Research*, vol. 14, p. 253–302, May 2001. [Online]. Available: http://dx.doi.org/10.1613/jair.855

[50] "IEEE Standard for Autonomous Robotics (AuR) Ontology," *IEEE Std 1872.2-2021*, pp. 1–49, 2022.

[51] J.-B. Lamy, "Owlready: Ontology-oriented programming in python with automatic classification and high level constructs for biomedical ontologies," *Artificial Intelligence in Medicine*, vol. 80, pp. 11–28, 2017. [Online]. Available: https://www.sciencedirect.com/science/article/pii/S0933365717300271

[52] W3C, "SPARQL 1.1 Overview." [Online]. Available: https://www.w3.org/TR/2013/REC-sparql11-overview-20130321/

[53] F. Ingrand and M. Ghallab, "Deliberation for autonomous robots: A survey," *Artificial Intelligence*, vol. 247, pp. 10–44, 2017. [Online]. Available: www.elsevier.com/locate/artint

[54] M. E. Cansev, H. Xue, N. Rottmann, A. Bliek, L. E. Miller, E. Rueckert, and P. Beckerle, "Interactive human–robot skill transfer: A review of learning methods and user experience," *Advanced Intelligent Systems*, vol. 3, no. 7, p. 2000247, 2021. [Online]. Available: https://onlinelibrary.wiley.com/doi/abs/10.1002/aisy.202000247

[55] V. Kruger, D. L. Herzog, S. Baby, A. Ude, and D. Kragic, "Learning actions from observations," *IEEE Robotics & Automation Magazine*, vol. 17, no. 2, pp. 30–43, 2010.

[56] S. Bøgh, O. Nielsen, M. Pedersen, V. Krüger, and O. Madsen, "Does your robot have skills?" in *Proceedings of the 43rd International Symposium on Robotics*. VDE Verlag GMBH, Aug. 2012.

[57] O. Ruiz-Celada, A. Dalmases, I. Zaplana, and J. Rosell, "Smart perception for situation awareness in robotic manipulation tasks," *IEEE Access*, vol. 12, pp. 53 974–53 985, 2024.

[58] J. Rosell, A. Pérez, A. Aliakbar, Muhayyuddin, L. Palomo, and N. García, "The kautham project: A teaching and research tool for robot motion planning," in *Proceedings of the 2014 IEEE Emerging Technology and Factory Automation (ETFA)*, 2014, pp. 1–8.

[59] I. A. Şucan, M. Moll, and L. E. Kavraki, "The Open Motion Planning Library," *IEEE Robotics & Automation Magazine*, vol. 19, no. 4, pp. 72–82, December 2012, https://ompl.kavrakilab.org.

[60] O. Ruiz Celada, J. Rosell, and R. Suárez, "Knowledge-based Execution Configuration for Adaptive Behavior Trees," Dec. 2025. [Online]. Available: https://doi.org/10.5281/zenodo.17952916

[61] M. Bamboat, A. Hafeez, and A. Wagan, "Performance of RDF Library of Java, C# and Python on Large RDF Models," *International Journal on Emerging Technologies*, pp. 25–30, 01 2021.

Oriol Ruiz-Celada received the double master's degree in Industrial Engineering and Automatic Control and Robotics from the Universitat Politècnica de Catalunya (UPC), Barcelona, in 2022. He is currently pursuing his Ph.D. in Automatic Control, Robotics and Computer Vision at UPC, working in the Institute of Industrial and Control Engineering (IOC). His research interests include task and motion planning, knowledge representation and reasoning, artificial intelligence, and robotic manipulation.

Jan Rosell received the B.S. degree in telecommunication engineering and the Ph.D. degree in advanced automation and robotics from Universitat Politècnica de Catalunya (UPC), Barcelona, Spain, in 1989 and 1998, respectively. He joined the Institute of Industrial and Control Engineering (IOC), in 1992, where he has developed research activities in robotics. He has been involved in teaching activities in automatic control and robotics as an Assistant Professor, since 1996, and an Associate Professor, since 2001. His current technical areas include task and motion planning, mobile manipulation, and robot coworkers.

Raúl Suárez received the degree in electronic engineering from the National University of San Juan, San Juan, Argentina, in 1984 and the Ph.D. degree from the Universitat Politècnica de Catalunya (UPC), Barcelona, Spain, in 1993. His current position is Research Supervisor with the Institute of Industrial and Control Engineering (IOC) at UPC. He was responsible for the research line on process control at the IOC from 1998 to 2003, Deputy Director of the IOC from 2003 to 2009 and Director from 2009 to 2016, and, from 2016, he is the responsible of the Division of Robotics of the IOC. He has been the coordinator of a Ph.D. Program on robotics at UPC from 1995 to 2023, and President of the IEEE RAS Spanish Chapter from 2019 to 2023. He is coauthor of about 100 scientific papers published in international conference proceedings and journals, and he has participated in 27 competitive research projects (National and European) being principal investigator in 12 of them. His research activity is mainly focused on intelligent robotics, particularly in the areas of motion planning, human-like movements, robotized grasping and dexterous manipulation.