Failure Recovery with Ontologically Generated Behaviour Trees

MSc. Thesis

Wissam Jaber

Delft University of Technology

TU Delft

Failure Recovery with Ontologically Generated Behaviour Trees

MSc. Thesis by

Wissam Jaber

Student Name	Student Number
Wissam Jaber	5156432

In order to obtain the degree of Master of Robotics
at the Delft University of Technology,
To be defended publicly on Friday September 22, 2023 at 3:00 PM.

Thesis committee: Dr. ir. C. Hernández Corbato
Dr. L. Peternel
Dr. C. Pek

An electronic version of this thesis is available at `http://repository.tudelft.nl/`.

TU Delft

Acknowledgement

I would like to express my gratitude to God for guiding me through this incredible journey and providing me with the strength and patience to overcome challenges along the way.
To my family and friends, your unwavering support, encouragement, and understanding have been my pillars of strength. Your belief in me has been a constant source of motivation, and I'm grateful for the love and laughter you've brought into my life.
I extend my heartfelt thanks to my supervisor, Carlos Hernandez Corbato, for his guidance, patience, and mentorship throughout this year. Your insights and expertise have been invaluable. I also want to acknowledge the remarkable individuals at AIRLab. Your camaraderie, knowledge-sharing, and collaborative spirit have enriched my research experience in countless ways. Lastly, I would like to convey my appreciation to the committee members for dedicating their time to their students even beyond their job description.
This journey has been challenging yet immensely rewarding, and I'm grateful to each and every one of you for being a part of it.

Wissam Jaber
Delft, September 2023

Failure Recovery with Ontologically Generated Behaviour Trees

Wissam Jaber[1]

***Abstract*— Behaviour trees (BTs) serve as a powerful hierarchical structure for task execution, simplifying complex tasks but posing challenges in their manual design. The automatic generation of BTs addresses this concern, yet often lacks robust failure recovery options. This study presents Failure Recovery with Ontologically Generated Behaviour Trees (FROGBT), a novel approach bridging this gap by integrating ontological reasoning into the process of automatically generating BTs. This integration establishes a profound link between an agent's knowledge and its capabilities, offering contextual insights into the agent's skills. FROGBT enhances skill representation for planning and recovery. The approach's effectiveness is indicated by its efficiency compared to the state-of-the-art framework for skill-based control, SkiROS, in a similar task. It showcases generality, uniting diverse skills, developed by various engineers, for recurring tasks, and introduces innovative failure recovery strategies. FROGBT highlights ontological reasoning's potential to enhance BT generation with context-awareness and reasoning abilities, paving the way for future research in failure recovery concepts in generated BTs.**

Keywords: Behaviour Trees, Ontological Reasoning, Failure Recovery, Fallback Branches, Behaviour Generation.

1 INTRODUCTION

Modern intelligent robots are facing increasingly complex tasks, necessitating correspondingly sophisticated system architectures for programming and control. Achieving greater autonomy requires a matching complexity in the underlying structure. Therefore, there is a growing demand for modularity, precise control, reactivity, and task planning mechanisms.

Behaviour trees (BTs) have emerged as a valuable tool for task execution due to their hierarchical structure, enabling streamlined coordination of intricate tasks in a modular fashion. They excel in breaking down complex sequences into manageable units, serving as an intuitive representation for implementing behaviours across autonomous systems. Widely applicable, BTs find their utility across different fields, from video game development [1] to robotics [2]. However, the manual design of BTs presents formidable challenges. Constructing effective BTs demands a deep understanding of task intricacies, behaviour interactions, and potential outcomes at each step. This process is time-consuming, prone to errors, and heavily reliant on human expertise. Scalability quickly becomes a concern, particularly in dynamic real-world scenarios.

As a remedy, methods for generating BTs have been explored by using classical planners such as Planning Domain Definition Language (PDDL)[3], Hierarchical Task Network (HTN)[4] or search algorithms. However, the ability of generated BTs to cope with failures has often been found limited. Existing mechanisms either demand resource-intensive re-planning or rely on predetermined scripted fallback behaviours, constraining adaptability and system reliability. This gap calls for more efficient and dependable BT generating methods for intricate applications.

To tackle these challenges, we introduce Failure Recovery with Ontologically Generated Behaviour Trees (FROGBT) approach. This method effectively blends ontological reasoning with BT creation, bridging the divide between generated BTs and failure recovery generation. FROGBT offers an inventive solution that infuses context-awareness into the planning and execution of intricate tasks. By incorporating fallback branches and behaviour recovery generation, this approach enhances recovery strategies and avoids expensive re-planning.

In the subsequent sections, we delve into the intricacies of FROGBT and its methodologies. We illustrate its effectiveness through motivating scenarios and comprehensive evaluations. Finally, we share some insightful conclusions and future directions. We aim to contribute to the advancement of BT generation and failure recovery by harnessing the power of ontological reasoning.

2 BACKGROUND

This section will briefly introduce a background of

[1]Delft University of Technology, Email address: w.jaber@student.tudelft.nl

Node	Returns Success	Symbol
Sequence	When all children succeed	$\rightarrow$
Selector	At least one of the children succeeds	?
Parallel	When m number of the children succeed	$\rightrightarrows$
Action Leaf Node	When the action is executed successfully	
Condition Leaf Node	When the condition is satisfied	
Decorator	Depends on the decorator	

Table 1: Information about node types of BTs, when the node returns success and symbol

the two core concepts of FROGBT: behaviour trees and ontological reasoning.

2.1 Behaviour Trees

Behaviour Trees (BTs) are a versatile and widely used approach for representing and controlling the behaviour of autonomous systems, including robots. BTs provide a structured and hierarchical way of orchestrating complex sequences of actions, enabling robots to perform tasks in a modular and organised manner. This hierarchical structure of BTs consists of various types of nodes that represent behaviours and govern the order in which these behaviours should be executed. Table 1 summarises the type of BT nodes.

BTs have become particularly popular in robotics due to their intuitive representation of complex tasks and behaviours. Their hierarchical structure allows engineers to design behaviours at different levels of granularity, promoting modularity and ease of maintenance. BTs are especially effective when dealing with tasks that require both reactive and deliberative decision-making, as they allow for the easy integration of different levels of control.

2.2 Ontological Reasoning

Ontological reasoning is a fundamental concept in knowledge representation. It involves organising knowledge in a structured manner using classes and instances. In this context, a class is a general category or type of object, and an instance is a specific individual belonging to that category. In an ontology, there are:

- Classes: They define the characteristics and properties shared by a group of similar objects. They represent categories or concepts. For example, *"Skills"* is a class that defines the skills an agent can perform.
- Instances: They are specific objects that belong to a certain class. Each instance has properties and attributes associated with it. For example, *"Pick"* skill can be an instance of the *"Skills"* class.
- Object Properties: They define relationships between instances of different classes. For instance, an object property *"Precondition"* can relate a *"Skills"* instance with a *"Checks"* instance.
- Data Properties: They link instances with specific data values, such as strings, numbers, dates ... etc. For example, a data property *"Tag"* links a *"Product"* instance with its specific tag for detection purposes.

Ontological reasoning helps in structuring knowledge, enabling more advanced reasoning and decision-making by the agent. It allows for defining relationships, hierarchies, and constraints that enhance the understanding and manipulation of information within various domains, including robotics, where it can assist in tasks like task planning, knowledge sharing, and context-aware behaviour generation.

3 RELATED WORKS

The section provides an encompassing review of related work in the domain of BT generation and failure recovery strategies in generated BTs.

3.1 BTs Generation

The landscape of BT generation methods in existing literature can be outlined based on their adoption of distinct BT design principles, namely sequence root or back-chaining.

The BTs generated using classical planners generally choose a sequence root design for their tree. A plan is composed of a sequence of ordered actions. These plans are subsequently transformed into a BT, with the actions forming the children of a sequence root.

Extended behaviour trees (EBT), as proposed in [5], employ PDDL to generate a plan, and an HTN planner plugin, which decomposes big tasks into smaller sub-tasks. The plan is further evaluated through a series of mathematical rules. These rules are designed to identify tasks that can be executed concurrently and, accordingly, assemble them under a parallel node. EBT's primary emphasis lies in optimising the execution time of generated BTs. Notably, the incorporation of fallback nodes is conspicuously absent in the mathematical model of this approach. Similarly, in [6], a PDDL planner is used to generate a plan. The execution graph of the plan is then used to discern skills that are capable of simultaneous execution and then grouped under parallel parents. Similar to EBT, the consideration of fallback nodes is absent in this approach as well.

However, these methods' reliance on classical planners for generating plans imposes a limitation on the flexibility of BT control nodes, regrettably excluding one of the most crucial control elements related to reactivity: the fallback node. Consequently, BTs generated under this paradigm are often rigid and better suited for controlled environments.

In contrast, constructing a BT using the back-chaining design principle operates through an iterative operation using search algorithms. The process commences with the goal condition as its foundation. This approach extends the tree by systematically appending atomic behaviour trees (AtomicBTs), each tailored to satisfy specific failed conditions. This strategy ensures greater reactivity during execution by capitalising on BT priority concept, left-to-right node execution order, and by prominently employing check leaf nodes to continuously verify post-conditions as essential predicates. These principles prompt constant assessment of actions closer to the goal, facilitating timely branch activation when needed.

Automated planning to generate BTs which was suggested in [7] depends on three lists: pre-conditions, post-conditions, and actions. Each post-condition from the list possesses its dedicated atomicBT. These atomicBTs are crafted as sub-trees, structured with pre-condition checks sequenced before the action to ensure the correct state for the action execution in order to satisfy that post-condition, an example can be seen in Figure 1. This systematic arrangement ensures that the execution of each action occurs within the correct state.

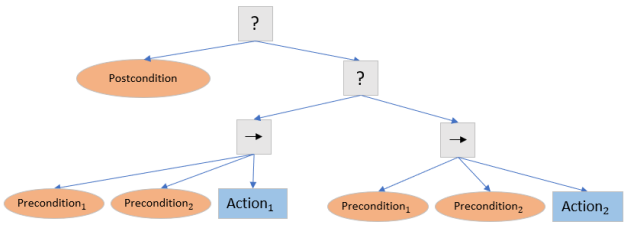


Figure 1: Template for an atomicBT with multiple actions to satisfy the same post-condition

Similarly, in [8] a BT is constructed iteratively following the same principle with the addition of differential logic rules (DL)[9] to ensure each action on the list satisfies a post-condition and each of its pre-conditions has an action that would satisfy it in return. adding a layer of safety of execution to the BT.

