Bachelor's Thesis

Goal Reasoning with the CLIPS Executive in ROS2

Ivaylo Dinkov Doychev

October 19, 2021

Rheinisch-Westfälische Technische Hochschule Aachen
Knowledge-Based Systems Group
Aachen, Germany

Advisors:

Tarik Viehmann M.Sc., Till Hofmann, M.Sc.

Supervisors:

Prof. Gerhard Lakemeyer, Ph.D., Prof. Sebastian Trimpe, Ph.D.

Contents

1 Introduction

Capabilities of robots rapidly increased over the last few years. They range from better actuators and environment perception to broader real-world applications. While this is a welcome advance for the world of robotics, there are several demands for high- and low-level control. To accommodate for the increased capabilities and higher demands, rich software solutions are required to build complex systems to operate robots.

Planning, reasoning, and execution in complex environments with possibly multiple collaborating robots require efficient solutions for those computationally expensive tasks. One reasonable approach in that direction is the CLIPS Executive (CX)[NHL19]. It acts as a high-level controller and manages all high-level decisions such as goal formation, planner invocation, goal execution, monitoring, and agent coordination. The executive is designed not only to be capable of deliberating in very dynamic environments but also supports the coordination of multiple agents. Its reasoning and the program flow are based on the principles of goal reasoning.

Goal Reasoning (GR) is "the study of agents that can deliberate on and self-select their objectives"[Aha18]. It is a method designed for intelligent agents to deal with high-level goals in a reactive and flexible fashion. Rather than completing a fixed sequence of actions in pursuit of achieving a set goal, the agents are capable of reasoning about the current goal and reacting to exogenous factors, which influence the goal's satisfiability, thus being able to re-evaluate the goal or even switch to a new one. This behavior is especially valuable in dynamic environments. This method has set the fundamental basis for the implementation of the CX, which utilizes goal reasoning in the form of goal lifecycles.

The CLIPS Executive has proven its capabilities in the RoboCup Logistics League (RCLL) [NLF15] - a robotics competition supporting the advancement in autonomous and smart logistics robots, where team Carologistics won the RCLL in 2019 [HLM+19] with it.

The main limitation of the current CLIPS Executive, however, is that it is tightly coupled to the Fawkes framework[NFBL10], where it is developed in. Many of the components and the plugins, which the executive utilizes, are implemented inside Fawkes. Consequently, the CX is limited to being integrated into software stacks, which depend on and use the Fawkes framework. To tackle this limitation and allow more people to benefit from the approach, we decided to expand the availability of the CLIPS Executive system.

In this thesis, we present the integration of the CX in the Robot Operating System (ROS)[QCG+09] and more precisely ROS2. ROS is the de-facto standard open-source middleware for robotics applications. It is supported by a large, lively community, which has provided a sheer amount of tools and packages that cover most requirements in robotics. Oddly enough, there aren't many high-level con-

trollers currently available in ROS. By implementing the CX in ROS2, we could reveal the potential of a GR-based executive to a wider set of people. To the best of our knowledge, this would be a novelty in the ROS ecosystem.

This integration focuses on developing a standalone CLIPS Executive on the side of ROS2. It provides a highly configurable and reliable CLIPS-based executive for the high-level decision-making process inside a robotics software. The first goal is to enable the CX to be integrated into ROS-based applications. This way, ROS agents can utilize the potential of the executive. As a proof of concept for this support, we run the CLIPS Executive in combination with the established ROS2 stacks - Navigation2 [MMWGC20] and Plansys2 [MGRM21]. The second goal of this integration is to provide the means of communication so that other robotics frameworks such as Fawkes can utilize the executive.

This thesis is structured as follows: Section 2 first gives an overview of the Fawkes and ROS platforms and then insight into all relevant technology and preliminaries behind the CLIPS Executive as well as outlining its concept and functionality. Next, we continue with Section 3, which introduces related work in ROS. It examines several systems providing high-level reasoning capabilities. Section 4 gives a basic description of the systems design and main system requirements. It proceeds with the elaboration of the system implementation. Section 5 describes the evaluation settings and criteria, and the actual evaluation of the CX system inside the ROS2 ecosystem. It also evaluates the possible integration of the CX into Fawkes. Finally, Section 6 provides the summary and conclusion of our approach and outlines future development possibilities.

2 Overview and Background

We start by providing an overview of the frameworks Fawkes, where the CX is currently being developed in, and ROS2, which has a broad community that could benefit from the CX. This would serve as a guideline for the similarities and differences between the two frameworks.

The rest of the sections provide information about the core methods and concepts utilized in the CLIPS Executive. A brief description of the reasoning process and the initial CX behavior is also provided.

2.1 Fawkes

Fawkes is a component-based, multi-threaded Software Framework for robotic real-time applications for various platforms and domains[NFBL10]. The main entities are plugins, which support the structuring and the implementation of the software (similar to packages in ROS, cf. Section 2.2). Each of them runs in a separate thread.

The main way of communication inside Fawkes happens through data exchange via a hybrid blackboard/messaging approach, where information is shared to a blackboard, and commands are sent via messages. The blackboard utilizes reader/writer principle, meaning that when a writer updates data, all readers receive an update message. Each data update triggers an update to the blackboard and thus allows event-based programming. Furthermore, a reader can request a specific behavior of the writer instance by sending a message over the network to that specific writer. There is a designated Fawkes main thread that creates the blackboard and stores different interfaces. Then, internal Fawkes threads (e.g., plugins) or remote apps access the blackboard memory via read/write transactions. One interface is designed to have a maximum of one writer at a time and an unspecified number of readers. A common example would be a plugin responsible for the navigation of the agent that would be registered as a writer for the Navigator interface. A second plugin, which is responsible for requesting the desired target pose would be registered as a reader and would send a ”Go-To” message with specific coordinates over the blackboard. The writer would then receive and process the message.

An important part of the Fawkes environment is the main loop. It acts as the main controlling block of the system. It manages the execution order of the threads and ensures that time constraints are met. Threads in the form of plugins are hooked to the main loop in a certain stage. The main loop runs in multiple stages in a specific order, where plugins beneath the same stage run concurrently.

Fawkes offers a Lua-based behavior engine. Users can, for example, define so-called skills (hybrid state machines), which get executed through the Skiller ex-

ecutor plugin. In this case, the mapping of the high-level actions to predefined skills inside the executor is required.

2.2 ROS(2)

The Robot Operating System (ROS) is open-source middleware, despite its name, and has been widely used for robotics applications. It is a middleware in the sense, that it provides a structured communications layer above the host operating systems[QCG+09]. It includes services designed for a heterogeneous computer cluster such as hardware abstraction, low-level device control, message-passing between processes, and package management. The main idea of ROS is that you shouldn't reinvent the wheel when programming a robot software, but stick to certain core methods and standards, as the vast majority of functionalities are already available in the package system, e.g., navigation or sensing. This speeds up the developing process immensely.

One ROS program consists of one or more packages (similar to plugins in Fawkes). The main building blocks inside each package are nodes. They are independent processes, thus code re-usability, fault isolation and modularity are strongly enhanced. The communication between nodes happens via an anonymous publish/-subscribe model. There can be several publishers and subscribers, which communicate among themselves via a unique topic. The publisher passes a message (to a certain topic), which has a simple predefined data structure. Next, all subscribers to that topic receive the message iff they expect the same message type. This is the most widely used communication model, but there are also services and actions. Services can be seen as one-to-one communication based on the request/response (Client/Server) model. Actions, on the other hand, are built on topics and services. They are very similar to services, with the key difference, that the action server provides constant feedback while executing the request. They also support requesting the cancellation of the current execution.

The significant drawback of ROS1 is that it is not very suited for real-time applications, as it required significant resources with no guarantee of process synchronization or deadlines. Therefore, its successor ROS2[OSR] was introduced. The aim of ROS2 is to provide support for small embedded platforms, real-time applications, and non-ideal network conditions. These happen through the addition of the Data Distribution Service (DDS), Zeroconf, Protocol Buffers, ZeroMQ, Redis, and WebSockets [MKA16]. The DDS communication layer serves as the lowest middleware layer that communicates with the OS and provides a publish/subscribe transport (similar to ROS topics) that allows any two DDS programs to communicate without the need for a central coordinator. This improves the transmission performance and makes the overall system quicker, more fault-tolerant, and lighter to work with.

One further addition is the lifecycle management for nodes, which enables better control over the state of the ROS environment. Figure 1 shows the design of the lifecycle state machine used for the implementation of managed nodes (lifecycle nodes). It enables to fully control the state of a ROS node. For example, all nodes initially are unconfigured. A dedicated manager can decide to configure a note, by invoking a user-implemented function (configure) and transition its state to Inactive, from where it can then activate the node and enable its communication to other nodes. This design allows the launching mechanism to ensure the proper loading of all components with respect to dependencies between the nodes, before continuing with execution.

These led us to the conclusion, that ROS2 and its newly introduced features were the better choice for such complex application as the CLIPS Executive. In addition, for developers in ROS1, who would like to use the CX, there are also interfaces, assuring backward compatibility. These are the ROS1 bridge[ROSb], which provides a network bridge that enables the exchange of messages between ROS1 and ROS2, and also action bridge[ROSa], which bridges between ROS1 action client and ROS2 action server, allowing the communication through actions.

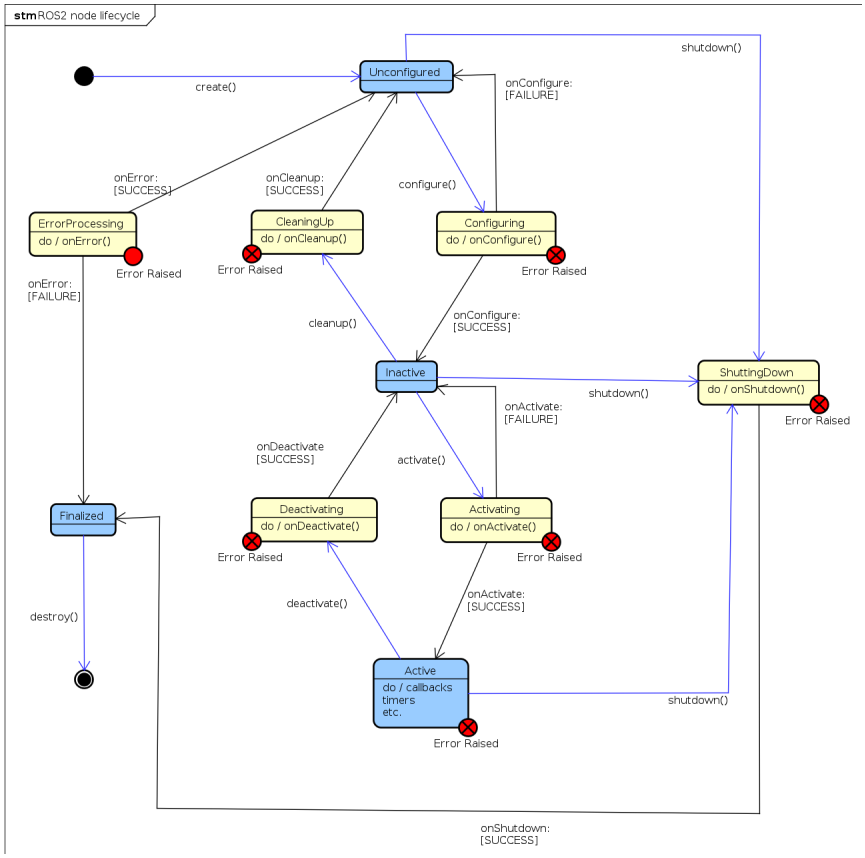

Figure 1: Managed nodes lifecycle state machine [LFS].

2.3 Goal Reasoning

Goal Reasoning (GR) in our work is based on Aha's definition (2018):

"Goal reasoning (GR) is the process by which intelligent agents continually reason about the goals they are pursuing, which may lead to goal change" [Aha18]

It is derived from the human ability to form a goal, given a limited amount of knowledge, and dynamically re-prioritize it in the occurrence of either internal or external events. The central entities of the approach are expressed as goals. The flexible and reactive nature of Goal Reasoning originates from the active formation of goals, combined with a process, in which a formulated goal goes through a set of different stages, defined as the goal lifecycle. The goal progression, as well as, exogenous factors are monitored in each of these stages, thus allowing the reasoning system to either adapt the current goal or to change it entirely. Such behavior is especially suited for autonomous Agents in a complex environment. Aha also defines two models of GR: Goal-Driven Autonomy (GDA) and goal refinement [Aha18] with the latter finding wider applicability as it allows the formation of goal constraints and thus the ability to compound more complex models.

