RHEINISCH-WESTFÄLISCHE TECHNISCHE HOCHSCHULE AACHEN
KNOWLEDGE-BASED SYSTEMS GROUP
PROF. GERHARD LAKEMEYER, PH. D.

Master's Thesis

Using Reinforcement Learning as Goal Selector in Goal Reasoning

Sonja Ginter

March 8, 2023

Advisors: Tarik Viehmann, M.Sc., Till Hofmann, M.Sc.
Supervisors: Prof. Gerhard Lakemeyer, Ph.D.

Zentrales Prüfungsamt/Central Examination Office

Eidesstattliche Versicherung
Statutory Declaration in Lieu of an Oath

Ginter, Sonja — Name, Vorname/Last Name, First Name

381667 — Matrikelnummer (freiwillige Angabe) / Matriculation No. (optional)

Ich versichere hiermit an Eides Statt, dass ich die vorliegende Arbeit/Bachelorarbeit/ Masterarbeit* mit dem Titel

I hereby declare in lieu of an oath that I have completed the present paper/Bachelor thesis/Master thesis* entitled

Using Reinforcement Learning as Goal Selector in Goal Reasoning

selbstständig und ohne unzulässige fremde Hilfe (insbes. akademisches Ghostwriting) erbracht habe. Ich habe keine anderen als die angegebenen Quellen und Hilfsmittel benutzt. Für den Fall, dass die Arbeit zusätzlich auf einem Datenträger eingereicht wird, erkläre ich, dass die schriftliche und die elektronische Form vollständig übereinstimmen. Die Arbeit hat in gleicher oder ähnlicher Form noch keiner Prüfungsbehörde vorgelegen.

independently and without illegitimate assistance from third parties (such as academic ghostwriters). I have used no other than the specified sources and aids. In case that the thesis is additionally submitted in an electronic format, I declare that the written and electronic versions are fully identical. The thesis has not been submitted to any examination body in this, or similar, form.

Aachen, 08.03.23 — Ort, Datum/City, Date

Unterschrift/Signature

*Nichtzutreffendes bitte streichen

*Please delete as appropriate

Belehrung:
Official Notification:

§ 156 StGB: Falsche Versicherung an Eides Statt
Wer vor einer zur Abnahme einer Versicherung an Eides Statt zuständigen Behörde eine solche Versicherung falsch abgibt oder unter Berufung auf eine solche Versicherung falsch aussagt, wird mit Freiheitsstrafe bis zu drei Jahren oder mit Geldstrafe bestraft.

Para. 156 StGB (German Criminal Code): False Statutory Declarations
Whoever before a public authority competent to administer statutory declarations falsely makes such a declaration or falsely testifies while referring to such a declaration shall be liable to imprisonment not exceeding three years or a fine.

§ 161 StGB: Fahrlässiger Falscheid; fahrlässige falsche Versicherung an Eides Statt
(1) Wenn eine der in den §§ 154 bis 156 bezeichneten Handlungen aus Fahrlässigkeit begangen worden ist, so tritt Freiheitsstrafe bis zu einem Jahr oder Geldstrafe ein.
(2) Straflosigkeit tritt ein, wenn der Täter die falsche Angabe rechtzeitig berichtigt. Die Vorschriften des § 158 Abs. 2 und 3 gelten entsprechend.

Para. 161 StGB (German Criminal Code): False Statutory Declarations Due to Negligence
(1) If a person commits one of the offences listed in sections 154 through 156 negligently the penalty shall be imprisonment not exceeding one year or a fine.
(2) The offender shall be exempt from liability if he or she corrects their false testimony in time. The provisions of section 158 (2) and (3) shall apply accordingly.

Die vorstehende Belehrung habe ich zur Kenntnis genommen:
I have read and understood the above official notification:

Aachen, 08.03.23 — Ort, Datum/City, Date

Unterschrift/Signature

Acknowledgement

I would like to express my deepest thanks to Tarik Viehmann for his advices and constructive feedback throughout the whole thesis project. This project would not have been possible without his contributions to the refbox changes for receiving partial points from the refbox. I also appreciate Tarik Viehman and Till Hoffmanns sharing their insight knowledge about the execution framework Fawkes which was very helpful. Furthermore, I want to thank Prof. Lakemeyer for supervising this thesis. Additionally, I would like to thank the Carologistics Team for the interesting conversations about the goal tree and goal structure of the central agent. The involvement in the team really helped to develop an understanding of the robotic setup. Furthermore, I would like to express my deepest thanks to my family and friends who mentally supported me throughout this time. Finally, I am grateful to my dad, Patrick Halbach, Ronja Keuper, Jasmin Tejkl for their support and encouragement.

Contents

Acronyms

A3C	Asynchronous Advantage Actor-Critic
AI	Artificial Intelligence
AOP	Aspect-Oriented Programming
BE	Behavior Engine
BS	Base Station
CLIPS	C-Language Integrated Production System
CS	Cap Station
CX	CLIPS Executive
DS	Delivery Station
GIL	Global Interpreter Lock
GR	Goal Reasoning
HTN	Hierarchical Task Network
MDP	Markov decision process
MPS	Modular Production System
NN	Neural-Network
PDDL	Planning Domain Definition Language
PPO	Proximal Policy Optimization
RCLL	RoboCup Logistics League
RL	Reinforcement Learning
RS	Ring Station
SS	Storage Station

1 Introduction

While humans sometimes struggle to choose the „right“ decision from a given set of various options, it is even harder to design algorithms for robots to do so. Nowadays, robots are a common part of our day-to-day live. 19 million service robots,for example vacuum cleaning or lawn mowing robots, were sold in 2021 [1]. An even larger and more sophisticated application can be found in the industry which utilizes autonomous robots in smart factories of production lines. In 2021, 517,385 industrial robots were installed in factories around the world [2]. These production robots determine and pursue short and long term production goals based on available resources. To achieve their goals these robots reason about their next set of actions. They handle complex tasks, which requires high-level decision making processes. Goal Reasoning (GR) is used for this purpose, which means that an agent reasons about its goals and decides what to pursue next. One particular step in this process is the goal selection step, where the agent decides which of the possible goals is most promising. This step is important on a long term strategy perspective. So far, mainly hand-crafted selection criteria are utilized. There exist multiple ideas and techniques to improve the reasoning through learning. Reinforcement Learning (RL) is a technique where an agent learns to decide which action to pursue next based on a given reward of its environment. Since both techniques are about making good decisions, we want to use the principle of rewarding a good decision. In GR it would mean rewarding a good goal selection. Therefore we want to use Reinforcement Learning (RL) for the goal selection to replace the hand-crafted selection criteria. We therefore propose a goal selection component which is utilizing RL.