Within approaches following the back-chaining design, the utilisation of check leaf nodes and fallback nodes is paramount. This process of generating a BT introduces an additional layer of control, achieved through the manipulation of the tree structure through active planning. Extension of the plan is done as required to attain the desired goal condition. This inherent flexibility, without the necessity for extensive re-planning, is a key attribute of BTs constructed in this manner and the reason this design principle was adopted in the FROGBT approach. This concept allowed FROGBT to make the most out of BT advantages and introduce additional functionalities made achievable by means of continuous tree execution monitoring and the adaptable nature of BTs.

3.2 Failure Recovery Mechanisms

Efforts to rectify plans prior to resorting to re-planning have been the subject of extensive research for a long time. NASA's SIPE1 initiative [10] exemplifies the long-standing interest in this area. While re-planning-based strategies may, in certain instances, yield better quality plans, it is generally acknowledged that recovery approaches offer better response time, according to [11].

Regarding failure recovery within BTs, there exists a main concept represented in fallback

branches, where handcrafted BTs commonly integrate sub-trees tailored for failure recovery purposes, subsequently appended as fallback branches to specific actions. which either guide execution towards a predefined recovery procedure devised by domain experts [12] or reset parameters to facilitate the reattempting of actions from a known state [13].

In the case of generated BTs, methods using classical planners predominantly depend on re-planning as the primary means of addressing failures. Alternatively, some generated BT approaches leverage fallback branches to circumvent re-planning when feasible. Nevertheless, these approaches necessitate manual scripting and integration.

For instance, in [14], handcrafted recovery behaviours tailored to specific actions are inserted into the sequential root BT constructed from a PDDL plan, thereby introducing an additional layer of reactivity.

FROGBT emerges as a structural framework designed to facilitate diverse failure recovery options within generated BTs. It not only enables the utilisation of scripted subtree-based fallback branches but also offers the capability to generate recovery behaviours through ontological reasoning. This augmentation broadens the scope of applicability for failure recovery techniques and presents a stimulating avenue for further exploration. By harnessing the modularity inherent in BTs, FROGBT streamlines engineers' efforts toward skill development, reducing the need for manual construction of BTs and their fallback branches.

3.3 Ontology Integration in BT generation

The integration of ontological reasoning, utilizing structured knowledge to improve logical inference and context comprehension, in BT generation is relatively unexplored, despite its significant potential.

In the realm of robotics, ontology usage for task execution is a young yet promising field of study [15]. Some established frameworks such as CRAM (Cognitive Robot Abstract Machine)[16] emphasise ontological reasoning, albeit without employing behaviour trees.

In contrast, SkiROS (Skill-based Robot Control Platform for ROS) [17] represents a framework that uses the Extended behaviour Tree (EBT) approach for BT generation and task execution. SkiROS employs an automated process to generate problem and domain files for a PDDL planner directly from the ontology. This ontology serves as a hub for information to store environmental, object, and agent parameters, sharing this information across BT nodes. The resulting plan is subsequently translated into a BT, following the sequence root design. SkiROS stands out as an actively developed open-source method, which motivated its selection as a benchmark for comparison with FROGBT.

Conversely, FROGBT goes beyond the conventional use of an ontology as a repository for environmental data. It not only hosts the agent's skills and checks, but also leverages ontological connections to iteratively construct BTs, adopting the back-chaining design principle for both the main task BT and the recovery BT.

This distinctive approach involves continual monitoring to address uncertainties and real-time execution failures. FROGBT's modular BT structure empowers engineers to concentrate on developing individual modular skills rather than intricate fallback branches or BTs. This ontology-driven approach highlights FROGBT's unique position and sheds light on its potential to advance the fields of BT generation and failure recovery.

4 MOTIVATING SCENARIOS

To comprehensively assess FROGBT's capabilities, three distinct motivating scenarios have been designed. These scenarios aim to scrutinise specific aspects of FROGBT's performance, offering valuable insights.

4.1 SkiROS vs FROGBT

This scenario serves as a benchmark comparison between FROGBT and SkiROS. It revolves around the efficiency in planning, while also spotlighting the characteristic design differences in the generated BTs between the two approaches. The scenario is represented as a kitting task. Wherein, an agent is tasked to perform a pick and place operation on an object. The task is provided as an example in the SkiROS GitHub repository[1].

To facilitate this comparison, both methods are subjected to an identical task. The scenario implements mock skills, each designed to simulate a one-second delay before setting the

[1] https://github.com/RVMI/skiros2_examples

relevant parameters to mimic successful execution. Furthermore, a set of checks is employed in FROGBT, while SkiROS employs a different approach integrating conditions within the skill unit. Furthermore, an ontology is constructed, adhering to the structure explained in Section 5.2. An overview of the skills and checks for both methods are described in the appendix in Table 1.

The scenario is repeated multiple times to ascertain an average planning time for each approach, enabling a comparison of their performance.

4.2 FROGBT Flexibility

This scenario aims to assess the generality and modularity within FROGBT's design. This scenario presents the task of preparing a list of products within a simulated supermarket environment. The process of preparing a product in this context entails performing a pick-and-place operation. The chosen environment is simulated, encompassing a panda arm robot endowed with a mobile base.

The objective here is twofold: to evaluate the versatility of FROGBT through the execution of repetitive tasks involving distinct products, and to gauge the adaptability of skills implemented by various engineers, provided kindly by AIRLab[1]. An overview of the skills and checks can be found in the appendix in Table 3. Additionally, an ontology is constructed for this scenario integrating the skills, checks and the connection between them as well as the products and elements of interest in the environment according to the structure explained in Section 5.2.

Executed across various products and diverse circumstances, this scenario not only evaluates FROGBT's generality but also sheds light on its aptitude in employing skills fashioned by different engineers.

4.3 Failure Recovery

This scenario runs in the same simulated supermarket environment and features the same agent. However, a failure is deliberately introduced during the execution of the *deliver* skill.

The selection of the *deliver* skill was deliberate. This choice was informed by the fundamental principle of skill modularity emphasized by FROGBT. Given that the pick-and-place pipeline had been developed by other engineers, the intention was to avoid any interference with the pre-existing modular skills and their implementations.

Consequently, the *deliver* skill was chosen as the focal point for testing failure recovery mechanisms, specifically to assess the functionality of fallback branches and the application of the 'hold-condition' concept in the generation of recovery behaviours.

Notably, additional skills and checks became necessary to facilitate the execution of this scenario, particularly within the recovery behaviour context. An overview of these skills and checks is described in the appendix in Table 4. These supplementary elements were integrated into the ontology following the structured approach explained in Section 5.2.

The goal of this scenario is to highlight, analyse and evaluate the failure recovery and recovery behaviour generation mechanism introduced in FROGBT.

Through these motivating scenarios, a deeper look at the strengths and contributions of the proposed method is provided. The results evaluated from these scenarios lay the foundation for further exploration and research regarding ontological reasoning in generating BTs, establishing FROGBT's potential for advancing the field of BT generation in robotics.

5 METHODOLOGY

This section aims to discuss the methodology used in making FROGBT[2] starting with the setup, then the ontology structure, followed by the main approach, and finally the failure recovery concept.

5.1 Setup and Packages

This subsection outlines the essential components, software packages, and configurations used in implementing the proposed approach:

- Robot Operating System (ROS) played a pivotal role in the development of FROGBT by serving as the execution platform and facilitating communication between FROGBT components. It acted as a wrapper, enabling the execution of BTs and their construction within the FROGBT structure.

[1]AI for Retail (AIR) Lab Delft: https://icai.ai/airlab-delft/
[2]https://github.com/wisjaber/FROGBT

- BehaviorTreeCPP[1] package was opted for due to its emphasis on the modular nature of BTs. This choice provided the advantage of effortless manipulation and editing of the tree structure.
- GROOT[2] was used for visualisation.
- Packages like lxml and owlready2 were used for constructing and manipulating the BT structure and facilitating interaction with the ontology respectively.
- Rviz and Gazebo plugins were employed for the simulation.
- The supermarket environment and a substantial portion of the robot skills were generously provided by AirLab, contributing to the evaluation process

5.2 Ontology Structure

A well-defined hierarchical structure, within an ontology, serves as the foundation for representing the cognitive elements of FROGBT. Graphs showing the ontology's detailed connections can be found in the appendix Figure 6 to 8.

The ontology structure is split into two main chunks: Execution classes facilitating the building blocks of the BT, and connection type classes facilitating the relationships between the environment, states, and the execution class instances.

5.2.1 Execution Classes

This type of class consists of three principal sub-classes: Skills, Checks, and Fallbacks. These classes form a comprehensive basis to encapsulate the essential elements of the BT building blocks.

The Skills class forms the cornerstone of agent proficiency, encapsulating the concept of action nodes within a BT. Each instance of this class embodies a discrete skill and is enriched with essential properties. Notably, instances possess Precondition, Holdcondition, and Postcondition ObjectProperties that establish connections with pertinent checks, ensuring that the requisite conditions for execution are fulfilled. These instances also encompass a Subtree DataProperty, a string encapsulating the XML representation of the corresponding action leaf within the larger BT structure.

Complementary to the Skills class, the Checks class stands as a representation of evaluative conditions within the BT. Mirroring condition leaf nodes, each Checks instance is linked to a specific state, facilitating the assessment of whether the given state has been attained. This linkage is achieved through the isCheckFor ObjectProperty. Additionally, each instance has Subtree DataProperty encapsulating its XML representation in the BT.

Concurrently, the Fallbacks class plays a key role in the ontology, catering to fallback behaviours. Instances within this class are categorised as either representing distinct fallback branches or shared instances interwoven with the Skills class. Fallback branch instances are enriched with a Subtree DataProperty that defines their role as encapsulated black-box behaviours. These instances are intertwined with their respective skills through the hasFallback ObjectProperty. However, shared instances, seamlessly bridging Skills and Fallbacks, are dedicated to recovery behaviour generation and remain unrelated to the primary BT planning.