The central idea of goal refinement is to progressively refine a goal through the addition of goal constraints. Goal Lifecycles are used as a form of goal refining [RSA+14]. Each goal goes through a set of different phases in its lifespan. The agent, for example, before committing to a goal, can either expand, re-initiate, or reject it, given the internal condition of the agent or exogenous factors such as processing sensor data. This technique enhances the ability of the agent to formulate, execute, evaluate and adjust a goal without the need for a complete model of all possible situations that could occur.

2.4 PDDL

The Planning Domain Definition Language, in short PDDL, is a planning language with the aim of providing a universal standard for AI-based planning[GKW+98a]. It is used to encode common planning tasks and to also provide separation of concerns. This way, the system can be independent of the provided planner, as the underlying planning software is interchangeable. The input planning problem is split into two instances: the domain description and the problem description.

(1) **Domain Description** A description of the behavioral capabilities of the system. This defines the domain model to be planned for. This model includes:

- **Object Types**: define the possible types of objects in the planning domain. These mainly represent actual physical entities (e.g., robots)
- **Predicates**: define the properties of the objects inside the domain

- **Actions**: define the possible ways of changing the environment. All actions include pre-conditions, which need to be satisfied, before applying the post-conditions

(2) **Problem Description**: The problem description includes the current knowledge about the system. The combination of available objects with initial state (mainly facts associated with these objects) and a pursued goal forms the representation of the problem instance. A planner is then called to provide a sequence of actions (plan) that achieves the desired goal.

2.5 CLIPS

The "C" Language Production System (CLIPS) is a portable, rule-based production system [Wyg89]. It was first developed for NASA as an expert system and uses forward chaining inference based on the Rete algorithm consisting of three building blocks[JCG]:

1. **Fact List**: The global memory of the agent. It is used as a container to store basic pieces of information about the world in the form of facts, which are usually of specific types. The fact list is constantly updated using the knowledge in the knowledge base.

2. **Knowledge Base**: It comprises heuristic knowledge in two forms:

   - **Procedural Knowledge**: An operation that leads to a certain effect. These can, for example, modify the fact base. Functions carry procedural knowledge and can also be implemented in C++. They are mainly used for the utilization of the agent, such as communication to a behavior engine. An example for procedural knowledge would be a function that calls a robot-arm driver to grasp at a target location, or a fact base update reflecting a sensor reading.
   - **Rules**: Rules play an important role in CLIPS. They can be compared to IF-THEN statements in procedural languages. They consist of several preconditions, that need to be satisfied by the current fact list for the rule to be activated, and effects in the form of procedural knowledge. When all its preconditions are satisfied, a rule is added to the agenda, which executes all the activated rules subsequently by firing the corresponding procedural knowledge.

3. **Inference Engine**: The main controlling block. It decides which rules should be executed and when. Based on the knowledge base and the fact base, it guides the execution of agenda and rules, and updates the fact base, if needed. This is performed until a stable state is reached, meaning, there

are no more activated rules. The inference engine supports different conflict resolution strategies, as multiple rules can be active at a time. For example, rules are ordered by their salience, a numeric value where a higher value means higher priority. If rules with the same salience are active at a time, they are executed in the order of their activation.

2.6 CLIPS Executive

The CLIPS Executive (CX) is a CLIPS-based production system[NHL19], which serves as a high-level controller, managing all the high-level decision making. Its main tasks involve goal formation, goal reasoning, on-demand planner invocation, goal execution and monitoring, world and agent memory (a shared database for multiple agents) information synchronization. In general, this is achieved by individual CLIPS structures (predefined facts, rules, etc.), that get added to the CLIPS environment.

2.6.1 Goal Reasoning in the CLIPS Executive

Goals are the core concept of the CX. They describe the objectives to achieve or conditions to maintain[NHL19], e.g., to bring a certain object from point A to point B. They are explicitly represented in the form of CLIPS templates, where specific rules guide the goal's execution and mode transformation during its lifecycle. The CX utilizes GR with the goal refinement mechanism[Aha18] in the form of goal lifecycles. This way, each goal goes through a predefined set of goal modes during its lifespan. This makes the program flow explicit and allows constant goal monitoring, observation of agent status and actions, as well as tracking and reacting to internal/exogenous factors. Some goals may be compound, e.g., cooking a recipe. Such a goal is comprised of several simple sub-goals, such as preparing different ingredients and heating the oven.

Figure 2 shows the goal lifecycle, through which every goal progresses over time. Nodes represent the current mode of a goal (stages of refinement).

In the initial stage, a goal is first *formulated*, meaning that it may be relevant for the agent and should be considered. A goal is then *selected* among other goals based on criteria, e.g., the goal is the most promising. Then, if the goal is compound it is expanded into sub-goals, otherwise, a course of action is determined, e.g., by invoking a planner, reaching expanded mode. Next, the goal *commits* to the produced plan or a sub-goal, acquiring all required resources. The plan or sub-goal is then executed (goal is *dispatched*). A simple goal reaches the *finished* stage when the plan execution has been completed and the outcome of the goal is determined (succeeded/failed/rejected).

The requirements to achieve a compound goal differs based on the goal type, e.g.,

a run-all goal, which runs all sub-goals and succeeds if all of them succeeded or a try-all goal, which succeeds if one of the sub-goals succeeds. Then, based on the goal outcome and former stages, the goal is *evaluated* and the world model is updated accordingly. This evaluation happens according to user-defined criteria

Finally, the goal is *retracted* by the CX, freeing all resources, plans, and everything related to the specific goal.

Note, that before *commited*, a goal can be rejected, directly going into *finished* mode. Furthermore, after evaluation, a goal can be re-initiated. With this approach, several goals can be selected and expanded. Then, the CX commits to the most promising/highest value goals and the other goals get rejected. This outlines the flexibility of the program flow that the GR mechanism offers.

To model complex objectives, compound goals may be nested into goal trees. The goal types of the inner nodes specify the handling of their respective sub-trees. Leaves of such goal trees are simple goals, which can be directly executed by determining a course of action.

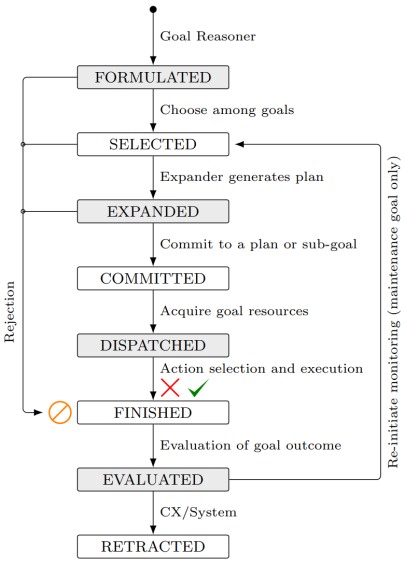

Figure 2: The goal lifecycle of CX [NHL19].

2.6.2 CX prerequisites and initialization

Before running the CX, there is a 3 stage initialization phase, which ensures that the features and files are loaded into the CLIPS environment, for the CX to function properly.

- **Stage 1**: Several features are initialized during the first stage in a certain order. Following are few examples of key CLIPS features.

  (1) **Blackboard**: The Fawkes specific hybrid blackboard (cf. Section 2.1) is loaded with the execution of the Fawkes main program. This step initializes the CLIPS access to the blackboard and adds the necessary C++ functions for communication and manipulation of the blackboard from within the CLIPS environment.

  (2) **PDDL Parser**: The CLIPS Executive uses the PDDL [GKW+98b] language as a domain and action model. Planning and execution are

based on the same common PDDL domain model in order to keep consistency[NHL18]. The parser feature is used to translate PDDL domains in CLIPS facts and rules representing the PDLL domain and problem files.

(3) **PDDL Planner**: Allows any PDDL-based planner to be integrated and used with the CX. This is possible due to the CX being able to provide the standard PDDL Domain and Problem files from the parsed domains. The agent uses it to invoke the planner.

(4) **Robot Memory**: The robot memory is a MongoDB-based replicated common database, which allows multiple agents to communicate with each other [NHL19]. The CX synchronizes the world model with the robot memory. Also, the robot memory plugin can be called to retrieve all current predicates and objects of the PDDL domain from the database and to write a PDDL Problem file from the current situation and a goal specification.

- **Stage 2**: It includes the *domain-independent* CX initialization. In this stage CLIPS fact templates and rules for goals, PDDL domain facts, etc. are loaded.
- **Stage 3**: It conducts the user-defined *domain-specific* executive initialization. Files are loaded in the given order. In this stage, several features such as domain representations, world model synchronization, custom goal reasoner and expander, execution monitoring, and executioners are loaded.

2.6.3 CX components

The CX supports separation of concerns. The following components can be distinguished inside the CLIPS Executive:

PDDL knowledge representation: Both the domain and plan model are represented using PDDL, based on its widespread use. The input domain grants knowledge about the object types, predicates, and possible actions (including their preconditions and effects). This knowledge is presented in form of facts inside the fact-base. The planning model is the current beliefs of all objects and facts, based on predicates and object types inside the domain. These beliefs are crucial for both the planning and execution process. The planner is populated with the existing knowledge relevant to its problem representation. To keep consistency throughout the planning/execution, the CX and the provided planner operate on the same PDDL domain, thus the provided actions inside the resulting plan can be executed.

World model representation: The world model includes all information concerning the internal and external environment. The world model and the planning model are synced and an update to one model is immediately sent to the other, in order to avoid inconsistent knowledge. The representation of the world model is

richer than the one in the planning model. It includes additional information concerning exogenous factors, such as other agents blocking the current path, which are not sent to the planning model, as this information is irrelevant for the planner. This information is useful during the execution and the execution monitoring of a provided plan.

Goal formation and reasoning: The developer has the flexibility of defining goals and how they change during their lifetime. A dedicated goal reasoner should be assigned the responsibility of controlling the transitions inside goals. The reasoning process is mostly connected and dependent on the current domain-specific knowledge (e.g, knowledge inside the world model). Both planning and execution are triggered based on the goal mode and the data inside the goal.

Planning: The planning component provides either dynamical or fixed planning. In the former case, the main requirement is the presence of a dedicated planner, such as a PDDL planner. The executive calls the planner, based on a selected goal. The CX provides the planner problem instance with the existing objects and facts and sets the desired goal. Upon successful planning outcome, the goal is expanded and a plan action is generated for each action inside the provided plan. These plan actions are relevant for the execution of the plan. In the latter case, the developer defines a plan and provides a fixed-sequence of plan actions.

Execution: In this component, the provided plan is executed, based on its plan actions. Multiple instances are running during the execution. First, there is an action-selection mechanism, which evaluates and selects actions, which can be executed given the current internal and external knowledge. Reason being, that the planner can indicate a plan action as executable, based on its problem file, but in reality, there can be, for example, exogenous factors or unavailable resources, which are part of the world model, that prevent the execution of that action. A selected action is then executed, using a dedicated executor. The execution is constantly being monitored, which assures reasonable reaction of the system to exogenous factors.

Interfaces between each component ensure that the relevant information is being shared between the different instances.

3 Related Work

The CX combines the principles of GR with high-level execution capabilities. There have been several systems that propose task execution mechanisms for autonomous robots. Such systems vary in implementation, but they all pursue planning and execution for agents. Despite ROS being the robotics applications standard middleware, there exists little to no such available packages. In the following, we describe the few systems known to us, which have contributed significantly in that area.

ROSPlan: It is the current standard framework for embedding a generic task planner in ROS systems[CFL$^+$15]. The main purpose of ROSPlan is to link PDDL to the ROS environment and allow the integration of different PDDL planners. It provides task planning and action execution. It consists of a knowledge base node, which gathers all relevant information about the world and the internal state of the agent. This information is then used to generate the initial state and goal and is passed to the planning system node, which constructs the PDDL problem instance and invokes the external planner. The planning system also constructs a filter for the plan, which updates the information, relevant for the problem instance, based on new information in the knowledge base. Finally, the planning system dispatches the set of actions, based on the generated plan. The planning system maps the high-level PDDL actions to low-level ROS action messages. This happens through a plan dispatcher instance, where the developer should specify how a PDDL action is to be executed in ROS.

ROS Based Planning and Execution Framework for Human-Robot Interaction:
Dondrup, C. et al. [DPNL17] introduces a system, which uses the ROSPlan framework for planning, but its main contribution is the replacement of the built-in action execution mechanism of ROSPlan with a Petri-Net Plans (PNP) for action execution. The knowledge base of ROSPlan is modified to use a MongoDB backbone. Furthermore, a bridging interface between the two frameworks is built, allowing the message passing between ROSPlan and PNP and native integration of a ROS action server. This proves, that the native execution mechanism of ROSPlan can be successfully substituted to meet the needs of the produced system.