The objective of this work is to improve and extend goal reasoning through a goal selection with RL. The idea of the proposed approach is to choose the next goal based on a list of goals and a current world state. The RoboCup Logistics League (RCLL) models a smart factory production, where two teams of autonomous robots compete in the fulfillment of receive dynamically generated orders. We choose the RCLL environment as the application field of this approach because it is a strategic game with multiple instant decisions to make.

The fundamentals of goal reasoning are explained in Section 2.1. Furthermore, RL and the classification of RL algorithms are described in Section 2.2 and Section 2.3. The application area is introduced in Section 2.5. In Section 3 the current state of science is outlined and set into relation to this novel approach. A conceptual model of the goal reasoning improvement is proposed in Section 4. The results are presented in Section 5 and are discussed in Section 6. Finally, Section 7 summarizes the objectives of our project. The source code of the goal selector is available on GitHub.[1,2]

[1] `https://github.com/fawkesrobotics/fawkes/tree/sginter/reinforcement-learning-on-goals`
[2] `https://github.com/carologistics/fawkes-robotino/tree/sginter/`

2 Background

This section describes the foundational knowledge for this work. First, goal reasoning is introduced followed by basic definitions of reinforcement learning. Third, we conclude with an introduction of the RCLL.

2.1 Goal Reasoning

Often nature and especially the human is taken as a role model. In particular, the characteristic of independent action and goal development is a motivation for the area of Goal Reasoning (GR) [3]. GR agents are one specific branch of autonomous agents. These GR agents are programmed to deliberate their next tasks and self-select their objectives. It can be defined in the following way:

> *„ Goal reasoning [is] the process by which intelligent agents continually reason about the goals they are pursuing, which may lead to [a] goal change. “* Aha [3]

GR agents are intended for use in complex environments [3, 4], where agents, subjects, and objects influence and modify the environment.

Therefore a goal represents a state of the world an agent would like to achieve or maintain. The robot can achieve a goal through completing tasks. Through the goal refinement process constraints are recursively added to the goal until the tasks are clear and a solution is found [5]. Complex goals are easier described by splitting them up in smaller *sub-goals*. This creates goal trees, introduced in Section 2.1.2.

One form of goal refinement is a goal lifecycle, introduced by Roberts, which captures the possible decision points of a GR actor and complements a plan's lifecycle [5], see Section 2.1.2.

To define these constraints, it is possible to use the rule-based production system CLIPS [6]. In the next Section 2.1.1 the specific goal reasoning system CLIPS Executive (CX) is introduced. It is based on the goal refinement approach from Niemueller et al. [7].

`goal-selection-via-rl/`

2.1.1 CLIPS Executive

The CLIPS Executive (CX) is an integrated goal reasoning and executive system to perform high-level decision-making processes. It provides a framework that allows a developer to express goals, ability of changing goals, and to formulate goal trees easily [7]. The system invokes and monitors planning systems and the execution models are built on top of the planning models. The CLIPS Executive (CX) consists of a fact base and a knowledge base. The knowledge base contains the rules and functions for reasoning about the current state of the world. The fact base includes the domain model, the goals, world model facts and other system facts.

CX uses a domain model encoded in Planning Domain Definition Language (PDDL) [8]. This allows utilizing planners to find action sequences. PDDL provides a planner independent interface for a better interchangeability [9]. PDDL is also used as execution model (validating preconditions, applying effects according to the model), hence domain and world model are continuously synchronized.

PDDL The PDDL is used to encode planning problems and provides an planner independent interface, this allows a better interchangeability [8, 9].

A planning problem is specified through a domain and a problem description [9]. The domain spezification contains the behavioral capabilities of the system in form of predicates and action descriptions. Predicates are properties of the objects in the domain. An action describes a state transition of the world, therefore it has preconditions and effects.

The definition of the put-down action in the well-known planning domain BlocksWorld.

Listing 2.1: Example PDDL-action for putting down a block.

```
(:action put-down
 :parameters(?x - block)
 :precondition(and (holding ?x))
 :effect(and
          (not (holding ?x))
          (clear ?x)
          (handempty)
          (ontable ?x))
)
```

2.1.2 Goal Lifecycle

In general the goal lifecycle describes the progression of goals. Figure 2.1 shows the CX specific goal lifecycle. The following paragraph is based on [7] and describes the steps of an agent using the CX framework.

Formulated: It is the initial state of any new created goal.

Selected: The goal reasoner is choosing one or more goals based on a set of criteria (e.g. resources available, priorities, cost metrics, etc.)

Expanded: The selection of a goal is triggering its expansion. In this state the goal is divided into sub-goals or a course of action is determined, e.g. a planner is called. Furthermore, it is possible to use predefined plans for a specific goal class or to invoke multiple different planners for one single goal to get multiple plans for one goal.

Committed: The CX picks one sub-goal or one plan and acquires the necessary resources. A goal can still be rejected if the evaluation of the plan's cost results in another goal being more desired.

Dispatched: Now the agent actually executes the actions of the plan or sub-goal.

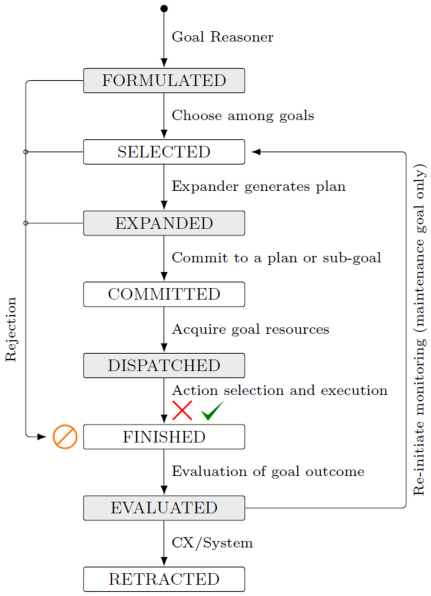


Figure 2.1: Goal life cycle from [7]

Finished: Eventually, the goal is finished and the outcome is recorded to indicate that the goal has succeeded or failed to reach the intended effects.