5.2.2 Connection Classes

Beyond the three execution-related classes, the ontology encompasses additional classes that are instrumental in context representation. These include environment-type classes, such as the Location and Product classes, encapsulating parts of the environment. As well as, affordance-type classes, such as ProductAffordance class, represent states that can be reached with a product.

Location instances offer key details about waypoints, including Position and Orientation DataProperties that contribute to their spatial definition. In parallel, instances of the Product class are connected to tags for product identification through the tag DataProperty. Furthermore, the isLocated ObjectProperty bridges the Product with Location instances, defining the waypoint in the environment from which the product can be interacted with.

ProductAffordance class emerges as an instrumental way in representing the varied states products can adopt. In this strategic extension, this class's instances denote states achievable by executing specific skills, thereby establishing a connection with the respective skill through the EffectOf ObjectProperty.

Notably, each instance within ProductAffordance is also linked to an associated check that

[1] https://github.com/BehaviorTree/BehaviorTree.CPP
[2] https://github.com/BehaviorTree/Groot

evaluates the state, fostering a comprehensive assessment mechanism. This bidirectional linkage is manifested through the ObjectProperties hasChecks and isCheckFor, effectively outlining the state-check relationship.

In this structured ontology, the intricate interplay of classes and properties constructs an approach that supports the inner workings of FROGBT, showcasing its formidable tools in intelligent agent control and failure management.

5.3 FROGBT Architecture

The FROGBT approach is structured around three key components, each serving a distinct purpose while collaboratively exchanging information to ensure seamless execution. A graph showing the main components and their connection can be seen in Figure 2 and the pseudo-code of it is shown in the appendix in algorithm 1.

5.3.1 Goal Parser:

At the core of the FROGBT approach, the Goal Parser plays a key role in extracting relevant information and parameters from the ontology, effectively guiding the generation of the BT. This component receives user-provided inputs in the form of four variables: Action, Goal, Goal_details, and Destination. These variables encapsulate essential task details:

- Action: Specifies the required action, whether it's "move," "pick," "deliver," or others.
- Goal: Identifies the target, which could be a product or a waypoint.
- Goal_details: Provides specific product identification through tag IDs.
- Destination: Points to the destination, be it another waypoint or a delivery location.

Using these inputs, the Goal Parser queries the ontology for details about waypoints, products, actions, and checks. It then updates the BT's goal checks and actions correspondingly, ensuring tasks are performed on the intended products or delivered to the designated destinations. Once identified, the goal condition and relevant information are dispatched to the BT Manipulator component.

5.3.2 BT Manipulator:

The BT Manipulator component is responsible for translating the extracted information into a functional BT structure. It starts by constructing an XML file with the initial goal check as the BT's first iteration. In cases of failure, it interfaces with the BT Monitoring component to retrieve the failed check's name and index. The BT Manipulator then consults the ontology to identify suitable actions for resolving the failed check. This process encompasses generating the atomicBT, updating the goals in the XML structure with the respective information of interest, and adding it to the XML file of the generated BT.

Additionally, when a recovery behaviour generation is activated, the BT Manipulator is notified through the BT Monitoring component. It then extends the checks with associated recovery skills which are defined in the ontology as shared instances between Fallbacks and Skills classes. The pseudo-code of the component can be found in the appendix in algorithm 2.

5.3.3 BT Monitoring:

The BT Monitoring component assumes the role of overseeing the BT's execution. Upon receiving the XML file from the BT Manipulator, it initiates execution via a ROS action server. It continuously monitors the BT's root execution status. In the event of a FAILURE signal from the root, BT Monitoring promptly identify the name and index of the last failed check. This information is then relayed back to the BT Manipulator for BT extension.

Moreover, when a reasoning node triggers the start of a recovery behaviour generation branch, BT Monitoring receives a message with failure information. This information, including the failed hold-condition and initiation of a recovery branch, is passed to the BT Manipulator for tree extension with the appropriate recovery skills. Subsequently, upon the recovery BT's root returning SUCCESS, the main BT's root is re-ticked, and the execution continues.

Ultimately, upon successful execution of the main BT's root, BT Monitoring signals the completion of the task. This intricate collaboration among the components ensures the execution robustness and recovery agility of the FROGBT approach. The pseudo-code of the component can be found in the appendix in algorithm 3.

5.4 Failure Recovery

The concept of failure recovery is the motivation behind the development of FROGBT. This arises

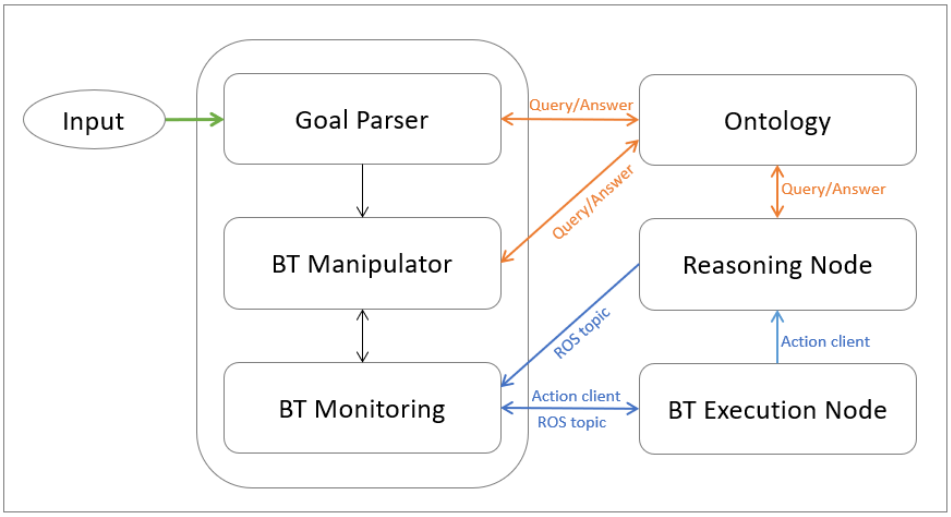


Figure 2: The connections between the main components of FROGBT's architecture

from a recognised gap in existing literature, where generated BT methods often lack comprehensive recovery mechanisms. Conventional methods typically offer re-planning or, at best, hand coded fallback branches added to the plan as options for recovery.

5.4.1 Hold-Condition Concept:

In FROGBT, each skill is associated with an action, accompanied by pre-conditions to establish the correct execution context, and a post-condition representing the action's intended outcome. Notably, some actions require certain conditions to persist throughout their execution, termed Hold-conditions. This concept ensures that specific conditions remain true while the action is underway. The distinct separation of checks and skill units within FROGBT allows for the modular utilisation of checks. For instance, when delivering an object, the *deliver* skill is the same as the *drive* skill from an implementation perspective. However, using the *drive* skill could be executed after the *pick* skill to achieve the same goal. Although the latter may indicate success after reaching the destination, the object might have been lost in the process. Whereas, integrating a Hold-condition for the *deliver* skill enforces the verification of the object being held throughout execution, thus enhancing its conceptual success.

This Hold-condition doubles as a failure detection mechanism and serves as a targeted recovery goal. Through a well-defined ontological relationship, this concept enables the generation of specialised recovery behaviours for specific failures using the skills associated with fallback.

5.4.2 Fallback Branch:

Certain skills might necessitate predefined sequences of actions to recover from failures. Fallback branches embody these scripted recovery behaviours. FROGBT accommodates such needs through the integration of Fallback branches. This integration is realised through ontological connections between skills and fallback instances, explicitly defining these branches. The specific origin of the branch is irrelevant, as long as it conforms to a structured XML BT format.

5.4.3 Reasoning Node:

In the FROGBT Manipulator component during the creation of a skill's atomicBT, the presence of a hold-condition or a defined fallback branch within the ontology triggers the inclusion of a "Reasoning node" alongside the hold-condition check. The template of the atomicBT with reasoning node can be seen in Figure 3.

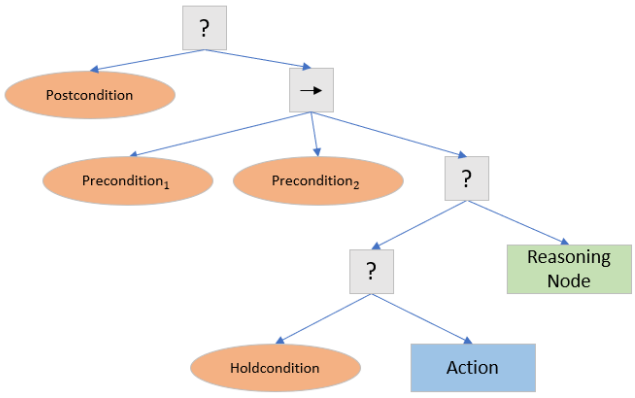


Figure 3: Template for atomicBT of skill with a fallback branch or hold-condition

The Reasoning node operates as a ROS action client that communicates the failed action's name to a corresponding ROS action server. The server interacts with the ontology to identify a defined fallback branch for the failed skill. If such a branch is found, the recovery branch is formulated using the specified Subtree structure and initiated as a separate XML file as a recovery BT.

Conversely, if a fallback branch is absent, the failed hold-condition is communicated to the BT Monitoring component, marking the start of a recovery behaviour generation branch. This branch takes the failed condition as its goal. The FROGBT monitoring component is alerted, prompting the expansion of the recovery BT with the relevant skills, thus constructing the recovery behaviour iteratively and organically following the same rules for the main BT generation.

6 EVALUATION

In this section, we will evaluate the result of running the motivating scenarios, highlighting the key features of FROGBT in each of these scenarios.

6.1 SKIROS VS FROGBT

This evaluation aims to establish a benchmark comparison between the FROGBT and SkiROS methods. This comparison centers around the difference in design aspects of the BTs generated by both approaches, as well as the time required to formulate a plan. The selected scenario is a basic kitting task.

SkiROS generates a sequential BT based on the plan derived through the PDDL planner, as depicted in Figure 4.