Plansys2: It can be considered as the successor of ROSPlan for ROS2 [MGRM21] with the additional support for multi-robot execution. The ROS2 Planning System provides an efficient, robust, and easy-to-use PDDL-based planning system for ROS2 applications. The *Domain Expert* reads and holds the PDDL domain description (available types, constants, predicates, functions, and actions) and shares them with the rest of the system. The *Problem Expert* contains the current knowledge about the system's state (objects, facts, goals). This knowledge is populated by a client application, which can then call the *Planner* to generate a sequence of actions (plan) by calling an interchangeable PDDL planner based on the informa-

tion available in the Domain and Problem Expert. This plan can then be used by a client application, in our case - the CLIPS Executive, or by the native Plansys2 Executor (similar to ROSPlan). Such a planning system is crucial for the implementation of the CX as the world model is represented in a form of a PDDL domain description, shared by the executive, and the underlying planning system and its PDDL planner. For our approach, we are replacing the native Plansys2 Executor with the running CX instance similar to how [DPNL17] have utilized ROSPlan.

GOLOG and ROS Integration: Kirsch, M. et al. [KMFS20] implements a high-level controller in GOLOG and proposes a ROS interface; *golog++*. The goal of this interface according to the authors is with "*golog++* to do for GOLOG what ROSPlan does for PDDL"[KMFS20]. The approach is more modular in comparison to ROSPlan and maps the core functionalities of ROS and ROS ActionLib to *golog++* instances. For example, it strictly requires the use of the mapped ROS ActionLib to *golog++* for action execution. This way, the developer only needs sufficient knowledge in C++ and little to none in ROS.

SMACH ("State MACHine"): A ROS package, which allows the execution of high-level tasks with finite state machines instead of PDDL models. It is a ROS-independent Python library that can be used not only to build hierarchical and concurrent state machines but also any other task-state container that adheres to the provided interfaces[BC10]. It defines several state classes serving as interfaces to parts of ROS. One example would be the ServiceState, which represents the execution of a ROS service call and implements the service request/response mechanism. This way, the developer could model a state machine to interact with ROS. **RAFCON** is another finite state machines based executive[BSBD16]. It is fairly similar to SMACH, with the key difference, that it includes a GUI, which enables graphical monitoring of state machines or execution status, and even the creation and observation of new state machines or modifications to existing ones during a running execution.

While both of these enlighten the implementation of an executive in ROS significantly, state machines tend to not perform well enough in complex environments with several unpredictable exogenous factors.

Real-Time BDI Model in ROS2: a real-time multi-agent approach to improving practical reasoning, integrating the Belief-Desire-Intention (BDI) model in ROS2[AG19]. Their BDI model explicitly considers timing constraints in the actions of the agents and the interactions between them (defined by a *designer*), thus the reasoning mechanism has a natural constraint on *plan* generation/fulfillment and *desires*, which define the preconditions, deadline, priority of a goal, and the actual goal to achieve. Each real-time agent has a predefined name, *belief-set*, *desire-set* and *plan-set*, which represent the knowledge of the agent about the world, which goals it could achieve, and how it can interact with the surrounding environment to accomplish a goal[AG19]. The real-time BDI model communicates with the agent's nodes (monitoring nodes, scheduler, and executor node),

running in ROS2 to instantiate at run-time the actual beliefs, goals, and intentions of the agent. The monitoring nodes are sensing the environment/internal state and publish a message to a given topic, whenever an update is perceived. The scheduler node serves the role of the reasoner for the agent. It subscribes to all topics and updates the agent sets accordingly. As the name suggests, it schedules the next plan to be executed based on the aforementioned criterion and then feeds it to the executioner node, which executes the given plan and publishes it to a belief and goal topic. While this approach seems very promising, it still lacks the needed autonomy for our approach, as the designer should not only explicitly define the deadline and priority of a desire but also the body of a plan (actions to perform and the exact order) and the priority of the plan, which is relevant for the scheduler, when choosing a plan.

While all mentioned approaches contribute significantly to task execution for autonomous robots in ROS, there is, to the best of our knowledge, currently no GR-based executive available either in ROS1 or ROS2. Such mechanism has proven its capabilities in the Fawkes framework, therefore, our proposed implementation will present a new and proven concept to the ROS community.

4 Integrating the CLIPS Executive into ROS

This thesis presents the open-source integration of the CLIPS Executive[RCX], currently implemented in Fawkes, in ROS2. We reckon this integration will be a step in the right direction not only for the CX but also for the ROS community, based on the aforementioned reasons, thus more developers could benefit from and/or contribute to the CX and the goal reasoning ideas.

There were two pursued milestones for this integration.

(1) **Standalone ROS system**: Our goal for this milestone was to achieve a robust and reliable goal reasoning system that could be used solely in the ROS ecosystem and environment and communicate with other systems, such as Plansys2 or the Navigation2 stack[MMWGC20]. The main limitations for this milestone are the tightly coupled Fawkes components (e.g., the PDDL Parser), as well as Fawkes plugins/libraries on which the CX depends. For that purpose, we implemented standalone Fawkes-independent libraries, e.g., the PDDL Parser and Protobuf, and replaced existing dependencies with more sophisticated systems in ROS. The ultimate goal is to provide an established high-level reasoner and executioner, that could easily be combined and/or integrated into the context of another ROS robotics application. This way, ROS developers could reliably use the CX for all high-level decisions and build their own agents.

(2) **Usability in other robotic frameworks**: The second milestone is a system that could be integrated/used in another robotics framework with Fawkes being the sought-after framework. The reason originates from the fact that the CX was originally developed in Fawkes, and we plan on using the ROS2 CLIPS Executive as a possible substitution of the current CX. There are several challenging parts for the back integration with Fawkes. First, the bridging between the ROS CX and the features that remain in Fawkes (e.g., the Skiller, cf. Section 2.1), and second, further implementation of features, which are not part of the core implementation of the ROS CX, but are necessary for the agents currently implemented using the Fawkes CX, such as the agent for the RCLL[HVG+21]. By implementing the necessary features and mechanisms, the ROS CX should be capable of providing a similar behavior as the current Fawkes CLIPS Executive, thus enabling proper communication and execution flow between the ROS2 CX and the running Fawkes plugins.

The common requirement is the ability to run the CX in both single-, and multi-robot simulations and real-world robots, thus extending the applicability of the CX for multiple scenarios.

Following, we begin by giving a high-level overview of the system's design, the primary system requirements that we build upon, and the purpose behind each

implemented component. In the last section, we dive a little deeper into the core implementation of each component, the communication between each of them, and the possible connection to an external application.

4.1 System Design

The primary goal of this integration is to achieve, simultaneously, an established CLIPS-based goal planning and reasoning framework available in ROS2, and a system that can easily be used in the context of other frameworks (e.g., Fawkes) with little configuration. To fulfill that demand, we designed a system, that takes several requirements into account. Figure 3 gives an overview of the design of the system, which will be used as a baseline reference throughout this section.

4.1.1 System Design Goals

First, as a reasoning system, ensuring *reliability, predictability, and robustness* is key. This is made possible by the ROS2 core mechanics and newly added features. The addition of a DDS communication middleware, which is commonly used in mission-critical systems (battleships, aircrafts, financial systems, and many others) makes the implementation of a real-time, secure application a possibility. This is especially important for a high-level reasoner, as all information should be kept up-to-date so the appropriate commands are reliably sent from the CX and received by the underlying layer.

Robustness and predictability are secured by implementing the main CLIPS Executive nodes as ROS2 *managed nodes* (lifecycle nodes). These nodes operate on a defined lifecycle, which is based on a finite state machine with transitions between given states, thus ensuring the consistent behavior of a node from its creation to its destruction. Each state is observable over services, which also allow internal/external triggering of a transition to a state and invoking the corresponding user-implemented functionality for the given transition. These are, for example, the configuration of the node, the activation of its communication network, and on error handling and clean-up. This behavior allows the deterministic startup and control of the system. In our case, the core nodes - *CLIPS Environment Manager*, *CLIPS Features Manager*, *CLIPS Executive* and *Skill Execution nodes* are implemented as managed nodes. The responsibility for the transitions between their states is coordinated by the implemented *Lifecycle Manager*, which is used by the main system *Bringup* to assure the correct system launch (cf. Figure 3).

Next, *extendability* and *customizability* are crucial to the CX so that its functionality can be adapted comfortably to the needs of the developers. The original CX is highly configurable to individual scenarios. We designed the ROS CX based on similar concepts. One such example is the configuration of the CX over a dedicated

config file, where most of the executive's behavior is defined. Furthermore, we use established ROS2 packages in the core implementation of the CX, which offer possibilities that are not currently implemented in the Fawkes CX. To illustrate this, we utilize systems, such as Plansys2 for the planning aspect or Navigation2 inside the evaluation.

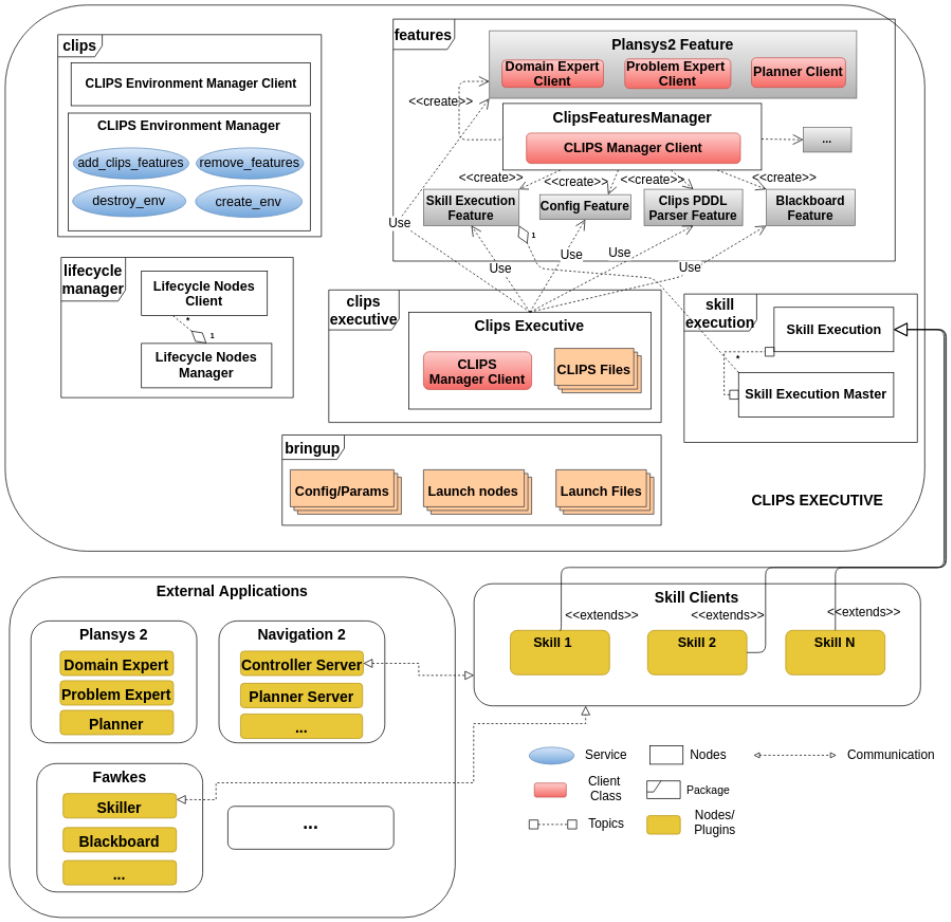

Figure 3: ROS CLIPS Executive design.

Lastly, as the CLIPS Executive is aimed to perform well in dynamic environments with multiple robots, such as the RCLL, support for *multi-robot* reasoning and execution is vital. The Fawkes CX is able to coordinate multiple agents by invoking different Skiller instances, which execute the input skill. The ROS CX supports similar behavior. We also use the term skill inside the ROS CX, but its semantics are different - skill represents the identifier of the PDDL action, for which an executor node is implemented, instead of referring to a modeled hybrid state machine within Fawkes. We implemented a skill execution mechanism to support the execution of actions on different robots. Multiple nodes in a network can implement

the same PDDL action. To indicate the robot, which will execute the skill, an agent id is specified inside the execution node (similar to namespacing). The CX starts the execution of an action based on the specified agent and skill. This is covered in-depth in Section 4.2.4.

4.1.2 Individual components

In this section, we will describe the functionality and the rationale behind each of the currently implemented ROS2 packages, which can be found in Figure 3.

Integrating CLIPS

The main thing we had to take into consideration was, how a CLIPS-based executive should function in the context of ROS2 or rather any robotics application. The CX is CLIPS-based (cf. Section 2.6), so the very first thing we needed was to secure that there is a CLIPS environment natively running in ROS. *Creating an environment* is the single most important step when embedding CLIPS into another system. Each environment maintains its own set of data structures and can be run independently of the other environments. It is a common practice to have a single initialized environment, which interacts with the rest of the system.