Evaluated: Based on the goal outcome, changes to the world model are applied.

Retracted: The agent does not need the goal anymore so the CX removes it and its plans, the resources and all associated data are released.

Goal Trees As mentioned in the goal expansion step, a goal can be split into sub-goals. Through this, it is possible to build up a goal tree recursively, introduced in [7]. A goal that is divided into sub-goals is called a *compound goal*. A *compound goal* without a parent is a *root goal*. Depending on the treatment of the sub-goals, it is possible to define multiple *compound goal* types, e.g. run-all, try-all, run-one, retry, timeout [7].

Goal Selection In the context of expert-based agent learning, goal selection plays an important role. In the work of Powell [10] three different types are presented and discussed. (a) Goal selection queries, the agent requests an expert for the goal to select in the current state. (b)Generalization confirmation queries here the GDA agent proposes a goal selection based on its knowledge and requests feedback from the expert about its hypothetical selection.

(c) Goal selection criticism. In contrast to the other two types this interaction is expert-initiated. The expert observes the agent and gives feedback if a goal selection in a given state was correct.

The goal selection process is one of the main parts of a goal reasoning system. If the agent would be given a complete function $F : S \rightarrow G$ which determines which goal $g \in G$ should be pursued for all possible situations $s \in S$, no goal reasoning would be necessary [3]. In many cases it is not feasible to create a complete function. In such cases the selection can be done based on different criteria like available resources, cost metrics or goal priorities [8]. In this work this selection function is learned through reinforcement learning.

2.2 Reinforcement Learning

A Reinforcement Learning (RL) problem consists of an agent and an environment, see Figure 2.2. RL is a sub-discipline of machine learning which deals with sequential decision making processes, where the goal is to select actions for maximizing the total future reward [11]. Sutton and Barto [12] give the following definition for RL:

> *„ Reinforcement learning is learning what to do — how to map situations to actions — so as to maximize a numerical reward signal. The learner is not told which actions to take, but instead must discover which actions yield the most reward by trying them. “* [12]

This means the RLagent receives rewards for its actions and based on this, independently develops a strategy to maximize the rewards. In contrast to unsupervised learning, RL problems have no hidden similarities of a group of data entries which can be discovered through clustering. In opposition to supervised learning there is no labeled data set for training and applying the prediction. Instead, a RL problem has an underlying Markov decision process (MDP) and the RL agents can consist of a policy, value function and/or a model. The MDP is either known or unknown. The mapping function from states to actions, which maximizes the reward, is called policy. Thus, the RL agents need to learn the optimal policy.

This section discusses the components of an RL agent and MDP based on [11] and [12]. The formal definition of [11] are used in Section 2.2.1.

2.2.1 Markov Decision Process (MDP)

Markov decision process (MDP) descibes problems where an agent tries to maximize the future *reward*. The agent chooses the next action from a set of available *actions*. Thereupon the environment *state* changes, so into a *state transition*. Which state the agent achieves is not deterministic, but the probabilities depend only on the chosen action and the current state. The reward which the agent receives is determined by the previous and current environment state, as

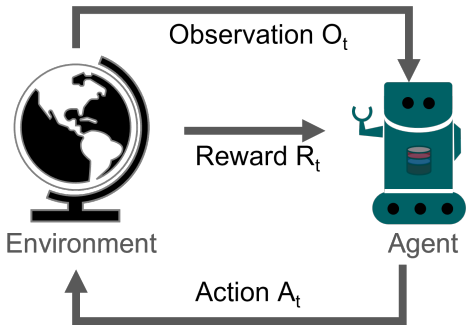


Figure 2.2: Standard RL framework, adapted from [12]

well as the selected action. All possible rewards are described in a *reward function*. The agent starts in an initial state and ends in one of multiple final states. After a final state no more action can be executed. To get to a final state several consecutive actions are necessary, accordingly the agent collects several rewards until a final state is reached. Whether the rewards of the distant future counts as much as the present reward, is determined by the *discount factor*.

Finding a strategy (policy) which assigns to each state the action of the highest profit solves such problems. The formal definition is:

Definition 1. ***Markov Decision Process:*** *[11]*
A Markov Decision Process is a tuple $\langle \mathcal{S}, \mathcal{A}, \mathcal{P}, \mathcal{R}, \gamma \rangle$*, where:*

- $\mathcal{S}$ *is a (finite) set of states and*
- $\mathcal{A}$ *is a finite set of actions*
- $\mathcal{P}$ *is a state transition probability matrix,*
  $\mathcal{P}_{ss'}[S_{t+1} = s' | S_t = s]$.
- *And* $\mathcal{R}$ *is a reward function,* $\mathcal{R}_s = \mathbb{E}[R_{t+1} | S_t = s]$ *and*
  $R : S \times S \times A \to \mathbb{R}$
  $R(s, a, s') = \mathbb{E}(r_t | s_t = s, a_t = a, s_{t+1} = s')$
- $\gamma \in [0, 1]$ *is a discount factor.*
  If $\gamma = 0$*, the agent is involved with maximizing the immediate reward. The long-term reward is ignored. Contrarily, if* γ *is close to* 1*, the distant future reward counts equal to the current reward.*

Figure 2.2 illustrates the process where an agent selects an action, the environment changes, the new state of the environment is observed by the agent and the agent receives feedback on his last action in the form of a reward. The interaction can be represented by a loop. Each time step t the agent performs an action A_t and in response the environment returns an observation O_t with a reward R_t. The Markov property holds, which states that the future state of the process depends only on its current state and not on the history prior.

This state and the reward is the information used by reinforcement learning algorithms [11].

Reward The scalar feedback provided by the environment at a certain time step t is called reward R_t. It indicates the performance of the agent. The aim of the agent is to maximize the cumulative reward, which is defined as the reward hypothesis. It also means that the agent might need to give up an intermediate reward to gain even more rewards in the future.

Agent The agent represents the learning algorithm, which learns to perform actions in that environment. The *model* is the agent's representation of the environment (based on observations). It is used to predict the next reward and the change of the environment:

next state: $\mathcal{P}^a_{ss'} = \mathbb{P}[S_{t+1} = s' | S_t = s, A_t = a]$

next reward: $\mathcal{R}^a_s = \mathbb{E}[R_{t+1} | S_t = s, A_t = a]$.