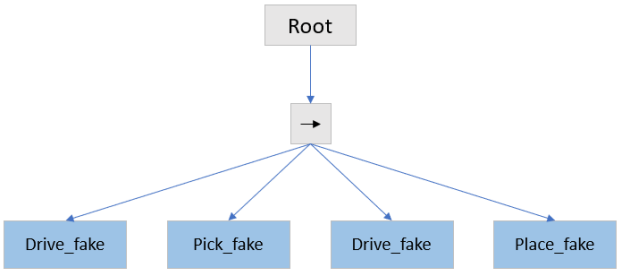


Figure 4: BT generated for the execution of the plan with SkiROS, SkiROS GUI is shown in the appendix in Figure 1.

Conversely, FROGBT adopts a deliberative back-chaining design principle, iteratively constructing the BT, as illustrated in Figure 5.

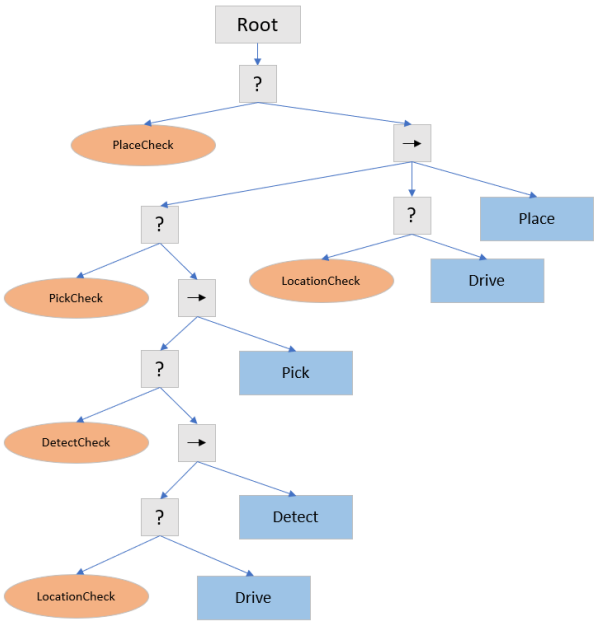


Figure 5: BT generated using FROGBT, the BT from this scenario is shown in the appendix in Figure 2.

Method	Avg. planning time
SkiROS	0.351 seconds
FROGBT	0.0936 seconds

Table 2: comparison in planning time between SkiROS and FROGBT methods

The evaluation involved conducting the scenario ten times on the same hardware setup. The average time for both methods was measured and recorded as outlined in Table 2. SkiROS exhibited average times of 0.28 and 0.042 seconds for generating problem and domain files, respectively, leaving 0.029 seconds for actual plan derivation, it generates those files for the PDDL planner each time a plan is sought. In contrast, FROGBT required 0.09 seconds to query the ontology and construct the BT, benefiting from the omission of domain and problem file generation overhead. This efficiency contributes to FROGBT's streamlined approach in generating plans.

6.2 FROGBT Flexibility

This evaluation examines the generality and reusability of the proposed approach within a simulated supermarket environment. The objective is to demonstrate the adaptability of the FROGBT method when employing skills crafted by various engineers. The scenario involved the task of preparing a list of three distinct products. Two

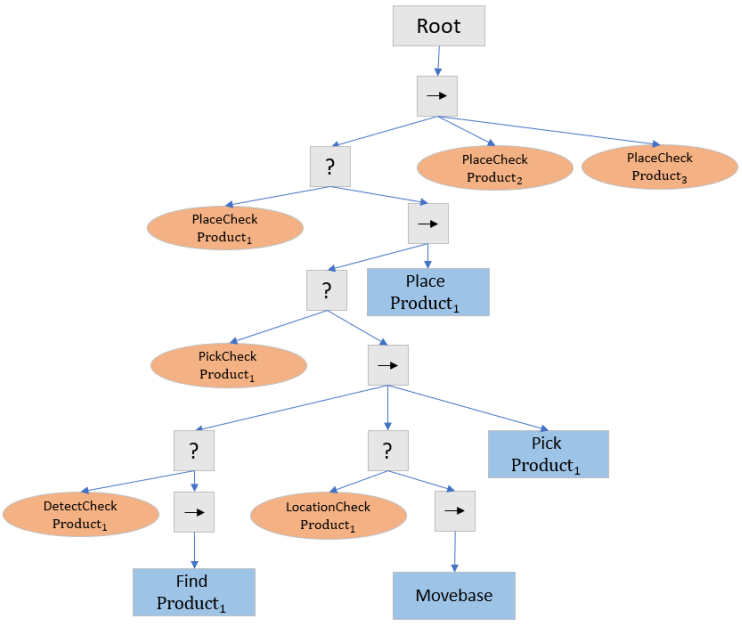


Figure 6: Example of BT generated to prepare a list of products, focusing on one product BT expansion, full BT generated and executed can be seen in the Appendix in Figure 3

distinctive methodologies were employed to address this scenario: using a sequence of goal conditions, and the reuse of a previously generated BT.

6.2.1 Goals Conditions Sequence

The list of products is given to the goal parser, which queries the relevant parameters for each item from the ontology. Subsequently, a sequence of goal conditions for each product is generated[1]. The BT structure is then expanded to accommodate these goal conditions iteratively, generating a BT to execute the task at hand Figure 6.

6.2.2 BT Reuse

Alternatively, FROGBT leveraged a previously generated BT for one product, capitalising on the structure saved in an XML file and the modularity of BTs. This method proved particularly advantageous when dealing with repetitive tasks in the same environment. The previously generated XML was dynamically updated by identifying the parameters specific to the new product, subsequently executing the adapted BT[2]. Notably, the approach actively monitored the BT's execution, dynamically extending it to address any unforeseen situations or uncertainties that may have arisen since the BT's original generation. Both BTs and the changes to adapt can be seen in the appendix Figure 4.

The results of this evaluation showcased the inherent generality and flexibility of the approach by successfully employing skills crafted by AIRLab engineers within a simulated environment. Moreover, the practice of reusing previously generated BTs highlights the modularity of BTs. Furthermore, FROGBT's active planning component notably demonstrated its capacity to adapt and expand the BT as required. This adaptability further heightened the reusability and overall effectiveness of FROGBT.

6.3 Failure Recovery

This evaluation examines the failure recovery mechanisms within the FROGBT approach. The scenario involves inducing a failure during the execution of the *deliver* skill thereby invalidating the skill's associated hold-condition. FROGBT offers two failure recovery strategies: using fallback branches, and recovery behaviour generation.

6.3.1 Fallback Branch

One of the ways to recover from failure within FROGBT is achieved through behavioural retrieval[3]. In this strategy, a predefined fallback branch, as shown in Figure 7, is found by the *"reasoning node"* and then invoked as a recovery

[1]Video demonstration for list of products: `https://youtu.be/J6N2Luowpic`
[2]Video demonstration of BT reuse: `https://youtu.be/0bR_RgNxoTk`
[3]Video demonstration of fallback branch recovery: `https://youtu.be/jNJ79VK3nFo`

branch. After the successful completion of this branch, the root node of the main BT is ticked again, allowing the plan to continue unhindered.

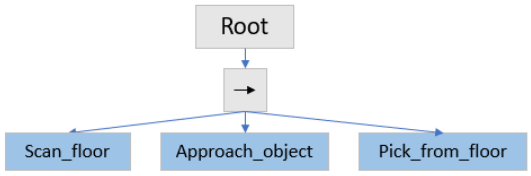


Figure 7: Fallback branch to recover from dropping the product, the branch used can be seen in appendix Figure 5a.

6.3.2 Behaviour Recovery Generation

An alternative strategy to deal with failure in a creative way within FROGBT is generating the recovery behaviour itself[1]. Upon triggering the failure, the *recovery node* is ticked, initiating the process of generating a recovery behaviour. The underlying cause of the failure, the hold-condition, is designated as the goal condition for the recovery sub-tree generation process. Subsequently, a recovery behaviour is generated to resolve the failure, as shown in Figure 8.

Upon the successful execution of the recovery behaviour, the root node of the original BT is ticked, enabling the seamless resumption of the plan from the point of interruption, thanks to FROGBT's back-chaining BT design.

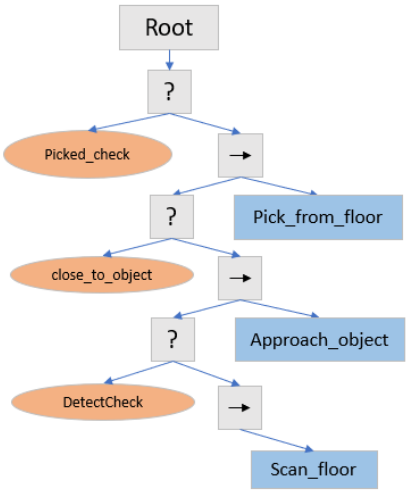


Figure 8: Final iteration for the generated recovery behaviour for the goal condition Picked_check, the generated BT for the recovery behaviour can be seen in the appendix Figure 5b.

These observations show the robustness of FROGBT and its ability to generate recovery behaviours without necessitating computationally intensive re-planning efforts. The chosen BT design architecture ensures that the plan can proceed seamlessly following recovery from a failure event. The recovery behaviour's generation based on the failed hold-condition enhances the adaptability of the method, enabling recovery from various failures that might be associated with the same skill.

Moreover, the incorporation of scripted recovery behaviours through the fallback branch introduces an adaptive facet to FROGBT. The concept of recovery behavioural retrieval further expands the approach's utility, permitting the integration of diverse blackbox behaviours, regardless of their origin or construction method. These include techniques like learning from demonstration [18], evolutionary algorithms [19], or human-engineered designs.

Nonetheless, it's worth noting that the size of the ontology can exert a noticeable influence on the efficiency of the plan generation process. In a comparative analysis, the initial scenario involved a relatively compact ontology designed to align with an example provided by SkiROS developers. In this scenario, the average planning time converged to approximately 0.09 seconds. However, as more extensive product sets were incorporated into the ontology in subsequent scenarios, the average planning time increased to approximately 0.23 seconds where a significant portion of this planning time, averaging 86.95%, was devoted to the goal parser's initial phase, highlighting the need for a more optimised algorithm for initial parameters retrieval from the ontology.