The common execution pattern is to have a centralized instance that first initializes CLIPS. Then all features, which are available during the current execution, are sent to that instance. Finally, upon an environment creation request, it loads the core CLIPS files and functions and notifies the newly created CLIPS environment about all available features. This completes the startup and the implemented executive is responsible for the CLIPS execution flow and the communication outside CLIPS, using the provided features.

The CLIPS Package

The CLIPS package design can be found in Figure 3. It consists of the CLIPS Environment Manager and its client wrapper, which provides ease of use for the interaction with the manager. The manager provides the functionality of the aforementioned external centralized instance, which is the main waypoint for interaction between ROS and CLIPS. It is designed as a lifecycle node and must be present at any time, while running the system, as both CLIPS and a corresponding environment are created, using the manager. As a consequence, it can be considered as the main starting point of the system. The manager is inspired by the Environment Manager inside the Fawkes CX. We adapted the logging component and the overall communication flow from and to the manager, based on ROS2 communication protocols. It fulfills several tasks, while being completely independent of the rest of the system, in order to achieve desirable decoupling from other instances such

as the features manager or the implemented executive. Its responsibilities include:

(1) **CLIPS Logging** The manager implements a custom default logger class, which is based on the logging component inside the Fawkes CX. The logging component class specifies what kind of information is received over the network and in which logging configuration it is being printed out. This information can range from the addition of facts or activation of a rule to error printing or backtracing a rule execution. The logger is accustomed to the specified logging mode of the system and the specified facts, that are being watched. This detailed tracing is extremely useful during debugging, as the developer can both target certain fact types or receive the complete CLIPS logs.

    The present logging component can easily be modified to the developer's needs. It can also be replaced with a new one or accompanied by another one, whose targeted logs are more specific. This is possible due to a priority-based mechanism, meaning that first the logs of a higher priority component are processed, before proceeding with others. The logger inside the *redefine warning feature*, which catches the re-definitions of CLIPS rules, is a good example of such customization.

(2) **CLIPS Interaction** As the manager is the main waypoint to CLIPS for the rest of the system, it provides several functions. These include environment creation/destruction and addition/removal of features. These functions are also supported and used by the corresponding CLIPS manager within Fawkes.

    The manager supports the *creation of multiple environments*, each identified with its unique name. Upon the creation of a new environment, the manager sets up all aforementioned configurations. The bare-bone environment is provided with simple utilities. These include the ability to load CLIPS files, time stamping, and others. It is also provided with a list of registered features, whose initialization can be requested at any time. On successful creation, the running environment is further accessible through the manager and can be used in other system components.

    Complementing this, *destroying a created environment* is also supported. It does the exact contrary of creating an environment. The destroyed environment can't be further accessed from other components.

    The manager also provides a function, through which an external instance *sends a list of available features* in the form of feature names, that could be requested by any CLIPS environment.

    Logically, the manager implements a way to *remove the names of registered features*.

Given its tasks, the Environment Manager should be the first available instance in the running system. All of the mentioned functions are implemented as services, so they are available both system-wide, as well as, reachable from an outside application.

To hide the complexity of implementing clients to the provided services and controlling the communication flow/outcome of the service for each instance, which needs to communicate with the manager, there is an implemented client wrapper - the *Environment Manager Client*. Its main goal is to provide functions, which realize all possible requests to the manager. It initializes a client for each provided service. It handles the complexity of creating and sending the client request to the corresponding manager service. Finally, it gives feedback on the outcome of that request. Its design purpose is to have a general client, which can be used inside or outside the system universally.

The Features Package

The features package is another essential part of the system. It is composed of the *CLIPS Features Manager* and all of the implemented CLIPS features. The design can be seen in Figure 3.

The main idea behind features is to enable the interaction between a CLIPS environment and the embedding of system/external applications with the purpose of adding functionality to the CLIPS environment. They can be seen as an application programming interface (API), which allows communication from within CLIPS with an external program. In the most common case, the feature implements several C++ functions for interaction with the specified system. The context initialization consists mainly of transforming these functions into CLIPS functions, so they can be called inside CLIPS. This mechanism is vital for a complex system, such as the CLIPS Executive, as the executive needs to control/send commands to multiple external instances. In Section 4.2 we will go into further depth of each implemented feature.

As the name suggests, the CLIPS Features Manager has the duty of controlling and managing all available features. It is also designed as a managed node to control the startup, as its main prerequisite is to have a running Environment Manager to interact with. Its tasks consist of:

1. Dynamically loading of all specified features. The Features Manager is the entity responsible for the actual creation of a feature and the control of its initialization/destruction.

2. Sending the list of available features to the Environment Manager.

3. Providing a function to the to CLIPS, which enables the CLIPS Environment to request the initialization of a given feature if the feature was previously

added to the Environment Manager. This request is sent to the Features Manager, which takes care of calling the context initialization of the requested feature if it is present inside the manager.

4. Providing the CLIPS Environment with the means to request to destroy the context of an already registered feature. This will result in the feature not being able to be accessed through the CLIPS Environment and in the system.

It allows easy loading/unloading and control over each future during its lifetime. Designing each feature to be based on a common features class and the dynamically loading of features by the Features Manager allows the user/developer to easily configure, which features are to be loaded. This simplifies the process of implementing and configuring new features significantly.

The CLIPS Executive Package

This package consists of the CLIPS Executive and several important CLIPS files, which are loaded into the created environment. It is heavily based on the CLIPS Executive plugin inside Fawkes, thus from a developer's perspective, CLIPS Executive agents do not require any changes when switching to the ROS CX.

The executive is the third core instance of the system. It is also implemented as a managed node. Both the CLIPS Environment and Features Manager can be viewed as the necessary tools to empower the capabilities of a user-defined CLIPS Environment. The CLIPS Executive is the most complex instance in our system. The CX is responsible for all high-level tasks connected to reasoning, planning, execution, and monitoring. These are carried out through the CLIPS engine. It is also responsible for communication with the outside system/systems, by the means of designated features.

The work of the CLIPS Executive can be split into two main parts. Part one consists of creating and configuring the CLIPS Executive node. Part two covers the initialization of the CX inside CLIPS and the consequent working process inside the CLIPS environment.

Part I

The idea of the CX is to provide a highly configurable executive, which can easily be adapted to individual scenarios, thus allowing the developer to experiment with different settings. Whether for testing purposes by switching between the loaded domains and files, or using the CX on a real agent, we wanted to keep the behavior as predictable and the provided configuration as modular as possible.

To accomplish the same configurability in the ROS CX as in the original CX we, took advantage of ROS2 parameters and a designated configuration file, whose parameters are mainly imported from the CLIPS Executive configuration file inside Fawkes. These parameters are used to configure different functionalities, such as

starting the executive instance in debugging mode, enabling loop events inside CLIPS, managing how and when certain instances and files are initialized in the environment. The executive enables the definition of multiple running scenarios inside the same configuration, so the developer can easily switch between agents.

The initial configuration of the CX consists of extracting parameters and requesting the default features, for example, the config feature, whose task is to parse the provided executive configuration file into CLIPS.

Part II

This part mainly covers the CLIPS-side initialization of the CX. As in the original CX, the ROS CX implements a phase-based initialization (cf. Section 2.6). A key difference is the available features, which can be requested. The ROS CX offers some similar features as the Fawkes CX but with different implementations. For example, we implemented a config feature, which is independent of the Fawkes central config library. Also, the PDDL Planner feature utilizes Plansys2 instead of Fawkes interfaces to certain PDDL Planners. During the initialization, the CX also loads several CLIPS files, which provide the functionality of the different components responsible for knowledge representation (PDDL representation), goal representation, planning, execution, and others. Section 4.2.4 covers the similarities and the major modifications between their functionality inside the ROS CX in comparison to the original CX.

The Lifecycle Manager Package

The design of this package is inspired by the Lifecycle Manager implemented in Navigation2[MMWGC20] and the ROS design suggestions for lifecycle nodes [LFD]. The package consists of the *Lifecycle Nodes Manager* and a client wrapper for a lifecycle node - the *Lifecycle Nodes Client*. The idea behind their implementation is to provide unified management of lifecycle nodes. The Lifecycle Manager is responsible for the proper configuration and triggering of wanted transition for each of the user-specified lifecycle nodes. In our system, the Lifecycle Manager manages the Environment and Features Manager, as well as the CLIPS Executive node. The manager is designed to monitor and react to either the successful or unsuccessful transition of these nodes.

The manager instance expects a list of lifecycle node IDs as input. The provided list is managed in the given order, meaning that the first node will be processed first (FIFO). This is important for dependencies between the nodes. The manager instantiates a Lifecycle Node Client for each node. The task of this client wrapper is to enable the transitioning of a managed node, as well as getting feedback for the current state. This is possible thanks to two services, implemented inside each lifecycle node instance - the service to get the current state and to trigger a possible state transition. After the initialization of all clients, a startup script is executed. The manager also provides the means to shut down, reset, pause, or resume the

system.

Having a centralized manager for the main system nodes enables the flexibility of customizing the system startup and configuration, as well as managing the expected system bring down.

The Bringup Package

The idea of this package is to give the resources to set up and start the CLIPS Executive system in CX applications. It consists of three main components:

(1) **Nodes Parameters**: The nodes parameters include all provided configuration files. They enable the developer to configure the CX system nodes to their needs, depending on the available parameters. This is especially important in the case of the CLIPS Executive. Its highly configurable nature allows the experimentation of different scenarios and settings. This behavior is comparable with the loading of the CLIPS Executive YAML configuration file inside Fawkes.

(2) **Launching files**: The launching files provide the main bringup of the system. They launch the system, based on provided launching descriptions - specific nodes, parameters, etc. This is similar to defining a CLIPS Executive meta plugin in Fawkes.

(3) **Composed startup nodes**: The composed startup nodes provide the ability to strictly control the startup flow of the system nodes and compose them in a single process, instead of launching each node separately. This ensures that CLIPS instances run within the same process, which is necessary to avoid conflicts inside different CLIPS components. They are used inside a launching file.

This package is aimed at applications that integrate the CLIPS Executive system.

This basic overview of the system and each individual component gave the fundamental basics and theoretical foundations necessary to jump into the implementation section of the ROS2 CLIPS Executive.

4.2 Implementation

In this section, we will focus on the implementation aspects of our system. We will examine the main prerequisite developing steps before integrating the CLIPS Executive in ROS. These include embedding CLIPS in ROS and providing the necessary interfaces. This section also provides a description of the system's configuration and startup, before giving a thorough explanation of its functionality,

capabilities, and program flow. Each of the main components is easily customizable or replaceable by the user, but we will focus on the current state of our system.

4.2.1 Integration Prerequisites

This development step was connected to the two main problems associated with enabling the integration of the CX in ROS. First, we needed to embed CLIPS inside ROS. The second problem was the aforementioned limitation of the Fawkes CLIPS Executive, namely, the CX is dependent on several libraries and controls several plugins, available only in Fawkes. Following, we will describe how we overcame those obstacles.

Native CLIPS in ROS

Integrating the CX starts with providing a mechanism, whose purpose is to enable the interaction with CLIPS. For this, we implemented the described centralized managing instance - the CLIPS Environment Manager. It employs the ability to initialize CLIPS and to create/destroy CLIPS environments, as well as configuring them according to the developer's needs. These functionalities are implemented in the form of ROS services, the client/server-based communication interface of ROS2. This provides easy access to the manager, from either the rest of the system or external application, thus decoupling it from the system. The Environment Manager can be seen as an API between CLIPS and ROS. Underneath, it utilizes another API - clipsmm[cli], which ensures the CLIPS-C++ bridging. Clipsmm is also utilized in the Fawkes CX. We also implemented a client wrapper for the Manager, which hides the complexity of interacting with ROS services and provides ease of use. Figure 4 shows the possible interaction between an external component, which utilises the client wrapper, and the Environment Manager.

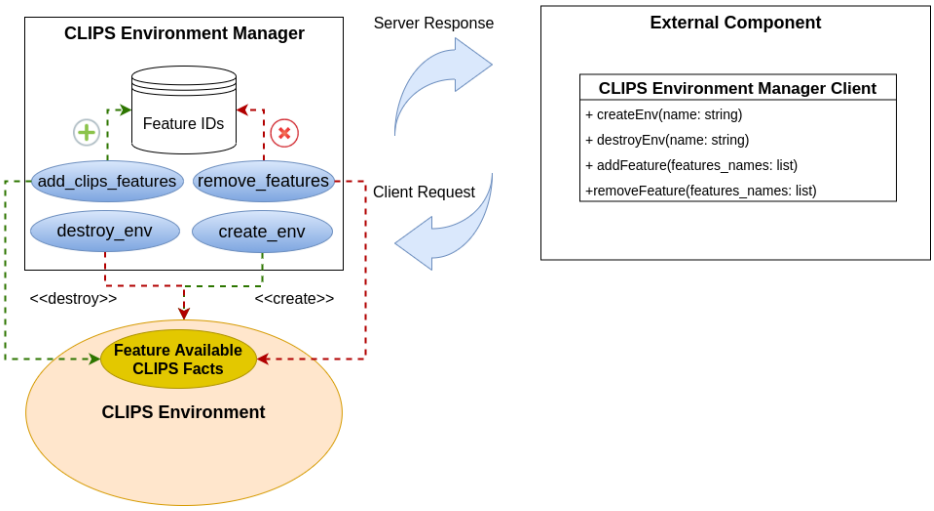

Figure 4: Interaction between the Environment Manager and the client wrapper.