Policy The policy is the agent's behavior function. Given a set of observations, the policy decides which action to take. $\pi : S \to A$ where S is the set of states and A is the set of actions. There are different policies, e.g. the most simplest is a deterministic one: $a = \pi(s)$, another one is the stochastic policy: $\pi(a|s) = Pr[A = a|S = s]$, which defines a probability distribution over the actions, given a state.

Return RL would be obsolete, if there would exist a feasible way to design the perfect policy (correctly command the right actions for every observed state) for a problem. The RL agents' objective is to choose the optimal policy, the one which maximizes the sum of rewards. The return G_t is the total reward from time-step t.

Value function (V-function or state-value function) In order for the agent to make a decision about which action to choose next, it must evaluate the current environment state. The value function is an evaluation scheme of states, according to which the agent can act. It determines the total future reward the agent can expect to receive based on the current state and suppose it performs the optimal action and from there on it acts optimally. Many popular reinforcement learning algorithms are based on learning the value function.

Definition 2. ***V-function:***

$$v_\pi(s) = \mathbb{E}_\pi[G_t | S_t = s] = \mathbb{E}_\pi[R_t + \gamma R_{t+1} + \gamma^2 R_{t+2} ... | S_t = s] = \mathbb{E}_\pi\Big[\sum_{k=0}^{\infty} \gamma^k R_{t+k+1} | S_t = s\Big]$$

Action-value function (Q-function) It defines the value of taking action a in state s under a policy π, denoted by $Q_\pi(s, a)$.

Definition 3. ***Q-function:***

$$Q_\pi(s,a) = \mathbb{E}_\pi[G_t|S_t = s, A_t = a] = \mathbb{E}_\pi\Big[\sum_{k=0}^{\infty} \gamma^k R_{t+k+1}|S_t = s, A_t = a\Big]$$

The probability that a policy π selects an action a at state s is denoted as $\pi(a|s)$, with $\sum_{a\in A} \pi(a|s) = 1$. The state-value function is equivalent to the sum of the action-value functions of all outgoing (from s) actions a, multiplied by the policy probability of selecting each action:

$$v_\pi(s) = \sum_{a\in A} \pi(a|s) * Q_\pi(s,a)$$

.

Exploration vs Exploitation The learning process of an RL agent can be compared to a child, it tries something out until it works. Each time it fails it starts again and might try another way. It is called *exploration* if the agent discovers new ways and collects more information about the environment. Exploration means the agent might lose reward while discovering but gets the change to discover a new path with higher rewards. If a discovered way with high reward is chosen over and over it's called *exploitation*. This is suitable to maximize the reward. The difficulty is to find the right balance between exploration and exploitation.

Depending on the components of the learning algorithm the classification of the RL algorithms is made.

2.3 Classification of RL Algorithms

A RL agent represents the learning algorithm used to interact and learn from the environment. It can consist of a policy, value function and/or a model. The categorization of RL agents is based on these components. A value-based RL agent has only a value function and no explicit policy function. Typically, value function based methods are either Monte Carlo or Temporal Difference Methods. Analogously, a policy-based agent directly searches in the space of the policy parameters to find an optimal policy, it has no value function. An agent trying to get the best out of both, a policy and a value function, uses an actor-critic RL algorithm. Correspondingly a model-free RL algorithm uses no model of the environment while a model-based RL algorithm contains a model. Model-free approaches can also be classified as being either *on-policy* or *off-policy*.

On-Policy vs Off-Policy **On-policy** methods use the current policy to generate actions and use it to update the current policy. This method has the following advantages: Better convergence properties, effective in high-dimensional or continuous action spaces [12]. But it typically converges to a local rather than a global optimum and the evaluation of the policy is typically inefficient and with high variance [12]. Contrarily, **off-policy** algorithms use a different exploratory policy to generate actions as compared to the policy which is being updated. Figure 2.3 gives an overview of some RL algorithms, it's the RL taxonomy defined by OpenAI.

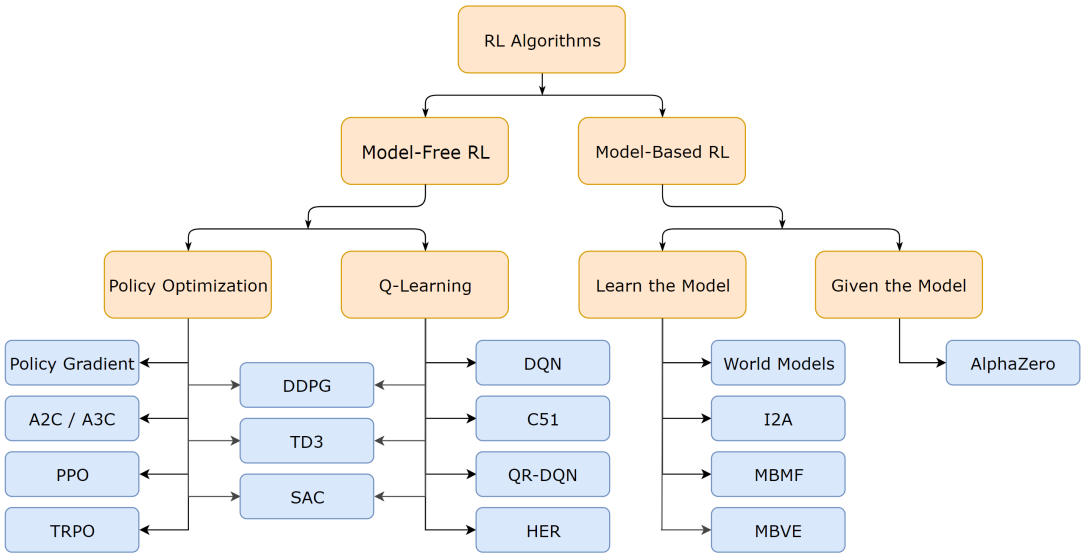


Figure 2.3: RL taxonomy as defined by OpenAI [13]

2.3.1 Monte Carlo Methods

Monte Carlo Methods update the value function based on the overall outcome of an episode. So the value function for each visited state of the episode is only updated once at the end. Monte Carlo Methods require many iterations for their convergence and suffer from a large variance in their value function estimation.