7 CONCLUSION

FROGBT represents a bridge for a literature gap in the field of generating BTs by incorporating ontological reasoning as a foundational structure. This novel integration of ontological reasoning mechanisms facilitates a profound connection between an agent's represented knowledge, encompassing its skills and capabilities. Moreover, it empowers the agent with contextual insights into the specific portions of the environment where it operates. This contextual knowledge is firmly grounded using ROS services, which ensure accurate accounting of observable environmental factors. This higher-level linkage provides FROGBT with the capacity to enrich the agent's

[1]Video demonstration of recovery generation: `https://youtu.be/wVvYLAHj1dc`

skills with context, whether for planning or failure recovery scenarios.

The approach has demonstrated its efficacy through comparisons with state-of-the-art framework SkiROS. It has showcased its generality and modularity by unifying skills developed by various individuals to accomplish recurring tasks. Furthermore, FROGBT introduces innovative failure recovery techniques, including the incorporation of fallback branches and the dynamic generation of recovery behaviours.

In conclusion, the approach casts a spotlight on the potential of ontological reasoning to augment BT generation with contextual understanding and reasoning capabilities. This work lays a foundation for future research endeavors, specifically in the domain of Failure Recovery with Ontologically Generated behaviour Trees.

8 FUTUREWORK

While FROGBT offers a promising avenue for generating BTs to execute tasks, it is not immune to certain challenges. It is noteworthy that the bottleneck was identified at the initial phase in the goal parser. This phase involves the identification of key parameters associated with a given product or a list of products.

As part of future work, it becomes imperative to explore avenues to enhance the efficiency of this parameter identification process. Some interesting solutions could be:

- Creating a cache of frequently queried parameters and their associated ontology elements. This can significantly reduce query times for commonly accessed information.
- Optimising ontology query construction and execution. By minimising unnecessary joins or complex operations, and utilising query optimisation techniques.
- Pruning unnecessary data from the ontology before querying. This can involve identifying irrelevant branches of the ontology tree or excluding data that is not needed for the specific query.

By addressing the intricacies of parameter identification, FROGBT can potentially overcome the performance bottleneck observed in the goal parser phase. This would pave the way for even more efficient task execution and bolster the overall efficacy of the FROGBT approach.

Appendix

Skill	Parameters	Pre-conditions	hold-conditions	Post-conditions
Locate	ContainerLocation	Robot at container		Object found in container
Drive	StartLocation TargetLocation	Robot at StartLoaction		Robot at TargetLoaction
Pick	ContainerLocation Object Location Gripper state	Gripper empty Object found in container	Robot at ContainerLocation	Gripper is full Object in gripper
Place	PlacingLocation Object Location Gripper state	Gripper holding object	Robot at PlacingLocation	Gripper is empty Object in PlacingLocation

Table 1: Skills of SkiROS example and their parameters, pre-conditions, hold-conditions and post-conditions.

Skill	Pre-conditions	hold-conditions	Post-conditions
Detect	Location check		Object state is detected
Drive			Robot is at Location
Pick	Location check Detect check		Object state is picked
Place	Picked check Location check		Object is placed Object in PlacingLocation

(a) Skills of FROGBT method and their pre-conditions, hold-conditions and post-conditions.

Check	Goal	Returns SUCCESS
Location check	Waypoint	When Robot located property is Waypoint
Detect check	Object state	When Object state is detected
Picked check	Object state	When Object state is picked
Placed check	Object state	When Object state is placed

(b) Conditions of FROGBT method and their evaluation return.

Table 2: Skills and Checks for the benchmark scenario using FROGBT.

Skill	Pre-conditions	hold-conditions	Post-conditions
MoveBase			Location check
Find	Location check ArmPose check		Detection check
Pick	¬ Vacuum check Detection check Location check		Picked check
Deliver	Picked check	Picked check	Location check
Place	Picked check Location check		¬ Vacuum check Placed check

(a) Skills of FROGBT method and their pre-conditions, hold-conditions and post-conditions.

Check	Goal	Returns SUCCESS
Location check	Waypoint	When Robot is at Waypoint
Detection check	Product's Tag	When Tag is detected
Vacuum check		When vacuum is activated (Gripper is full)
ArmPose check	Pose	When Arm is in the goal pose
Picked check	Product's Tag	When Vacuum check and product state is picked
Placed check	Product's Tag	When ¬ Vacuum check and product state is placed
Close check	Product's Tag	when Approach_object skill is completed successfully

(b) Conditions of FROGBT method and their evaluation return.

Table 3: Skills and Checks for the preparing of products scenario using FROGBT approach.

Skill	Pre-conditions	hold-conditions	Post-conditions
Scan_floor			Detection check
Approach_object	Detection check		Product is close
Pick_From_Floor	Close check		Picked check

Table 4: Additional skills in the fallback branch for Deliver skill.

Algorithm 1 FROGBT Approach

1: **Input:** action, goal, goal_details, destination
2: goal_condition, parameters_of_interest ← goal_parser(action, goal, goal_details, destination)
3: BT_iteration ← BT_manipulator(goal_condition, parameters_of_interest)
4: BT_monitor(BT_iteration)

Algorithm 2 BT_manipulator Function

```
function BT_MANIPULATOR(check_name, parameters_of_interest, recovery)
    actions,preconditions ← query_ontology(check_name)
    atomicBTs ← []
    if len(actions) > 1 then
        for action in actions do
            if action.HoldCondition or action.hasFallback then
                atomicBT ← build_atomicBT_with_reasoning(action,preconditions,action.HoldCondition)
            else
                atomicBT ← build_atomicBT(action,preconditions)
            end if
            atomicBTs.append(atomicBT)
        end for
    else
        atomicBTs ← build_atomicBT(actions,preconditions)
    end if
    insert_under_fallback(atomicBTs, check_name)
    update_parameters(BT_iteration, parameters_of_interest)
    Return BT_iteration
end function
```

Algorithm 3 BT_monitor Function

```
function BT_MONITOR(BT_iteration)
    execute(BT_iteration)
    while BT_Root() returns Failure do
        failed_check ← get_failed_check()
        failure_recovery ← check_failure_recovery()
        if failure_recovery then
            recovery_BT ← BT_manipulator(failed_check, parameters_of_interest, recovery=true)
            execute(recovery_BT)
            while recovery_root() returns Failure do
                failed_check ← get_failed_check()
                recovery_BT ← BT_manipulator(failed_check, parameters_of_interest, recovery=true)
                execute(recovery_BT)
            end while
            execute(BT_iteration)
        else
            BT_iteration ← BT_manipulator(failed_check, parameters_of_interest)
            execute(BT_iteration)
        end if
    end while
end function
```