Interface Libraries and Features

The Fawkes CX uses several features that are coupled to Fawkes. To keep modularity and re-usability, we implemented interfaces, that decouple the Fawkes-specific plugins and assure the correct behavior of the CX. These interfaces aim to offer a comparable behavior as in the original plugins. For instance, the Fawkes planner feature provides an interface for interaction between the CX and a provided PDDL planner. In this case, we built a feature that allows the interaction between the CX and a planning system on the side of ROS. Additionally, many of these plugins depend on Fawkes-specific libraries, which we needed to implement as standalone libraries to provide the desired functionality. In the following, we will examine the main features that we implemented for the ROS CX.

(1) **Config Feature**: The implemented config feature provides identical behavior to the one in Fawkes, but is entirely independent of the Fawkes config library. It allows the parsing of an input configuration directly into CLIPS in the form of facts. This way, the CX can parse the user-specified config file and can operate according to the provided parameters. This mechanism allows experimenting with different settings and scenarios.

(2) **PDDL Parser Feature**: The purpose of this feature is to allow the parsing of a provided PDDL domain file into specific CLIPS facts, corresponding to the specified object types, predicates, and actions. This feature is extremely important, as the knowledge about the world and the possible system behavior is build upon the provided domain. For this purpose, we decoupled the Fawkes PDDL parser library and the CLIPS PDDL parser plugin and implemented them as standalone libraries. Consequently, the CX is able to parse the provided domain and populate the fact base, which is a key prerequisite for the core CX features.

(3) **Planner Feature**: The goal of this feature is to provide the CX with the ability to interact with an external planning system/planner. Its purpose is to substitute the planner mechanism inside the original CX with a more sophisticated planning system. For this purpose, we currently utilize Plansys2, which provides several additional functionalities, such as an elaborate PDDL Planner interface compared to the rudimentary one in Fawkes, and more capable natively supported planners (e.g., temporal planner support).

(4) **Skill Execution Feature**: The purpose of this feature is to provide an interface to the new Skill Execution mechanism, provided by our system. This mechanism completely substitutes the execution mechanism of the Fawkes CX and allows the execution of a specific action inside the system, without the dependency of the Fawkes Skiller. Additionally, it provides the possibility to interface with the Fawkes Skiller, thus a skill in the ROS CX can be handled the same as a skill in the Fawkes CX. Using the provided sequence

of actions (plan) and reasoning about the executable actions, the CX creates and manages execution instances using this interface, which then requests and monitors the execution of a user-implemented skill inside the embedded system. The Skill Execution mechanism is covered in-depth in the following Section.

All features are designed as loadable plugins. This enables the user to flexibly configure, which features need to be loaded by the manager and then provided to the Environment Manager. This also enhances the ease of implementing a new feature, as they all derive from the same feature instance, thus the developer only needs to implement the actual functionality, which the feature would provide to CLIPS, and then specify the loading of this feature inside the manager's configuration. The Features Manager utilizes the Environment Client for interaction with the Environment Manager. On startup, it sends a list of available features to the manager and implements a function, which allows the CLIPS Environment to initialize the context of a specific feature. In Fawkes, the implemented CLIPS aspects provide similar functionality. Figure 5 shows how the Plansys2 feature is loaded into a CLIPS Environment.

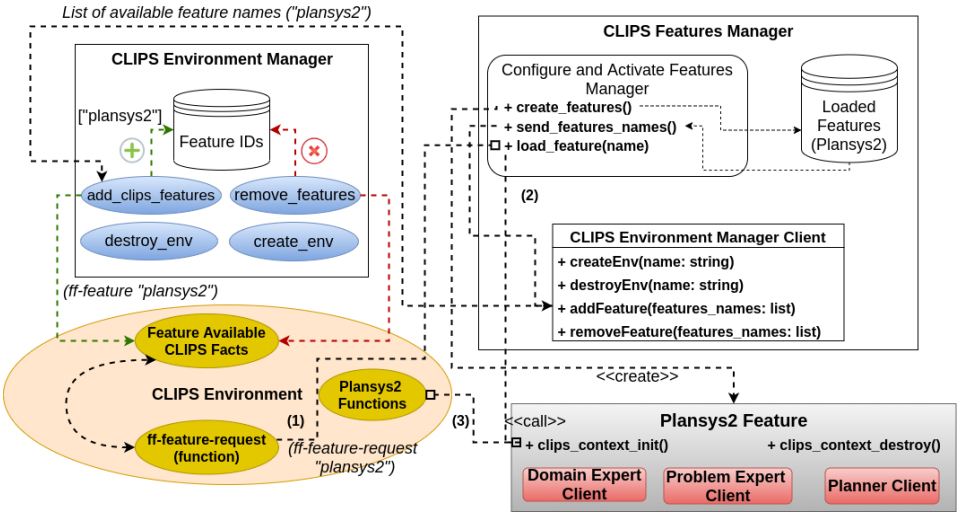

Figure 5: The interaction between the CLIPS Environment Manager, the Features Manager and a CLIPS Environment to load the Plansys2 feature.

4.2.2 The Novel Skill Execution Mechanism

The execution of skills in Fawkes is orchestrated by the Fawkes Skiller executor. This plugin, however, remains in Fawkes, as it is strongly coupled to the Fawkes core mechanisms. For this reason, we implemented a novel Skill Execution mechanism to enable the execution of skills on the side of the ROS CX, while also

enabling interfacing with the Fawkes Skiller and also keeping the execution interface within the CLIPS environment consistent between the two CXs.

The *Skill Execution* mechanism allows the binding of different executors. This mechanism is implemented inside the skill execution package of the CX and is partially inspired by the action execution approach inside Plansys2 [MGRM21]. There are two entities - the Skill Execution Master and the Skill Execution. All instances of these entities communicate over the same topic - a virtual, centralized skill board.

1. **The Skill Execution Master** - The purpose of the skill master is to provide control over the user-implemented Skill Execution nodes. It is initialized with the provided parameters for the action. These include the action name, parameters, different mappings, and an agent id, which indicates the agent to execute the action, in the case of multiple agents. It enables requesting the execution of a skill. Additionally, it provides constant monitoring of the currently executed action, as well as the option of canceling the execution of the skill. Finally, it saves the outcome provided by the execution node.

2. **The Skill Execution** - The purpose of the Skill Execution node is to provide the base class for the implementation of a specific PDDL action. Each instance corresponds to exactly one action, is provided with the execution mechanism by default, and implements the actual execution of the skill. We implemented it as a managed node so it can be easily controlled by the skill master, depending on whether the execution is required, or the node can be idling. For multi-robot scenarios, a dedicated parameter is used to specify, which id corresponds to the agent, implementing the specific action. For example, the agent robotino1 would only accept requests which include its agent id. These skill nodes are aimed to be implemented based on the system, which utilizes the CX. For example, a move action for Navigation2 would differ from the one in the case of Fawkes. Upon activation, the individual function responsible for the execution is run. We also provide the possibility to give constant feedback during the execution, as well as informing about the outcome of the execution.

The interaction between the two instances is the following (cf. Figure 6):

- The Skill Master sends a message of type REQUEST with the specific skill, parameters, and agent id.
- A Skill Execution node, which implements the action that is running on the provided agent, and is currently idle sends a RESPONSE providing its id.
- The Skill Master receives the RESPONSE and decides to either CONFIRM the execution or REJECT it.

- Upon receiving a confirmation, the execution node is activated and the execution begins, based on the implemented execution function. The node can't process further requests.
- The execution node can choose to periodically send feedback over the skill board
- The Skill Master can send a CANCEL request to abort the execution
- Finally, the execution node send information about the executed skill

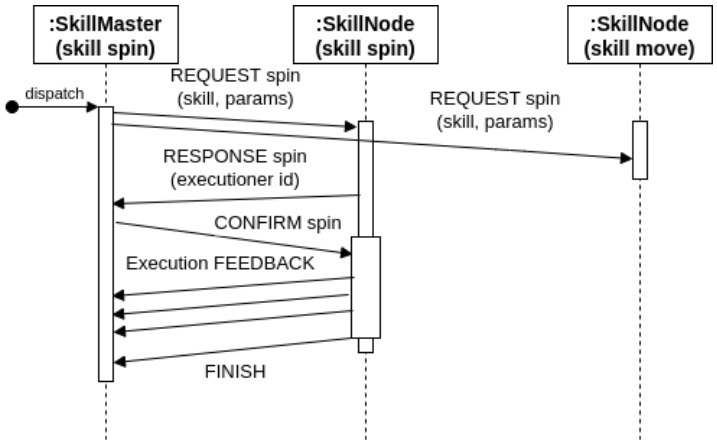

Figure 6: Skill Execution Sequence.

The skill execution mechanism is utilized in the CX for the execution of plan actions. The Skill Execution feature provides an interface, which allows the CLIPS Executive to create and control skill execution master nodes, based on the current action.

4.2.3 System Initialisation

This section gives a high-level overview of the common system startup flow and the possible system configuration. We use the implemented *bringup* package, which provides the system's main starting mechanism and configuration files.

System Configuration

Both in the original CX and the ROS CX the developer can configure different parameters:

- *CLIPS debug level* - The user can choose to enable or disable CLIPS debugging. CLIPS provides very comprehensive logs by default. The debugging includes the watching of facts, rules, activation of rules, printing in certain

functions/rules. To meet different debugging requirements the debugging is split into several predefined levels.

- *Overall CLIPS configuration* - There several parameters, which decide how CLIPS functions. There is the assertion of time each loop and the automatic retraction of goals to name a few.
- *Defining Scenarios* - This is the most important part of the executive's configuration. It allows the definition of multiple scenarios inside the same config file. Each scenario defines its own parameters, as well as, which instances to load in the specific phase of the executive's phase-based initialization. The CX loads only the specified scenario.

In the case of the Lifecycle Manager, the developer can specify the lifecycle nodes to be managed.

System Startup

The CLIPS Executive can both be started with or without the Lifecycle Manager, where the developer is responsible for managing the lifecycle nodes. Here, we will focus on the startup with a manager, as it is the more common and robust pattern.

Figure 7 represents the launching sequence of the CLIPS Executive using the Lifecycle Manager. On startup, the listed nodes are created and receive the provided configuration parameters. Initially, all managed nodes are in their unconfigured state. The responsibility of the manager is to control the flow of the provided lifecycle nodes. Upon its initialization, the manager launches a startup script. The aim of this script is to configure and then activate all nodes in the specified order. In this case, the provided order is the Environment Manager, then the Features Manager, and lastly, the CLIPS Executive. The manager constantly monitors the state of the current node and the outcome of its transition. The system is in a working state iff. all nodes have been transitioned to the active state, else system shut down would follow. During this initialization the corresponding configuring and activating functions of the provided nodes are called upon transitioning to the next state.