2.3.2 Temporal Difference Methods

Temporal difference (TD) learning starts with randomly chosen V-values and iteratively improves them. Therefore the temporal error, which is the difference of the new estimate of the value function and the old estimate, is calculated. By considering the reward received at the current time step the value function is updated. This kind of an update reduces the variance but increases the bias in the estimate of the value function.

2.3.3 Actor-Critic Methods

Actor-critic methods use both a policy and a value function. The policy $\pi_\theta(a|s)$ is called **Actor**, as it provides the action in a given state. The value function is named as critic, it evaluates the policy and helps the actor in learning. The value functions are either the state value $V(s)$, state-action value $Q(s,a)$, or advantage value $A(s,a)$ functions. These methods can be on-policy or off-policy [14], the former is more common. The actor aims to improve the current policy, while the critic evaluates the current policy [15].

There are many ways to implement an actor-critic architecture [15].In case of a small action-space, the critic may, e.g., use an approximate action-value function and the actor could follow a greedy exploration strategy.If the action-space is large or continuous, the actor itself may use function-approximation.

Policy Gradient The policy gradient algorithm aims to find a local maximum in a policy objective function $J(\theta)$ by ascending the gradient of the policy, w.r.t. parameters θ, $\delta\theta = \alpha\nabla_\theta J(\theta)$, where α is a step-size parameter and

$$\nabla_\theta J(\theta) = \begin{pmatrix} \frac{\partial J(\theta)}{\partial \theta_1} \\ \vdots \\ \frac{\partial J(\theta)}{\partial \theta_n} \end{pmatrix}.$$

So the methods mainly differentiate through the used policy gradient. The methods can be further specialized through setting parameters and hyperparameters to adapt them for the use case. The important parameters for this thesis are presented in the following Section 2.4.

2.4 Hyperparamter

Hyperparameters are parameters that are set prior to training a learning model and are not learned from the training data. In RL, hyperparameters can have a significant impact on the performance of the trained RL agent. The most common hyperparameters are the learning rate α and the discount factor γ. Relevant parameters of the Proximal Policy Optimization (PPO) and the MaskablePPO algorithm are discussed. As MaskablePPO is an extension of PPO, the parameters are very similar. The key differences between the parameters include the masking with the masking function. The section is based on the documentation of the StableBaseline3 implementation [16][**?**].

learning rate α It determines the step size at which the RL agent updates its policy based on the reward received after taking an action. A larger learning rate can lead to faster learning but may also make the agent more unstable, while a smaller learning rate may lead to slower learning but more stable convergence.

discount factor γ It affects the importance of future rewards versus immediate rewards. A discount factor of 1 means that the agent values all future rewards equally, while a discount factor of 0 means that the agent only values immediate rewards. Choosing an appropriate discount factor can help the agent balance long-term and short-term goals. In the RCLL application scenario the long-term rewards are more important, therefore a discount factor close to 1 is chosen.

policy This hyperparameter determines the type of model that will be used to represent the policy, which can impact the learning performance of the algorithm. The MlpPolicy is used which is an alias of MaskableActorCriticPolicy.

n_steps It is the number of steps that the algorithm makes in each environment before updating the policy. E.g. $n\_steps = 3$ means that after three times calling the step function, the policy is updated.

iteration Count of policy updates.

total_timesteps $total\_timesteps = n\_steps * iterations$ The total count of calling the step function.

n_episode An episode includes all the steps till a final state is reached. Thus in our case the number of episodes corresponds to the number of played RCLL games.

batch_size The minimal batch size determines the number of samples used to update the policy at each iteration. Its calculated as as $n\_steps * n\_env$, where n_env is the number of environment copies running in parallel. In our case the batch size equals to n_steps, as we have only one environment running. A larger batch size can lead to faster learning but may also make the algorithm more sensitive to noise in the data.

In summary, the learning rate, discount factor and number of episodes are important hyperparameters for RL that can have a significant impact on the performance of the trained RL agent. They can be tuned through hyperparameter optimization to achieve the best possible performance.

Tuning these hyperparameters can be done through a process called hyperparameter optimization, which involves training the RL agent multiple times with different hyperparameter settings and selecting the settings that lead to the best performance.

2.5 RoboCup Logistics League

The RoboCup Logistics League (RCLL) [17] is a competition of the international RoboCup [18] where the aim is to produce products with robots simultaneously, to simulate a smart factory scenario.

In the competition, two teams are playing against each other on one 14m × 8m field. The working stations of each team are randomly placed over the field. Each team has three autonomous robots, the agents are handling the production of dynamically posted orders. An order defines the complexity of the product, the delivery time window and the required quantity. The teams score for single production steps and the delivery of the finished product [19, p.3].

There are four different product types regarding their complexity. The simplest product is a C0, consisting of a base and a cap. All other products have one to three rings between the base element and the cap (C1-C3). Figure shows an example product in a schematic manner. Depending on the ring color, the ring stations need some additional bases or capcarrier as payment for the ring. Each team has five different types of Modular Production System (MPS).

- The Base Station (BS) dispenses bases.
- The Cap Station (CS), has a shelf with caps mounted on cap carriers. To mount a cap, the station must first receive the cap, which is buffered in the machine and assembled on the next product placed on the input conveyer belt. While buffering the cap, the cap carrier is dispensed at the output side of the MPS.
- There are two Ring Station (RS). They mount a ring on the product on the input conveyer belt if the payments have been paid to the slide of the RS.
- A Delivery Station (DS) consumes the finished products.
- Samples of each possible C0 configuration are provided by the Storage Station (SS). [19, p.6ff]

The possible configurations arise from the different colors of the workpieces.

Workpiece	Colors
Base	red, black, silver
Ring	green, blue, yellow, orange
Cap	black, grey

Figure 2.4: Colors of the workpieces.

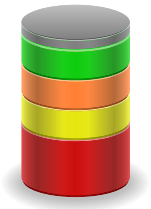


Figure 2.5: Example C_3 [19, p.18]

The frequently winners of the RCLL game is the Carologistics Team from the RWTH Aachen and FH Aachen.

To participate in the RCLL, the Carologistics Team uses a GR agent to process and complete incoming orders as goals, for more informations about the setup see [20]. The goal selection of this agent is based on priorities. The aim of the thesis is to explore a goal selection with RL.

Fawkes The software framework Fawkes [21] is a middleware that connects multiple software components, which are realized as Fawkes plugins.