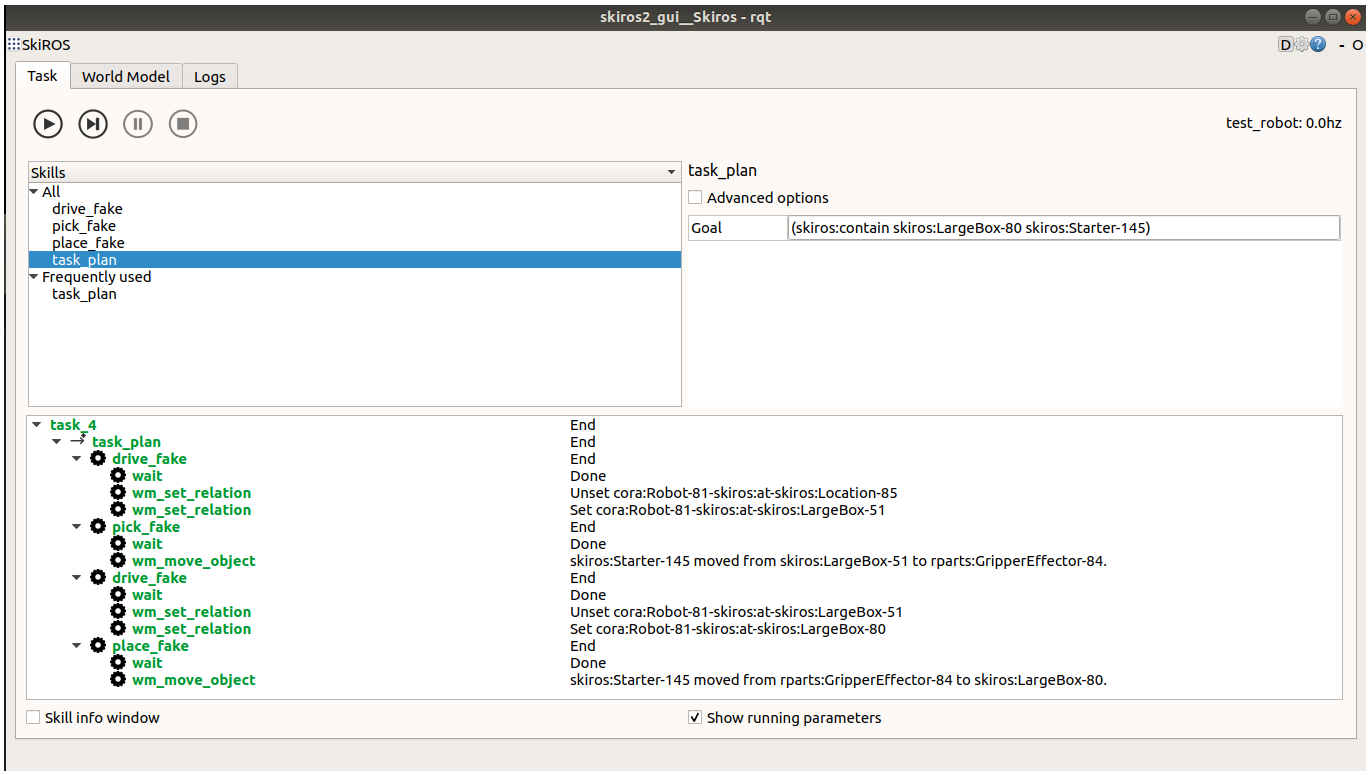


Figure 1: SkiROS GUI executing the plan for the kitting task

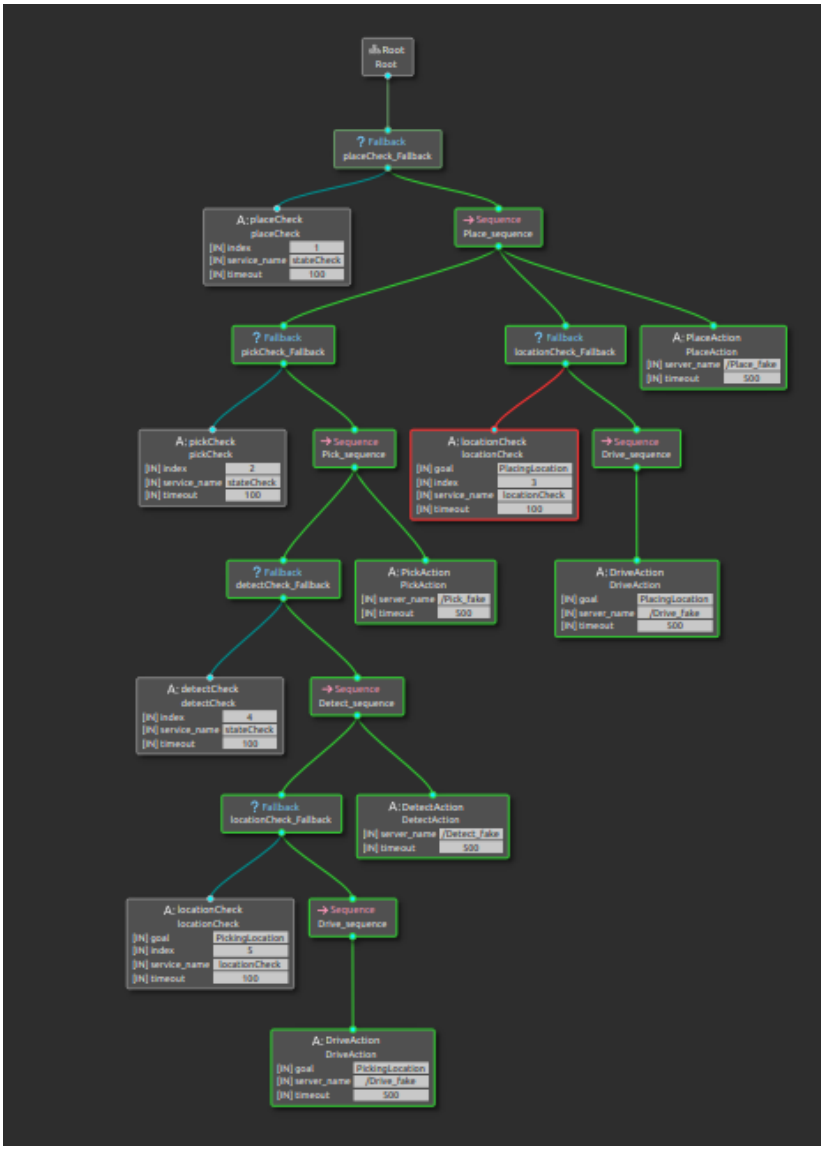


Figure 2: FROGBT's generation and execution the plan for the kitting task

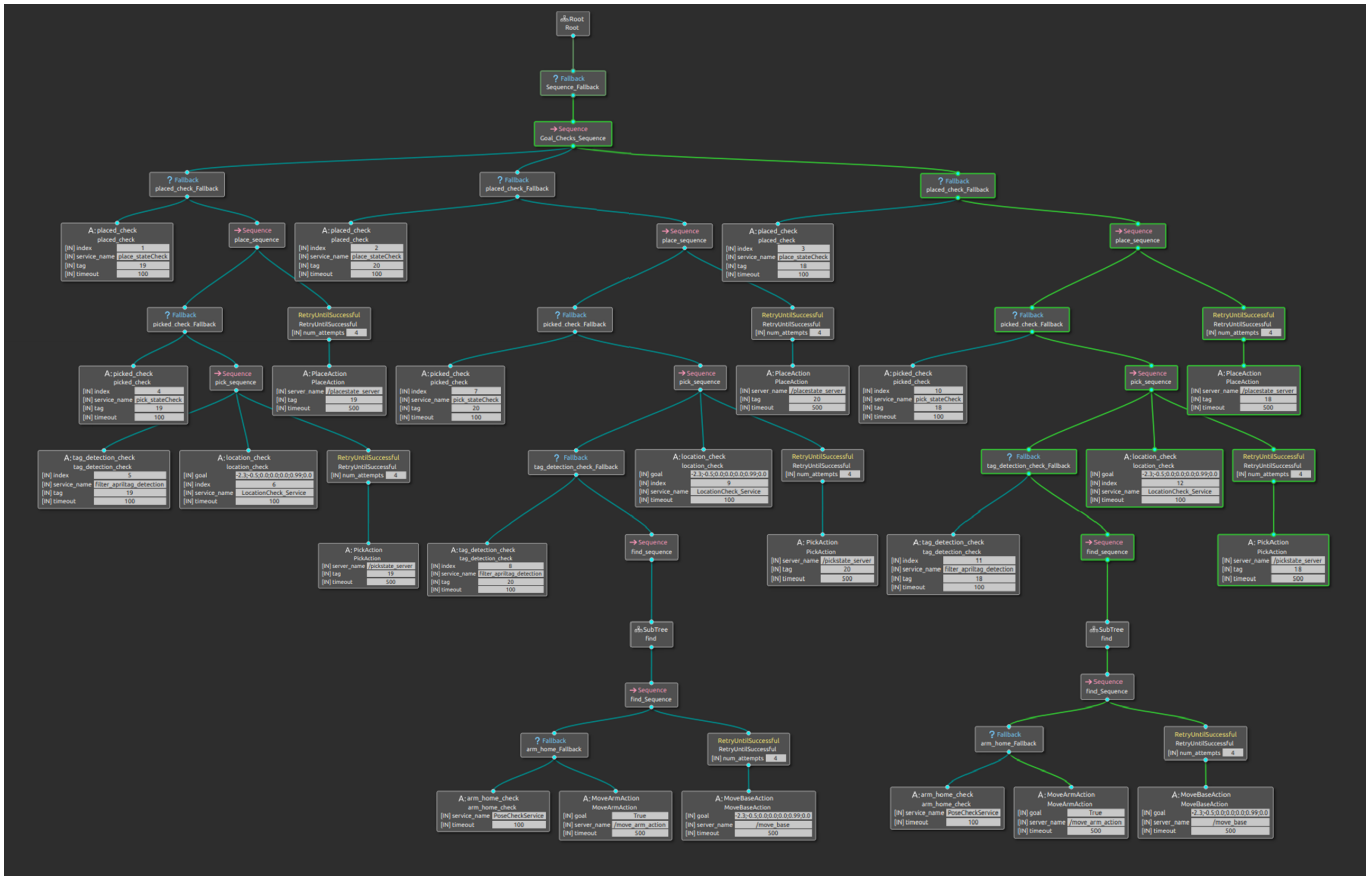
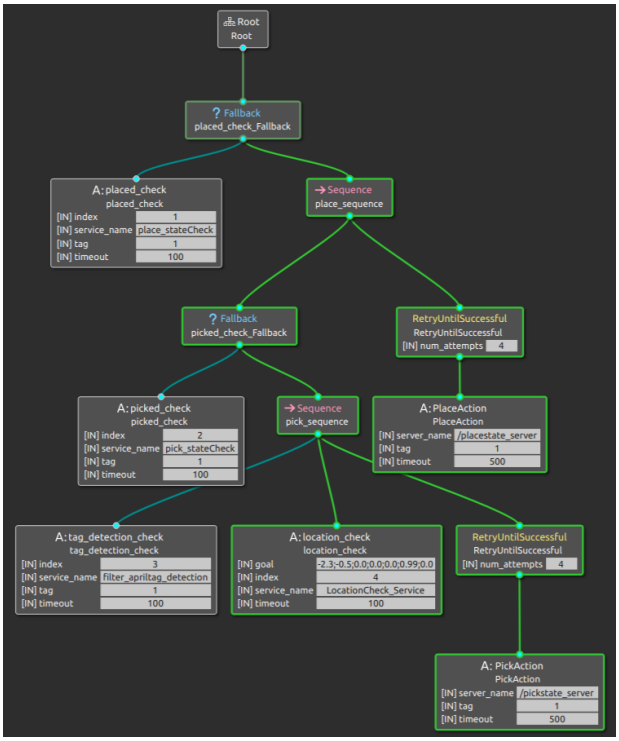
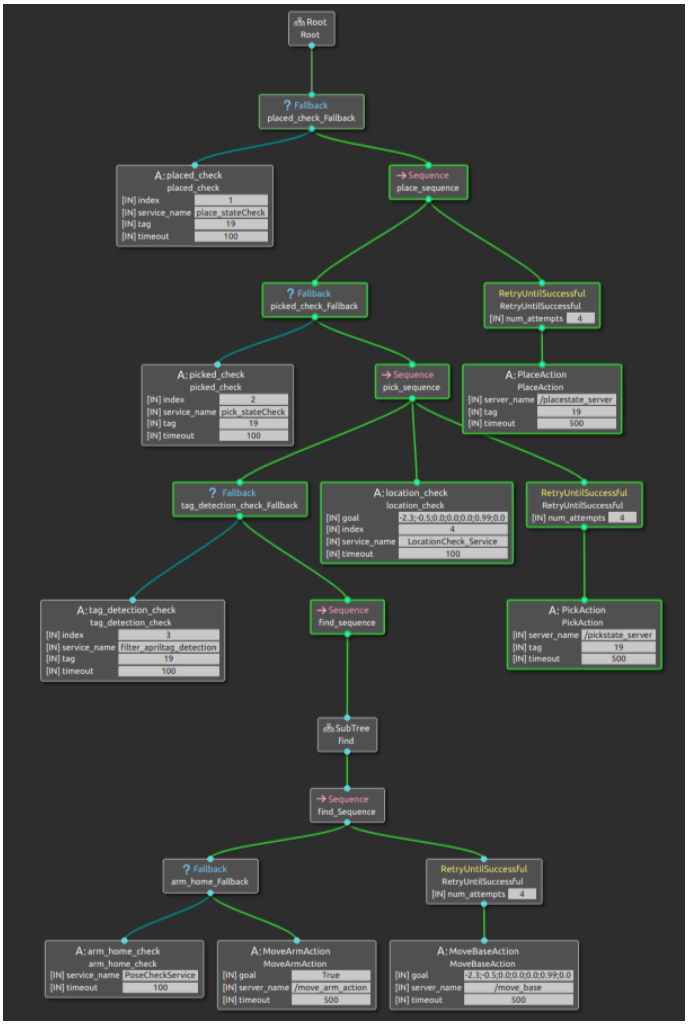