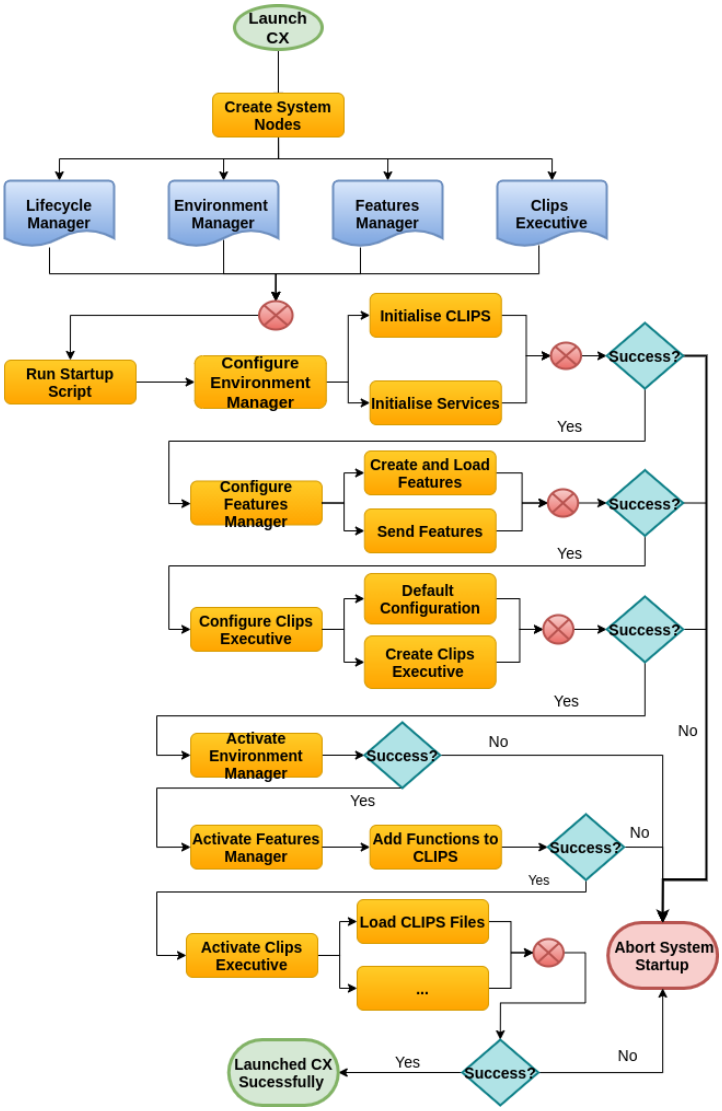

Figure 7: CX launch sequence diagram (with Lifecycle Manager). The red circle represents a synchronisation point.

4.2.4 Using The CLIPS Executive

This section gives a more in-depth description on how to use and set-up the CX inside a System. The purpose of this section is to provide the basic fundamentals, which are required to build an agent with the CX. This section can be seen as a more in-depth description of Section 2.6 and Section 2.3. Most of the described functionality is the same for both the original CX and the ROS CX, as most functions are imported from the Fawkes CX. The key functional novelties for the ROS CX are inside the planning and execution components.

We will first examine goal formation, progress, and reasoning, as goals are the core concept for the executive. They give information about the pursued objective and are the single most important entity, based on which other components function. A particular way of expanding a goal is to use an external planner, which would provide a course of actions. Therefore, we will describe the planning mechanism. Lastly, we will examine the components, responsible for the execution and monitoring of the provided plan.

Goals

Being a GR-based approach, the execution flow is determined by the formation of goals, their selection, and their refinement over their lifespan.

A dedicated goal reasoner is the main component responsible for managing goals, triggering their goal mode transitions, and making active choices in the current goal mode. This entire functionality is heavily dependent on the provided domain, the developer's needs, and the system, which utilizes the CX. This execution step is identical to the concepts and program flow, which is used in the Fawkes CX. To ease the implementation process of the goal reasoner, the CX provides a framework that allows the developer to express goals, how they change, and to formulate goal trees easily [NHL19]. The CLIPS Executive implements several goal types by default, each providing different functionality. In the case of compound goals, for example, the *try-all* goal runs sub-goals until at least one sub-goal succeeded. If none of them succeed the goal fails, thus providing disjunction functionality. In contrast, the *run-all* goal provides conjunction functionality, where all sub-goals need to succeed, whereas the *timeout* goal is used to form an input time constraint on the execution of a sub-goal. Furthermore, the representation of goals and their lifecycle is implicitly defined inside the CX.
The developer is thus left with the task of creating different goals, based on the provided goal representation. These goals should be relevant for the provided domain and describe the objective to be achieved or maintained. The goal reasoner would need to also decide what goals are relevant and to manage their progress over their lifetime. This behavior is based on the different goal modes and their functionality inside the goal lifecycle (cf. Section 2.6). A typical implementation of a goal reasoner would act as follows.
Initially, the goal reasoner formulates a set of goals, which indicates, that these goals may be relevant. It proceeds with picking one or more of these goals on a given criterion, e.g., the most promising goals or the least expensive ones. Next, it requests a course of actions (plan) for a selected goal. Upon successful generation of the plan and corresponding plan actions, the goal is expanded. Then, the goal reasoner would commit to one plan and acquire the required goal resources. Finally, the reasoner dispatches the goal, which triggers the execution of the actions based on the provided plan.

Planning

The expansion of a selected goal is connected with the generation of a suitable

plan. Currently, the CX supports a PDDL-based planner model [NHL18]. There are two main prerequisites for this model. For one, a *domain* file should be provided to the CX, which defines the behavioral capabilities of the embedding system. Secondly, the CLIPS Executive needs a way to parse this domain into its fact-base. This function is currently provided by the PDDL parser and the implemented CLIPS PDDL parser feature. Using this interface, the CX is able to extract the possible object types, predicates, and actions from the provided file, which formulate its domain model.

Currently, two ways of generating a plan are supported - using a planner or providing a fixed sequence of actions (similar to the Fawkes CX).

1. **Fixed Planning**: This type of planning is only dependent on the provided domain and the current knowledge about the world inside the CX. In this case, the developer is left with the option of defining a certain plan instance with a specific plan id, corresponding goal id, and the goal of this particular goal. The actual sequence of actions and the order of execution of these actions, which need to be executed to achieve the desired outcome of this plan, is also defined explicitly and bound to the plan. This type of planning is particularly useful for goals, whose objective can be statically achieved and does not involve many uncertainties. This way, the call to an external planner is omitted and thus the extra overhead that comes with it, especially for plans that would need a lot of processing.

   The following listing shows a very basic example of fixed planning:

```
(defrule goal-expander-create-sequence
    ?g <- (goal (mode SELECTED) (id TESTGOAL))
    =>
    (assert
      (plan (id TESTGOAL-PLAN) (goal-id TESTGOAL))
      (plan-action (id 1) (plan-id TESTGOAL-PLAN) (
    action-name move)
      (goal-id TESTGOAL) (param-values "r1" "wp_init" "wp1"))
      (plan-action (id 2) (plan-id TESTGOAL-PLAN) (
    action-name move)
      (goal-id TESTGOAL) (param-values "r1" "wp1" "wp2"))
    )
    (modify ?g (mode EXPANDED))
)
```

Listing 1: Fixed Sequence Example

   It represents a basic rule inside the goal reasoner, which would search for a selected goal with the id of TESTGOAL. If such a goal is found, it creates the plan and the plan-actions for this plan and asserts them into the fact-base, thus expanding the goal. The plan-actions are then considered by the execution mechanism in the specific order, which is indicated by their id. The purpose of this plan is to move the robot (r1) from the initial waypoint to waypoint 2.

2. **Planning with dedicated planner**: The more flexible and common way of planning is using an external PDDL planner to generate a suitable plan. To avoid any inconsistencies the external planning system and the CX are provided with the same domain description. This also guarantees that the resulting plan will actually be executable according to the given domain model.

   Before calling the planner, the problem description is generated. It is populated with the objects and facts, which are provided by the CX based on the current knowledge inside the planner model. The PDDL goal is set according to the information inside the goal, which is being expanded. Finally, the external planner is called concurrently to the running CX, which allows concurrent activities, such as planning with another planner for the same goal or the execution of another goal, while the planning is running. To increase the efficiency of the planner, it is strongly suggested to limit the knowledge inside the planner model, thus reducing the size of the problem description. This behavior is identical to the one in the Fawkes CX.

   The biggest difference in comparison to the Fawkes CX is the better planner support and overall planning flow. The planning inside the original CX is limited to few PDDL Planners, which can be called. It is also unnecessarily complex by requiring a running database just to interface with a planner. The Fawkes CX synchronizes domain and plan model data with a replicated database, which is used as an API between the CX and the planner.

   The ROS CX improves this functionality by providing wider planner support through the Plansys2 planning system and direct interaction. Plansys2 supports reliable, flexible, and multi-robot planning and execution. For our purpose, we use only the planning mechanism of the system and not the execution mechanism as the actual execution handling is realized by the CX. It brings new possibilities, such as more capable natively supported planners (e.g., temporal planner support). The implemented interface - the Plansys2 feature allows direct access to the nodes responsible for planning - the Domain Expert, Problem Expert, and the Planner.
   The typical flow is to provide the domain file to both the Domain-, and Problem Expert. Then the knowledge inside the Problem Expert is populated based on the planner model of the CX. Finally, the Planner of Plansys2 is called with the domain and problem instances. The planning process is monitored by the CX and the outcome is processed accordingly.

Plan Execution

Once the goal is expanded and the course of actions is determined, the program flow proceeds with the execution of these actions. Before executing the actual actions, there is a prepossessing stage. The reasoner picks an expanded goal, based on predefined criteria, and *commits* to that goal. The goal is then dispatched and the plan execution starts. Currently, the CX supports the execution of sequential

plans, where the plan actions are selected and executed in the provided order. The implementation of plans with concurrent actions is also possible. The execution involves several components:

A. Action Selection

The responsibility of the action-selection component is to select the next action to be executed by the executor, based on the provided order of execution. This selection takes two things into consideration.
First, it is worth noting, that the planner model used for the problem instance is a sub-set of the world model. An action, marked by the planner as executable, may in reality be infeasible in the current state, because of e.g., exogenous factors. Therefore, the preconditions of each plan action are checked against the current knowledge inside the world model, and it is marked as executable iff. these preconditions are met. This ensures consistency between the planner model and the world model during execution.
The action-selection then picks the next plan action based on predefined criteria (e.g., smallest plan id). The state of this action is then changed to *pending*, which signals the execution of this action.

B. Action Execution

Figure 8 shows the interaction during a skill execution.

The CX creates a skill master node, providing the skill, its parameters, and agent id. Then, the executive uses the master to request a skill execution over the skill board. If an execution node for this skill is implemented and is currently idle, a response is sent, which the master node can decide to reject or confirm. Upon confirmation, the execution starts. The skill execution is running concurrently to the CX, thus allowing the executive to continue working on tasks, such as monitoring the execution or evaluating the executability of the next action(s). The Skill Execution feature constantly monitors the information inside the master node and passes it to the CX as feedback. The CX can also decide to abort the execution at any time. When the execution is finished, either successful or it has failed, the execution node should send the according finishing message and the outcome of the execution is provided to the CX. The task of implementing an execution node for the external system is left to the developer, as it is strongly dependent on the components inside the system.

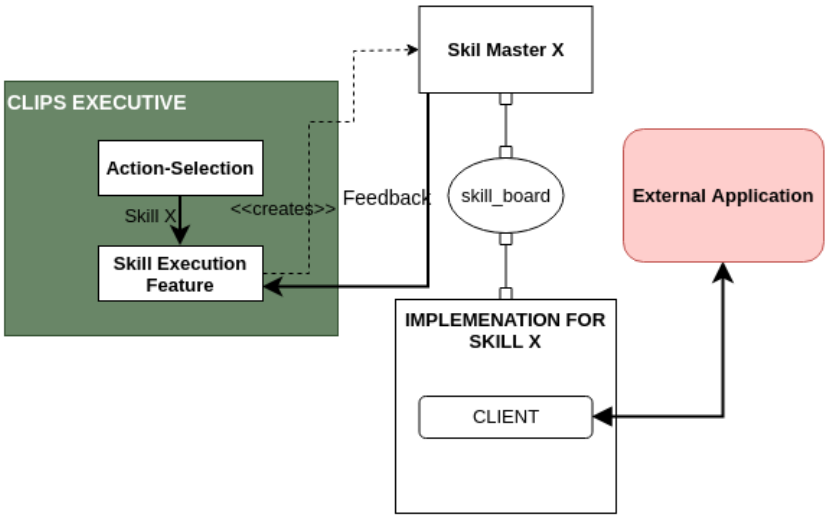

Figure 8: Shows the execution flow of a mockup skill X. The CX starts the execution using the Skill Execution feature. Execution Node is implemented for skill X, which determines how to execute the skill based on the external application.

C. Sensing and Execution Monitoring

We ported the sensing and execution monitoring from the Fawkes CX.

Upon successful execution of actions, its effects are directly applied to the world model by default. To alter this behavior, the CX supports *sensed effects*. In this case, the CX waits for the actual observation and confirmation of the sensed effect before applying the remaining effects and transitioning the goal to the *final* state.

The execution of the plan is continuously being observed. This allows reacting to effects, resulting from exogenous events or a failed execution of the current plan. In the occurrence of such events, the agent would need to react accordingly. The implemented execution monitoring provides three ways of dealing with such situations. First is to *retry* the failed action, based on reasoning on gathered information, such as the number of failures and others. Next, the agent can decide to *abort* the current plan preemptively. This is useful, in case the plan is no longer feasible, or a timeout is reached. Lastly, instead of failing the goal, execution monitoring allows the adaptation of the current plan, for example, by providing additional actions or reordering current ones. The execution monitoring is also not limited to these three ways. Programmers can easily customize and extend execution monitoring according to their needs.