Fawkes provides multi-threading for these components. To increase the modularity Fawkes uses the Aspect-Oriented Programming (AOP) paradigm, therefore the interfaces are realized through *aspects*. A plugin can access features from Fawkes or other plugins by inheriting from the specific aspect, which is a class implementing a specific ability. Fawkes core provides a blackboard that is based on a reader-writer model allows to pass data from the writer to the reader.

The Fawkes main thread is responsible for loading and unloading plugins. A Fawkes plugin is implemented as dynamically loadable library (shared objects on Linux system), with a particular interface. A plugin has a mode, this mode can be either continuous or wait-for-wakeup. Furthermore, a plugin has a *loop()* function. In the continuous mode, the thread runs until it exits or is terminated by another thread. Thus, all the time the *loop()* function is called. While in the wait-for-wakeup mode, the thread blocks, until a wake up call. When woken up, the *loop()* function it executes for a single iteration. [21] The wake-up signal can either come from another function (e.g. through a specific blackboard message) or from the main loop of the main thread. The main loop iterates over different stages, in each stage all plugins registered for this stage are woken up and executed concurrently [21].

The central agent of the Carologistics Team is integrated into Fawkes and therefore the framework is used in this approach as well.

Lua-based Behavior Engine and Skiller The Behavior Engine (BE) in the middle layer offers a set of tools and a framework for supervision of reactive behaviors. This system acts as a mediator between the high-level reasoning process and low-level execution mechanisms by occupying an intermediate layer. It is responsible for executing and monitoring behaviors and reporting status information to the higher level. The BE is implemented in Lua scripting language, which is lightweight expressive and well-suited for implementing the Skill-HSMs [21].

RCLL Referee Box The autonomous referee box (refbox) is designed to automate the playing field and minimize the workload of referees. Acting as an agent, the refbox plays a crucial role in enabling the smart factory aspect of the RoboCup Logistics League (RCLL) scenario. It generates randomized game layouts and schedules. Furthermore, the refbox determines appropriate field reactions by processing incoming MPS sensor data and robot communication through a knowledge-based system. [22] [23] The refbox is responsible for supervising the game and enforcing the rule of the rulebook. [19]

3 Related Work

Artificial Intelligence (AI) planning and reinforcement learning (RL) are different techniques which both address the problem of sequential decision-making. In the past decade there have been various approaches investigated to combine them.

3.1 RL: Image based observation - Action selection

Dittadi et al. [24] learns a neural network with raw pixel data to represent the domain dynamics in a way that can be used for planning.

In 2013 Mnih et al. [25] published the well-known DQN RL algorithm, which is applied to seven different Atari 2600 games using the pixel images of the Atari environment as input for the algorithm. The goal is to play the games by selecting actions in a way that maximizes the future reward [25]. Using images as input has the consequence that the relevant information about the current state of the world must first be extracted. In contrast to that, this thesis stems on an existing high-level abstract representation of the world in form of facts in the knowledge-base. Therefore, an abstrac representation of the world can be used for the process of learning.

3.2 Hierarchical approaches

Although both approaches Mukadam et al. [26] and the thesis use RL for tactical decision making, they differ in their mapping of the high level decision onto the RL action. In [26], the high level decisions are translated to five actions that can be performed by he RL agent at any time step. Q-masking is applied to eliminate impossible actions.

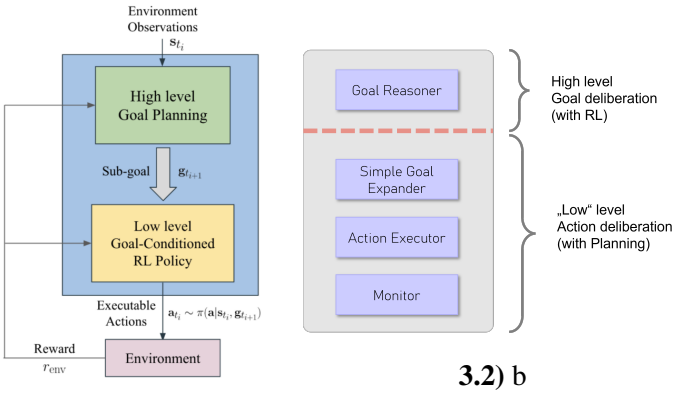
The decision for which action to perform next also belongs to the area of Hierarchical Task Network (HTN). The idea of HTNs is to combine a sequence of primitive tasks into a larger compound task.

If a long-term goal is too complex to be tackled at once, it can be divided into sub-goals. In order to fulfill this long-term goal, the belonging sub-goals and their tasks have to completed which requires planning [27].

Thus, HTN planning is a method to generate a plan automatically. A plan is a sequence of actions to achieve a goal task. The goal trees are built accordingly. Recent research has proposed an

architecture design with a high-level goal planning component and low level goal-conditioned RL Policy component in [28] or a high-level graph search-based planner [29].

Figure 3.3 compares the proposed architecture of the RL based goal selector with the architecture of the hierarchical goal-conditioned offline reinforcement learning framework of [28].



3.1) a **3.2)** b

Figure 3.3: The hierarchical goal-conditioned offline reinforcement learning framework [28] (a) vs. goal selection via RL Framework (b)

Besides for high level goal planning components, GR is also used in the literature for high level decisions. The goal driven autonomy-cooperation (GDA-C) approach of Hutchison et al. [30] uses GR to select a goal and then, depending on the goal class, calls an RL agent which determines the plan. Thus each RL agent is only responsible for choosing the right action from a subset of the whole action space. In contrast to this thesis, RL is not used for the goal selection, but rather for the action selection.

Task and motion planning are related problems which require simultaneously solving a high-level, discrete task planning problem, and a low-level, continuous motion planning problem. Nair and Finn [31] solves the related task and motion planning problem by utilizing hierarchical reinforcement learning. Table 3.1 summarizes and highlights the architectural differences of the introduced related work.

	Li et al. [28]	Hutchison et al. [30]	Nair and Finn [31]	This thesis
Goal level	High level goal planning	GR	hierarchical RL	RL
Action level	Low level goal-conditioned RL policy	Using RL to determine a plan	hierarchical RL	predefined fixed plan

Table 3.1: Architectural comparison between the thesis and related work

3.3 Goal Selection

Wilson and Aha [32] introduces a GR model which is selecting a sequence of goals at once. Therefore the goal sequence selection is treated as an instance of the Orienteering Problem (OP) [33], whereas we want to select a single goal to pursue next.