Figure 3: One of the full BTs generated using FROGBT for the scenario of preparing a list of products

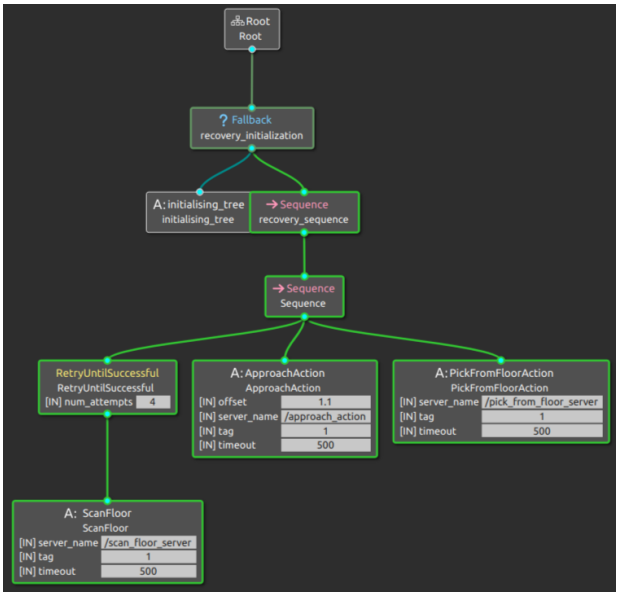


(a) BT generated for picking one product

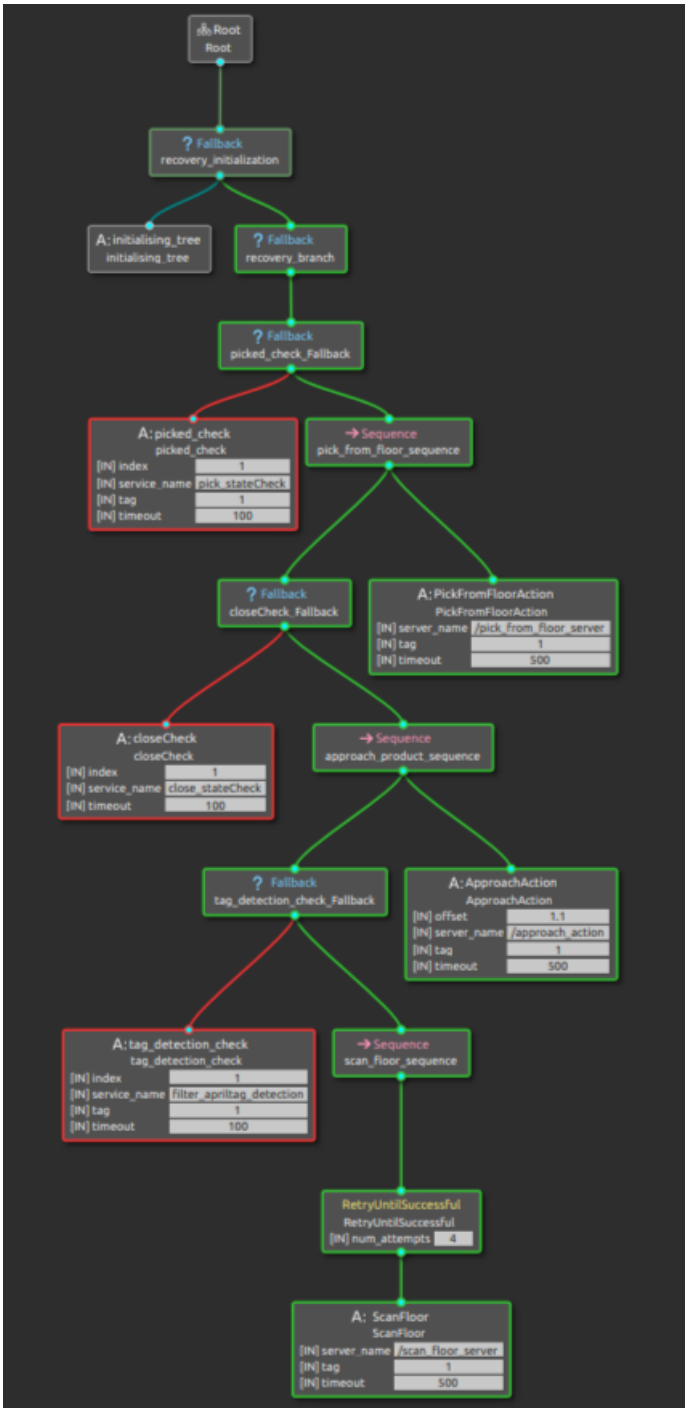


(b) BT updated after reuse for picking the second product

Figure 4: BT reuse, in fig 4a the BT was generated for one product and reused for the second product, however, more actions were needed and FROGBT updated the BT accordingly for the second product fig 4b



(a) BT for fallback branch recovery



(b) BT for the generated recovery behaviour

Figure 5: Two ways of failure recovery using FROGBT, using a hand crafted sub-tree as a fallback branch, and generating the recovery behaviour from the failed hold-condition

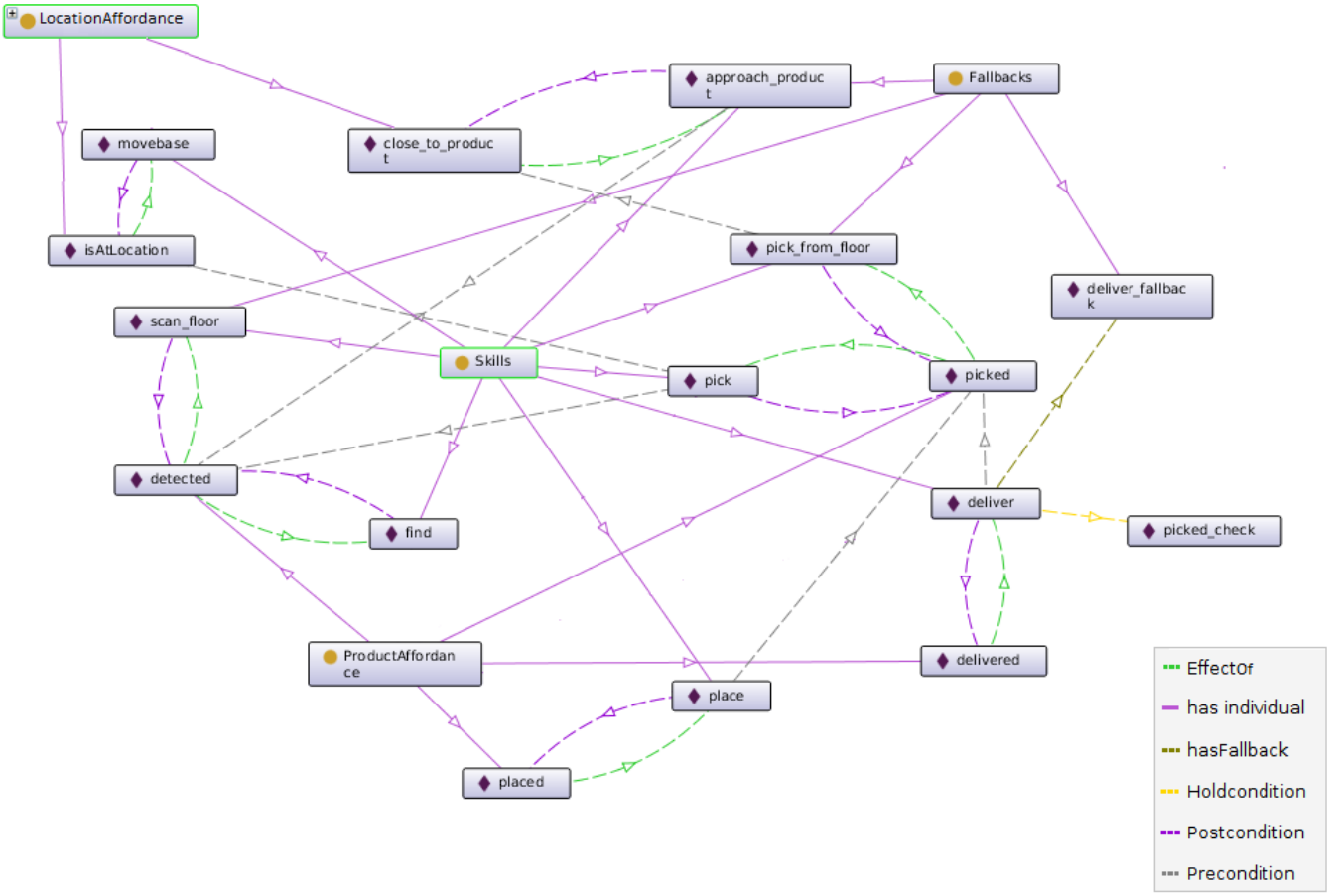


Figure 6: The skills individuals and their connections, as well as the shared instances between skills and fallbacks class