4.2.5 Interfaces for Fawkes

Our last goal for the implementation was to provide a mechanism, which would enable the communication from the ROS CX to Fawkes. This way, the ROS CX will be able to provide comparable behavior and ideally be capable of substituting the Fawkes CX upon implementation of interfaces on the side of Fawkes.

As we mentioned, several Fawkes plugins interact with the running instance of the CLIPS Executive. This interaction happens primarily over the dedicated Fawkes blackboard (the central communication medium between Fawkes components). The interface between the blackboard and the CX is provided by the blackboard feature, which allows the CX to make transactions, such as writing, reading, or sending messages over the blackboard. Typically, when needing to communicate and receive information from a certain plugin, the CX creates a reader for the corresponding feature. This allows the CX to continuously read the interface information, while also enabling it to control the plugin by passing a predefined message. For example, the CX would create a reader instance for the SkillerInterface, whose writer is an instance of a Fawkes Skiller that executes the skill. The CX can then send messages to start the execution of a specific goal.

In our implementation, we provided a separate package, which mimics a blackboard with a reader/writer principal. We provided a common Interface class, which works based on implemented interfaces on the side of ROS2. These interfaces imitate the interfaces implemented in Fawkes. The actual communication is realized over dedicated channels for the different interfaces, based on registered readers and writers. As in Fawkes, this communication is realized over one-to-many relation, with a writer being the single instance. Each interface also defines commands in the form of messages, which a reader instance can send to the writer. Furthermore, we implemented a blackboard feature, which allows the CX instance the same behavior capabilities as in the Fawkes blackboard feature.

This allows to mimic the behavior of a blackboard on the side of ROS and enable the interaction between the ROS CX with the running instance of Fawkes.

5 Evaluation

In order to prove the capabilities of the ROS CX, we split the evaluation of our system into two parts - proof of concept in the context of ROS2, using a specific problem in the domain of logistics robotics and common ROS platforms. This experiment proves the ability of the CX to be used as a standalone system inside ROS, as well as, its ability to interact and be integrated with other ROS systems, such as the Navigation2 stack and Plansys2. The ability of the CX to interact with other complex ROS packages, which offer a variety of features that are currently not supported in the Fawkes CX, such as multi-agent navigation and more comprehensive planning support, further improve the capabilities of the ROS CX compared to the Fawkes CX. The second proof of concept is in the context of the Fawkes framework, using dedicated test-scenarios, which the Fawkes CX already provides to demonstrate its capabilities and usage. This thus prove the ability of the ROS CX to run the same test-scenarios and to interact with Fawkes, which will serve as the baseline towards the ongoing work on being capable of substituting the current CX inside more complex, real-world agents, such as the RCLL agent [HVG+21].

5.1 Proof of concept in ROS2

In the following, we will describe the experimental setup used for this evaluation, before examining the actual experiment.

5.1.1 Experimental Setup

To perform the evaluation we used a combination of multiple renowned and established platforms throughout the ROS ecosystem:

- **Navigation2** : The successor of ROS Navigation, which is the most commonly used 2D navigation software framework. It controls and enables robots to move autonomously and reach a certain destination. Based on the input information, it generates a plan and outputs command, which drive the robot movement with respect to exogenous factors. It also provides support for multi-robot navigation and offers different simulation setups. [MMWGC20]
- **Gazebo**: 3D robot simulator, which offers rich visualization for physical simulation with support for sensors and cameras. It also provides several models and environments, as well as interfaces, which are used to bind a specific program. [KH04]

- **TurtleBot3 (TB3)**: A small, programmable, ROS-based mobile robot. It supports a simulation development environment that can be programmed and developed with a virtual robot in the simulation. It can be used to run Gazebo simulations in combination with Navigation2 [TB3]
- **Rviz2**: ROS2 GUI, which allows the visualization of data, coming from ROS components. It can use the received data to create and render a 3D map or to render a robot when a robot model is provided.
- **Plansys2**: The successor of ROSPlan for ROS2. Further information can be found in Section 3.

The Navigation2 stack provides a highly configurable and comprehensive simulation setup. The *Navigation2 bringup* package can be used to launch Navigation2 in simulation with Rviz2, Gazebo, and a specific robot (in our case TurtleBot3). The developer is free to adjust the provided configuration files or edit default ones by providing, for example, specific gazebo world, robot models, and adjusting parameters, such as the initial pose of the robot, when a simulation started.

We decided to use the Navigation2 simulation setup for the experiment, as it utilizes different robotics frameworks and it gives a detailed overview of the execution flow. By integrating the CX as the high-level controller for this simulation, we can show the ease of use within the ROS ecosystem and the wide applicability of the system.

5.1.2 Experiment

In this experiment, we combine the CLIPS Executive and Plansys2. For this purpose, both systems are launched and activated by the Lifecycle Manager. The CX requests the Plansys2 feature, which enables the interaction to Plansys2, which is used for the generation of plans. Furthermore, the implemented Skill Execution mechanism is utilized by the executive, based on the functionality, provided by the Skill Execution feature. This allows the CX to start the execution of skills inside the running Navigation2 instance, which executes the actual driving of the robot inside the simulation.

We created a dedicated PDDL domain file. The domain defines two types - robot and waypoint. The waypoint represents a specific position, to which the robot can navigate. We defined two predicates, which indicate the current position of the robot and a possible connection between two waypoints. The implemented PDDL action (move) navigates the robot from one waypoint to another. The two preconditions are that the robot is currently *at* the provided waypoint and this waypoint is *connected* to the desired waypoint. The post-condition is the robot being at the desired waypoint.

Initially, the domain is loaded into the executive using the CLIPS PDDL parser feature. The Domain and Problem Expert of Plansys2 are provided with the same

domain file. The test domain populates the initial knowledge about facts and objects based on the provided domain, which forms the plan model. The knowledge for this experiment represents a single robot (named ”tb3”) and 8 waypoints (”wp-init”, ”wp1” to ”wp-6” and ”wp-final”).

We defined a simple goal, whose objective is to bring the robot (tb3) from the initial waypoint (”wp-init”) towards the final waypoint (”wp-final”). The two waypoints are not directly connected, thus the robot would need to navigate through all waypoints to achieve this goal. The selected goal is expanded by the CX by providing the current knowledge to Plansys2 and calling its planner with the desired goal (in this case: ”robot-at tb3 wp-final”). The provided plan is parsed in the form of plan actions inside the CX. The robot executes 7 plan actions (thus 7 skills) in total. These include moving from the initial waypoint to the 1st waypoint, then to the 2nd waypoint, and so on to the final waypoint (”wp-final”). Figure 9 shows the provided knowledge to Plansys2, as well as the resulting plan.

```
> get problem instances
Instances: 9
       tb3     robot
       wp_init waypoint
       wp1     waypoint
       wp2     waypoint
       wp3     waypoint
       wp4     waypoint
       wp5     waypoint
       wp6     waypoint
       wp_final        waypoint
> get problem predicates
Predicates: 8
(robot_at tb3 wp_init)
(connected wp_init wp1)
(connected wp1 wp2)
(connected wp2 wp3)
(connected wp3 wp4)
(connected wp4 wp5)
(connected wp5 wp6)
(connected wp6 wp_final)
> get problem goal
Goal: (and (robot_at tb3 wp_final))
> get plan
plan:
0       (move tb3 wp_init wp1)  0.001
0.001   (move tb3 wp1 wp2)      0.001
0.002   (move tb3 wp2 wp3)      0.001
0.003   (move tb3 wp3 wp4)      0.001
0.004   (move tb3 wp4 wp5)      0.001
0.005   (move tb3 wp5 wp6)      0.001
0.006   (move tb3 wp6 wp_final) 0.001
```

Figure 9: Plansys2 Terminal Output.

The CX interacts with Navigation2 over a dedicated execution node, which implements the *move* action. This node acts as a client for the Navigation2 server, responsible for the navigation to a specific pose, and defines the coordinates of the different waypoints. The executive creates a skill master node for the current plan action and requests the execution of the action, providing the desired waypoint. The move node passes the desired pose to Navigation2, which executes the navigation on the TurtleBot3, while also visualizing the current path in Gazebo and in Rviz2 (cf. Figure 10). The execution finishes, when the TurtleBot3 reaches the final waypoint (”wp-final”).

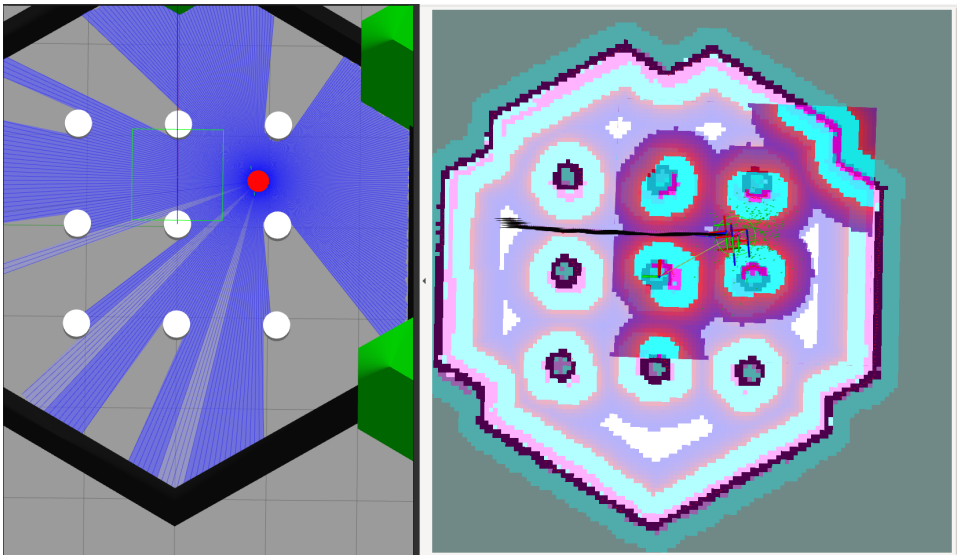

Figure 10: Navigation2 and TB3 simulation (Gazebo left, Rviz right). The red dot corresponds to the agent, the white dots to obstacles (on the left), and the black line to the current planned path navigation to a pose (on the right). The pose is provided by the CX.

This experiment's main conclusion is the ability of the CX to be used, independently of Fawkes, as a standalone ROS application and the ability to interact with other ROS projects. It also shows the direct interaction with planning system inside the ROS CX instead of depending on an interaction over a database. Furthermore, the experiment has shown that the CX can execute actions on other ROS systems, in this case, Navigation2 with TB3, simply by providing a domain file to the CX and implementing an execution node for each PDDL action. The capabilities of the ROS CX can be further improved in comparison to the Fawkes CX, as it is able to be integrated into other complex systems. For example, the Navigation2 stack provides multi-robot navigation, which is not implemented in the original CX. This hence extends its usability in different scenarios.

5.2 Proof of concept with Fawkes

5.2.1 Experimental Setup

For this evaluation, we will start the Fawkes instance, which implicitly loads the blackboard, and use the *Skiller* plugin. The Fawkes Skiller is the main plugin inside Fawkes, which is responsible for the execution of skills on a hardware level. It passes the mapped skill (cf. Section 2.1) to the Lua-based behavior engine, which dictates the actual execution. The prerequisite for the execution is the user

implementation of that skill inside Lua. For example, for a PDDL action 'say-hello' a dedicated skill "say" will be implemented. Then, a skill mapping of that action the say skill would be required. One such mapping would be:

```
"say-hello": "say{text="Hello ?(name)y"}"
```

The name will then be substituted with the provided object's name inside the action.

The original CX provides two testing scenarios, namely, *test-scenario and test-scenario-pddl*. In the context of Fawkes, these tests serve the purpose of validating the interaction between the Fawkes CX and the Skiller, as well as, the Planner plugin. The former test aims to prove the ability of the CX to parse a PDDL domain file, demonstrate the usage of goal trees, and execute a predefined sequence of actions by interacting with the Skiller. The later test improves on this by using a dedicated planner to generate the sequence of actions, instead of providing a predefined plan.

We decided that these two scenarios provide the ideal set-up to prove the interaction between the ROS CX and Fawkes, as well as the capabilities of the ROS CX to completely substitute the Fawkes CX and provide identical behavior in those scenarios.

5.2.2 Experiment

For the experiment, we used the same testing scenarios as their corresponding definitions inside Fawkes. For both tests, the CX utilizes the implemented blackboard-like mechanism and the blackboard feature on the side of ROS2, which enables the interaction with Fawkes.