Corresponding to the subgoal selection, Bonanno et al. [34] combines subgoal selection with deep learning, by using a Convolutional Neural Network (CNN) to classify the game images into specific actions. Thus, the CNN [35] learns to select subgoals from images. It is trained by a hard-coded expert procedure in a supervised fashion and the set of eligible subgoals is always the same, regardless of the state of the game.

3.4 RL for generating plans

Perhaps the most similar prior work to this thesis is Núñez-Molina et al. [36], which integrates Deep Q-Learning along with planning. In a similar manner to this thesis, [36] demonstrates the ability to learn the subgoal selection. The method is designed to decrease the load of a planner. Therefore the goal selection module selects the subgoal with the minimum length of its respective plan. [36] takes a different approach in the world representation. They chose the General Video Game AI (GVGAI) Framework where game levels are characterized through simple text files. Each game level has a corresponding planning problem, which encodes the initial state and the goal to achieve in PDDL. The CNN gets the game state and subgoal encoded into a one-hot matrix. Each cell of the game field is associated with a one-hot vector encoding the objects in this cell. The objective of their paper was to minimize the length of the entire plan, thus the CNN predicts the length of a plan given the state s and subgoal g. Table 3.2 shows the differences between [36] and this thesis.

RL	Núñez-Molina et al. [36]	Proposed approach of this thesis
Action a	Subgoal g	Subgoal g
Reward r	Plan length $l_{p(s,g)}$	Reward r
Cumulative Reward R	Length of the entire plan $l_{P(s,g)}$	Cumulative Reward R
Maximize R	Minimize $l_{P(s,g)}$	Maximize R

Table 3.2: Correspondence between RL, the work of Núñez-Molina et al. [36] and this thesis

The work of [37] selects goals based on priorities and utilities RL to learn the priorities of a goal based on its goal-priority profile mapping. This differs from the approach taken in this thesis as the latter does not use priorities but learns the goal selection directly.

Lee et al. [38] also aim to bring RL and Planning closer together. Their idea is to create a mapping between the planning task and the reinforcement learning task. With the mapping, RL algorithms can be applied to an option of a planning task. Similar to this approach, they are using the RL framework StableBaseline3 [39]. They apply the PPO [40] algorithm and

PDDLGym [41] for their experiments. However, they implement PDDLGym to translate a planning task into a MDP and introduce a „SMDP Learning with PPO“ algorithm to apply it on the planning problem. For the evaluation of the RL agents they utilize the average rewards and average lengths of the plans.

4 Approach

The RL based goal selector is extending a robotic execution framework which uses goal reasoning for high level decision making. The CX is a goal reasoning system, introduced in Section 2.1.1. A PDDL based domain model is part of the CX. The domain facts describe the current world state. The environment of the robot is constantly changing, on the one hand through the actions of the robot itself, on the other hand through factors that cannot be influenced.

A change in the world is perceived by the sensors and the world model facts are adjusted accordingly. As basis, the RCLL robotic setup of the Carologistics team is used. A core component within the goal reasoner is the goal selector, which is hand-crafted and priority based in the current version of the Carologistics.

A RL based goal selector is developed to tackle the weaknesses of a user-defined domain specific goal selection. Deciding which goal makes the most sense to pursue in which situation is a complex decision-making process.

Goal selection involves making decisions in a dynamic and uncertain environment. In order to select an appropriate goal, an agent must take various factors into account, such as the current state of the environment, the available actions, and the potential outcomes of those actions. This requires the agent to reason and plan, balancing short-term and long-term goals while always respecting the uncertainty and unpredictability of the environment.

Furthermore, the goals themselves may be complex and hierarchical, requiring the agent to reason about how to decompose a high-level goal into subgoals and how to prioritize those subgoals. This involves not only selecting goals but also sequencing them in a way that maximizes the chances of achieving the overall objective.

As RL is a method for developing complex long-term strategies, it is suitable for tackling the goal selection process. The key idea is to use RL to find an approximation for the goal selection function, which assigns a goal to each situation.

Compared to many other RL algorithms which use images as representation of the current world state, the novel goal selector combines the GR area by using the existing knowledge of the fact base to represent the world state.

The RCLL game is used as domain. The long-term selection strategy of the order to pursue next, is the critical factor for the competitiveness and overall performance in the RCLL game.

For the decision making process, the availability of resources and the related executability of a goal as well as the goal-duration and goal-effect must be considered. Furthermore, the distance between machines can vary depending on the game field. Thus, the production time of a product

differs. This leads to a complex decision making process in the RCLL. The novel goal selector is integrated into the CXand is responsible to predict the next goal based on the current situation.

In order to improve the goal selection capability, the focus is first set to connecting GR and RL. Concurrently, the RCLL is used as application area for the novel goal selector. The following RCLL-specific milestones will elaborate more on the 2-step approach taken in this thesis:

1. **Connecting GR and RL:**
    - System architecture
    - Goal selection process
    - Choice of RL framework
    - Introduction of key idea behind action and observation space
    - Basic mechanism of the RL agent based on the execution mode
2. **Implementation in the RCLL domain:**
    - ClipsGym: Extending Python
    - RLAgentManager and PyGuard: Embedding Python
    - In this context, the single modules of the goal selector components are further introduced.
    - Training process: Basic mechanism of training the RL agent
    - Integration into the goal tree
    - Action and observation space in the RCLL context
    - Action masking and Reward
    - Reset process

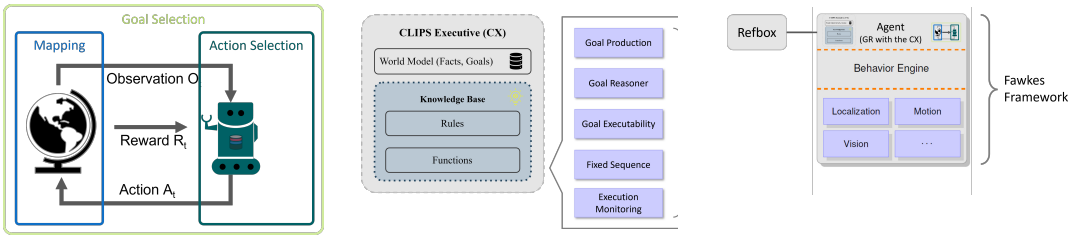
4.1 Connecting GR and RL

The connection between the real robot setup in the real world and the RL setup in a model world is explained below by different layers.