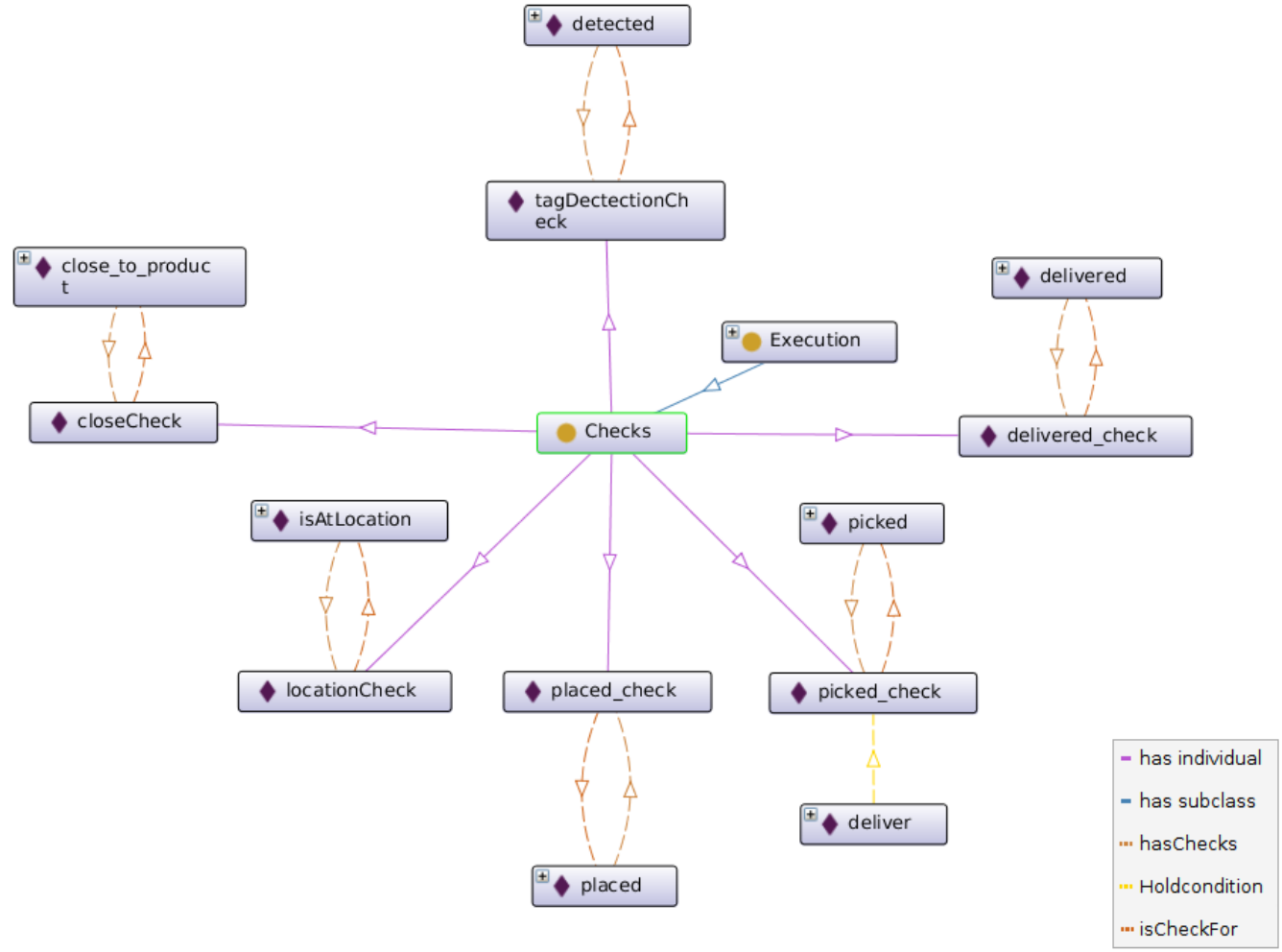


Figure 7: The checks individuals and the relationship between them and the states in Affordance type classes

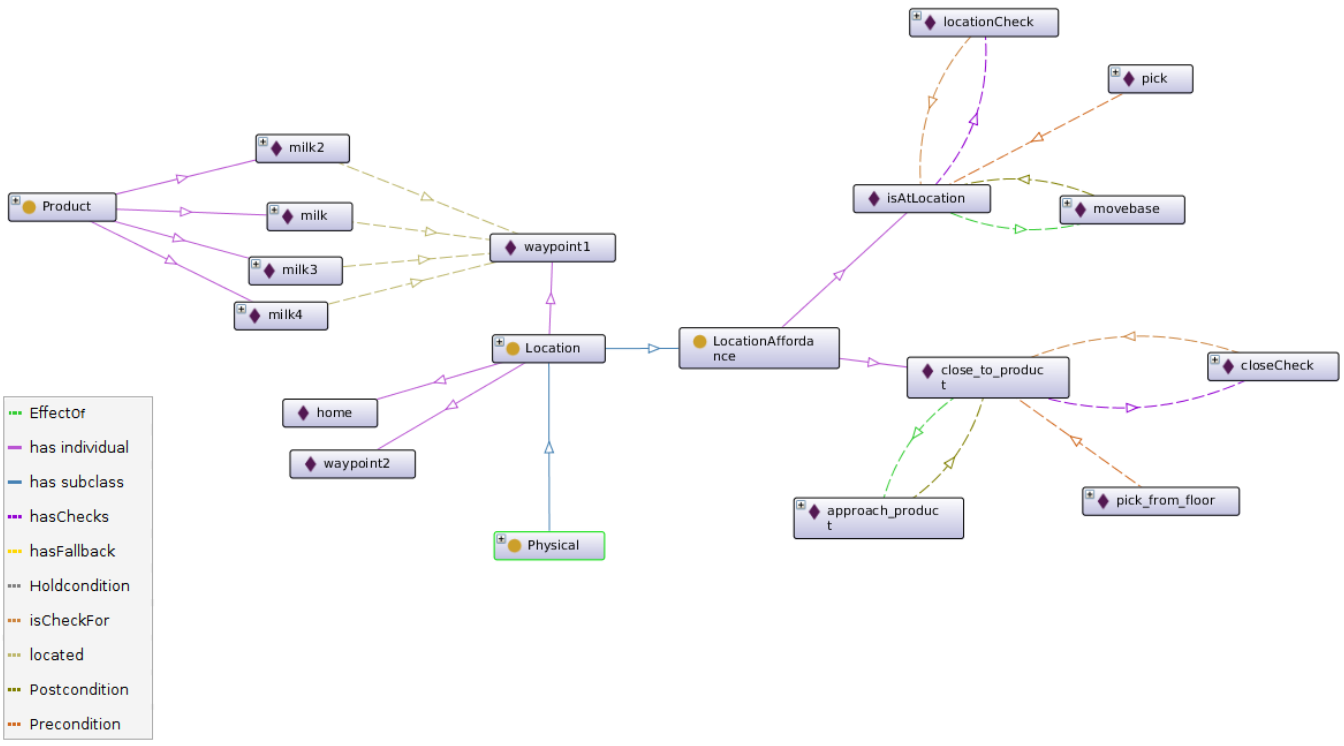


Figure 8: The connections for products with the Affordance type classes

References

1. D. Isla. *Handling complexity in the halo 2 AI* https://www.gamedeveloper.com/programming/gdc-2005-proceeding-handling-complexity-in-the-i-halo-2-i-ai.
2. Colledanchise, M. & Ögren, P. *Behavior trees in robotics and AI: An introduction* (CRC Press, 2018).
3. Younes, H. L. & Littman, M. L. PPDDL1. 0: An extension to PDDL for expressing planning domains with probabilistic effects. *Techn. Rep. CMU-CS-04-162* **2,** 99 (2004).
4. Ghallab, M., Nau, D. & Traverso, P. *Automated Planning: theory and practice* (Elsevier, 2004).
5. Rovida, F., Grossmann, B. & Krüger, V. *Extended behavior trees for quick definition of flexible robotic tasks* in *2017 IEEE/RSJ International Conference on Intelligent Robots and Systems (IROS)* (2017), 6793–6800.
6. Martín, F., Morelli, M., Espinoza, H., Lera, F. J. R. & Matellán, V. *Optimized Execution of PDDL Plans using Behavior Trees* 2021. https://arxiv.org/abs/2101.01964.
7. Colledanchise, M., Almeida, D. & Ögren, P. *Towards blended reactive planning and acting using behavior trees* in *2019 International Conference on Robotics and Automation (ICRA)* (2019), 8839–8845.
8. Tadewos, T. G., Shamgah, L. & Karimoddini, A. *Automatic Safe Behaviour Tree Synthesis for Autonomous Agents* in *2019 IEEE 58th Conference on Decision and Control (CDC)* (2019), 2776–2781.
9. Platzer, A. *Logical analysis of hybrid systems: proving theorems for complex dynamics* (Springer Science & Business Media, 2010).
10. Wilkins, D. E. Recovering from execution errors in SIPE. *Computational Intelligence* **1,** 33–45 (1985).
11. Chang, K.-H., Han, H. & Day, W. B. A comparison of failure-handling approaches for planning systems—replanning vs. recovery. *Applied Intelligence* **3,** 275–300 (1993).
12. Macenski, S., Martín, F., White, R. & Clavero, J. G. *The Marathon 2: A Navigation System* in *2020 IEEE/RSJ International Conference on Intelligent Robots and Systems (IROS)* (2020), 2718–2725.
13. Hu, D., Gong, Y., Hannaford, B. & Seibel, E. J. *Semi-autonomous simulated brain tumor ablation with ravenii surgical robot using behavior tree* in *2015 IEEE International Conference on Robotics and Automation (ICRA)* (2015), 3868–3875.
14. Segura-Muros, J. Á. & Fernández-Olivares, J. *Integration of an automated hierarchical task planner in ros using behaviour trees* in *2017 6th International Conference on Space Mission Challenges for Information Technology (SMC-IT)* (2017), 20–25.
15. Manzoor, S. *et al.* Ontology-Based Knowledge Representation in Robotic Systems: A Survey Oriented toward Applications. *Applied Sciences* **11.** ISSN: 2076-3417. https://www.mdpi.com/2076-3417/11/10/4324 (2021).
16. Beetz, M., Mösenlechner, L. & Tenorth, M. *CRAM—A Cognitive Robot Abstract Machine for everyday manipulation in human environments* in *2010 IEEE/RSJ International Conference on Intelligent Robots and Systems* (2010), 1012–1017.
17. Mayr, M., Rovida, F. & Krueger, V. *SkiROS2: A skill-based Robot Control Platform for ROS* 2023. arXiv: 2306.17030 [cs.RO].
18. French, K., Wu, S., Pan, T., Zhou, Z. & Jenkins, O. C. *Learning behavior trees from demonstration* in *2019 International Conference on Robotics and Automation (ICRA)* (2019), 7791–7797.

19. Colledanchise, M., Parasuraman, R. & Ögren, P. Learning of behavior trees for autonomous agents. *IEEE Transactions on Games* **11,** 183–189 (2018).