Test-Scenario The provided domain for this test represents a simple hello world domain, which enables two objects to talk to each other. A simple goal is created, which is expanded by a fixed sequence of plan actions. The plan essentially contains numerous "speech" actions, such as saying hello and goodbye. For these actions, a skill inside Fawkes is implemented, which uses a library to transmit the speech over the computer's speakers.
In Fawkes, the CX processes all plan actions sequentially and sends a message to request the execution of the current skill over the Fawkes blackboard (inside the Skiller interface) to the Skiller, which then executes the skill, and constantly sends updates about the current execution. The ROS CX also processes these plan actions sequentially but instead sends an execution command over the ROS2 CX blackboard to the Skiller interface on the side of ROS2. We implemented a Fawkes plugin, which receives the upcoming skill execution messages(acting as a writer to the ROS2 interface) and passes them over the Fawkes blackboard (using the Skiller

interface) to the running Skiller. The execution updates are passed to the plugin, which writes in the ROS2 blackboard, from where the CX reads the updated information and acts accordingly.
Using this mechanism, the CX is able to interact with Fawkes and execute all skills as the Fawkes CX does.

Test-Scenario-Pddl The key difference of this test compared to the previous is that a dedicated planner is used to plan and provide the sequence of actions. The rest of the test is the same. To just be able to call a planner for the generation of the plan, the Fawkes CX requires the MongoDB plugin, the Robot Memory plugin, which provides the API between the replicated database and the CX, synchronization of the world-model to that database, and the actual planner plugin. In contrast, the ROS CX only requires the Plansys2 feature and running the Plansys2 framework to do the same. The CX just passes the initial knowledge and the domain file to the Problem Expert and calls the planner to generate the plan, which is then parsed inside the CX. From then, the CX proceeds as in the previous test.

The experiment's main conclusion is the possibility of substituting the Fawkes CX entirely with the ROS CX. The integrated CX offers new possibilities, such as planning with a dedicated system, namely, Plansys2, and the possibility of multi-agent path-planning (using Navigation2).

This proof of concept also provides the fundamental basis for the ongoing work towards the ability to interchange between the Fawkes CX and the ROS CX inside complex real-world agents, such as the RCLL agent [HVG+21]. The key strength of the ROS CX is the possibility to implement new features, based on its ability to interact with other ROS systems, which provide different functionalities. This can be significant for the RCLL agent. To achieve this substitution, the implementation of several interfaces is still required. Their purpose will be to provide a comparable interaction between the ROS CX and other plugins to the one of the Fawkes CX, thus making the substitution a possibility. Our ultimate goal for the ROS CX in connection to Fawkes is a fully functional CX in the context of the aforementioned RCLL agent which is executed on a real robot.

6 Conclusion

In this thesis, we complement the current goal reasoning approach of the Fawkes CLIPS Executive by integrating its state-of-the-art functionalities into ROS. The CX is a well-established CLIPS-based system, implemented in Fawkes, which is capable of coordinating multiple agents and reason in both dynamic and static environments. Its main capabilities build upon the ideas of goal reasoning, where the execution flow revolves around the progression of goals based on their specified lifecycle. The constant monitoring enables goals to be either re-evaluated or entirely substituted by a more promising goal. This program flow is especially suited for intelligent agents operating in dynamic settings.

With this thesis, we tackled the CX's main limitation, namely, being tightly coupled to the Fawkes robotics framework, thus being only usable in software stacks and agents that are strictly dependent on Fawkes. We solved this issue by integrating the CX into the ROS ecosystem, which provides a highly supportive community, widespread use, and several different components. Our implementation exploits the core new ROS2 features, as well as, existing community tools.

We implemented central components, which enable the interaction between an external system and CLIPS, as well as a manager for the built interfaces (features), which allow the CLIPS environment to communicate with and use external applications. We combined the functionality, provided by these components, with ROS2 concepts, and implemented a highly configurable and robust CLIPS Executive to conduct all high-level decisions inside a system. The CX provides separation of concerns. Being embedded in ROS2, we showed how the CX can benefit from available community packages, such as Plansys2, which can be used by the CX as an underlying planning system, or Navigation2 which offers rich multi-agent navigation capabilities that are currently not available in Fawkes. Our system also integrates the ability to execute plan actions. The CX employs comprehensive reasoning about the next executable action. We also provided a mechanism, which enables the executive to request and monitor the execution of action provided by a client application (e.g., move action with Navigation2).

Put briefly, we presented a full integration of the current CX in the ROS ecosystem, following the core mechanisms behind the Fawkes CLIPS Executive, and adapting them in ROS accordingly. We implemented the CLIPS Executive as a standalone ROS2 application to be utilized on different ROS systems for both simulated and real-world agents. To validate this, we used the CX as a high-level tasks controller of the Navigation2 stack, which utilizes the TurtleBot3 robot in a Gazebo simulation. Furthermore, we provided a mechanism, which enables the interaction between the ROS2 CX and the Fawkes framework, as well as the implementation of several standalone libraries of former Fawkes-specific libraries. Finally, we described the necessary means to substitute the Fawkes CX with the ROS CX.

This implementation will open several avenues for future improvements and acces-

sibility, derived from the supportive and lively ROS community and novel mechanism of GR for ROS, respectively.

6.1 Future Work

As a next step in improving the ROS CX, we will provide additional features, which will enrich the executive's functionality. One such example is MongoDB interaction support. For this purpose, we will implement a standalone library, which will ensure the interaction between a MongoDB database and the CX. This will then allow to port the multi-agent coordination possibilities from the original CX. Additionally, we plan on further testing the ROS CX in the ROS ecosystem on new agents, which will utilize other ROS frameworks that provide new functionality in comparison to the one in Fawkes (e.g., Navigation2 multi-agent path planning).

As we mentioned, there is ongoing work of providing the means to be able to substitute the Fawkes CX inside the RCLL agent with the ROS CX. For this, we still need to decouple Fawkes instances and implement the means to interact with the ROS CX on the side of Fawkes. These include the aforementioned MongoDB interaction support, further implementation of blackboard interfaces on the side of the ROS CX, and plugins that will interact with those interfaces. Furthermore, there is ongoing work for the RCLL agent towards replacing the current utilization of the Move Base framework with Navigation2. In this case, the ROS CX will provide better support with direct interaction with the Navigation2 stack in comparison to the built Fawkes interfaces, which enable the interaction with the Navigation2 framework.

Our ultimate goal is to actually replace the Fawkes CX with the ROS CX because we would rather utilize the ROS CX as it is easier to access for developers due to the popularity of ROS compared to Fawkes. Furthermore, it will leave us with the task of maintaining only one system.

Bibliography

[AG19] Francesco Alzetta and P. Giorgini. Towards a Real-Time BDI Model for ROS 2. In *WOA*, 2019.

[Aha18] D. Aha. Goal Reasoning: Foundations, Emerging Applications, and Prospects. *AI Mag.*, 39:3–24, 2018.

[BC10] Jonathan Bohren and Steve Cousins. The SMACH High-Level Executive [ROS News]. *IEEE Robotics Automation Magazine*, 17(4):18–20, 2010.

[BSBD16] Sebastian G. Brunner, Franz Steinmetz, Rico Belder, and Andreas Dömel. RAFCON: A graphical tool for engineering complex, robotic tasks. In *2016 IEEE/RSJ International Conference on Intelligent Robots and Systems (IROS)*, pages 3283–3290, 2016.

[CFL+15] Michael Cashmore, Maria Fox, Derek Long, Daniele Magazzeni, Bram Ridder, Arnau Carrera, N. Palomeras, N. Hurtós, and Marc Carreras. Rosplan: Planning in the robot operating system. *Proceedings International Conference on Automated Planning and Scheduling, ICAPS*, 2015:333–341, 01 2015.

[cli] clipsmm: C++ bindings for CLIPS. `https://github.com/timn/clipsmm`.

[DPNL17] Christian Dondrup, Ioannis Papaioannou, Jekaterina Novikova, and Oliver Lemon. Introducing a ROS based planning and execution framework for human-robot interaction. pages 27–28, 11 2017.

[GKW+98a] Malik Ghallab, Craig Knoblock, David Wilkins, Anthony Barrett, Dave Christianson, Marc Friedman, Chung Kwok, Keith Golden, Scott Penberthy, David Smith, Ying Sun, and Daniel Weld. PDDL - The Planning Domain Definition Language. 08 1998.

[GKW+98b] Malik Ghallab, Craig Knoblock, David Wilkins, Anthony Barrett, Dave Christianson, Marc Friedman, Chung Kwok, Keith Golden, Scott Penberthy, David Smith, Ying Sun, and Daniel Weld. PDDL - The Planning Domain Definition Language. 08 1998.

[HLM+19] Till Hofmann, Nicolas Limpert, Victor Mataré, Alexander Ferrein, and Gerhard Lakemeyer. Winning the RoboCup Logistics League with Fast Navigation, Precise Manipulation, and Robust Goal Reasoning. In *RoboCup 2019: Robot World Cup XXIII*, pages 504–516. Springer International Publishing, 2019.

[HVG+21] Till Hofmann, Tarik Viehmann, Mostafa Gomaa, Daniel Habering, Tim Niemueller, and Gerhard Lakemeyer. Multi-Agent Goal Reasoning with the CLIPS Executive in the Robocup Logistics League. In *Proceedings of the 13th International Conference on Agents and Artifical Intelligence (ICAART)*, 2021.

[JCG] Ph.D. Joseph C. Giarratano. CLIPS User Manual. `http://clipsrules.sourceforge.net/documentation/v640/ug.pdf`.

[KH04] N. Koenig and A. Howard. Design and use paradigms for Gazebo, an open-source multi-robot simulator. In *2004 IEEE/RSJ International Conference on Intelligent Robots and Systems (IROS) (IEEE Cat. No.04CH37566)*, volume 3, pages 2149–2154 vol.3, 2004.

[KMFS20] Maximillian Kirsch, Victor Mataré, Alexander Ferrein, and Stefan Schiffer. Integrating golog++ and ROS for Practical and Portable High-level Control. pages 692–699, 01 2020.

[LFD] ROS managed nodes design. `https://design.ros2.org/articles/node_lifecycle.html`.

[LFS] Managed nodes. `https://design.ros2.org/img/node_lifecycle/life_cycle_sm.png`.

[MGRM21] Francisco Martín, Jonatan Ginés, Francisco J. Rodríguez, and Vicente Matellán. PlanSys2: A Planning System Framework for ROS2. In *IEEE/RSJ International Conference on Intelligent Robots and Systems, IROS 2021, Prague, Czech Republic, September 27 - October 1, 2021*. IEEE, 2021.

[MKA16] Yuya Maruyama, Shinpei Kato, and Takuya Azumi. Exploring the performance of ROS2. pages 1–10, 10 2016.

[MMWGC20] Steve Macenski, Francisco Martín, Ruffin White, and Jonatan Ginés Clavero. The Marathon 2: A Navigation System. In *2020 IEEE/RSJ International Conference on Intelligent Robots and Systems (IROS)*, 2020.

[NFBL10] Tim Niemueller, Alexander Ferrein, Daniel Beck, and Gerhard Lakemeyer. Design Principles of the Component-Based Robot Software Framework Fawkes. pages 300–311, 11 2010.

[NHL18] T. Niemueller, Till Hofmann, and G. Lakemeyer. CLIPS-based Execution for PDDL Planners. 2018.

[NHL19] Tim Niemueller, Till Hofmann, and Gerhard Lakemeyer. Goal Reasoning in the CLIPS Executive for Integrated Planning and Execution. 07 2019.

[NLF15] Tim Niemueller, Gerhard Lakemeyer, and Alexander Ferrein. The RoboCup Logistics League as a Benchmark for Planning in Robotics. 07 2015.

[OSR] OSRF. Open Source Robotics Foundation. ROS2 https://github.com/ros2.

[QCG+09] Morgan Quigley, Ken Conley, Brian Gerkey, Josh Faust, Tully Foote, Jeremy Leibs, Rob Wheeler, and Andrew Ng. ROS: an open-source Robot Operating System. volume 3, 01 2009.

[RCX] ROS CX github repo. `https://github.com/fawkesrobotics/ros2-clips-executive/tree/master`.

[ROSa] ROS1 action bridge. `https://github.com/ipa-hsd/action_bridge`.

[ROSb] ROS1 bridge. `https://github.com/ros2/ros1_bridge`.

[RSA+14] M. Roberts, Vattam S., R. Alford, B. Auslander, J. Karneeb, M. Molineaux, T. Apker, M. Wilson, J. McMahon, and D. W. Aha. Iterative Goal Refinement for Robotics. In *1st ICAPS Workshop on Planning in Robotics (PlanRob)*, 2014.

[TB3] TB3 manual. `https://emanual.robotis.com/docs/en/platform/turtlebot3/simulation/`.

[Wyg89] Robert M. Wygant. CLIPS — A powerful development and delivery expert system tool. *Computers and Industrial Engineering*, 17(1):546–549, 1989.