For linking gradually RL and GR, first the steps from the general perspective are layed out, before in Section 4.2 a showcase of the implementation contribution of the thesis is given.

The two sides are: on the one hand a RL framework (first picture) and on the other hand a robotic execution framework (last picture).

The core of our novel goal selection process is the RL function Figure 2.2. With the help of a world and the agent, an action is selected based on an observation or a world state s. Both, observation and action space are chosen as discrete spaces. The RL world is interacting with the



a) Center and Inner Layer **b**) Middle Layer **c**) Top Layer

Figure 4.4: Basic overview of the system based on different levels of considerations

mapping module, which is responsible for the mapping the GR domain facts and the discrete observation. Furthermore, it is in charge of the projection $M : A \rightarrow G$ of the action space to the goal space, introduced in Section 4.1.1. This is tied into the goal selector Figure 4.4, a). The prediction for next action is made by the RL agent. In conjunction with the mapping, a projection of the subsequent goal to be pursued is established. This prediction is the prerequisite for the goal selection via RL. Based on the goal class and the parameters, the corresponding concrete goal is determined and selected based on the id of the goal. Taking a higher-level perspective of the system, it becomes evident that the selection of goals is an essential aspect of the goal reasoner. In the goal reasoning mechanism, goal selection is a necessary step as goals must be selected before they can be executed, particularly when following the goal life-cycle. This is illustrated in Figure 4.4, b). The goal reasoner in turn is part of the entire high-level agent. Where the action executor is the connection to the underlying behavior engine. Figure 4.4, c) provides the top view of the whole robotic framework.

4.1.1 Goal selection

The goal selection in the goal life-cycle, introduced in Section 2.1.2, takes place in the goal reasoner after one or more goals have been formulated and before a goal changes its state to selected. Reasoning is needed, if there is no complete function that assigns a goal to every possible situation in a given domain. As soon as there are several formulated goals, all of which are executable, the agent must use reasoning to decide which goal to pursue next. One possibility is to prioritize the goals and carry out the selection on the basis of these priorities. Here we introduce another new way to make this decision about which goal to pursue next. The novel RL based goal selector uses the RL policy to make the decision. Since the RL agent works with actions and a model of the right world, a mapping must be done. The mapping includes on the one hand the conversion from an RL agent action to a goal and on the other hand the conversion from a real world observation to an observation based on the world model.

Definition 4. ***Goal Selection:***

- *G is a set of goals with goal $g \in G$ and*
- *S is a set of possible situations with situation $s \in S$ and*
- *A is a finite set of actions with action $a \in A$ and*
- *O is a (finite) set of observations (discrete environment state) with observation $o \in O$*

- *$F : S \to G$ this function describes the main task of the goal selector.*
- *$E : S \to O$, a function mapping to each possible situation in the real world to an observation of the RL world*
- *$RL : O \to A$, using the policy of the RL agent to predict an action based on a given observation*
- *$M : A \to G$, mapping a RL action to a goal*
- *Therefore, we can represent the goal selection function as: $F(s) = M(RL(E(s))) = g$.*

So the proposed goal selector is responsible to predict a goal given a possible situation. As goal reasoning system we use the CX, introduced in Section 2.1.1. A PDDL based domain model is part of the CX.Therefore, a possible situation s is described by domain predicates and domain objects. The representation of the situation s is converted into the observation RLagents of StableBaseline3 are able to handle either discrete or continuous environments. But in any case, they do not directly coop with facts, therefore a discretization of the possible situations S is necessary. This discretization can be illustrated as CX facts or in more detail domain predicates to discrete environment state function E. On this we apply the RL algorithm and get an action. Therefore an action to goal mapping is required. If we apply the mappings, we can describe the goal selection function using RL.

The two functions M and E represent the action and observation spaces of the RLagent and are discussed in more detail later in Section 4.1.3.

The $RL(s) = a$ function, which is basically `action = model.predict(s,environment)`, where model means the RL agent. Based on the given state of the environment it does the prediction. This means that we can already identify two parts, world and agent of this function.

4.1.2 Choice of RL framework

Following criteria are relevant for the RL framework of this master thesis:

- **Provides a good usability and good documentation (with examples)**
  So that people can easily familiarize themselves and adjustments be made easily.

- **Offers a wide range of algorithms**
  The goal selector should be able to be used in different domains. Depending on the domain, one or the other RL algorithm is more suitable. Deep Deterministic Policy Gradients (DDPG) [42], requires a continuous action space. A2C can handle a discrete action space. Mehta [43] denote that the training speed of DQN [25] is slower than of A2C and Asynchronous Advantage Actor-Critic (A3C) [44]. Therefore, it is important that the RL framework used has a large selection of models that can be easily exchanged. Due to its reliable performance in many RL tasks [40] PPO is initially used as RL algorithm. In the future it can be investigated what advantages it has to use the goal selector with another RL algorithm.
- **Uniform RL algorithm interface**
  As mentioned in the last point, it is conceivable that in the future another RL algorithm will be used for goal selection. Therefore, it is important that the algorithms can be easily exchanged to perform experiments with others which is possible if an uniform interface is provided.
- **Based on OpenAI Gym environment**
  As described in 4.2.3 this is the standard for RL environments. The implementation is done as an OpenAI Gym environment, it is necessary that the algorithms provided by the framework can handle it.
- **Deals with customization of environments**
  No existing OpenAI Gym environment can be used fo this usecase. A specific environment suitable for the goal selection is developed, therefore the framework should give a possibility to customize the agent for the appropriate environment. Thus it can run with the self-written environment.
- **Action masking has to be supported**
  Action masking, curiosity-driven exploration are methods for sparse valid actions or sparse reward. Depending on the domain, training could be better and faster when using such a method. Therefore, when selecting the RL framework, it is important to us that these models are also available

Based on these criteria StableBaseline3 [39] is chosen as open-source library. It provides a bundle of reliable state-of-the art RL algorithms [39] with a uniform interface. Further, it allows using and implementing a customized gym environment. The extension *common* also provides examples for action masking. Therefore, StableBaseline3 is a suitable framework for integration into Fawkes .
The system design aims to be as domain independent as possible by providing a *configurable* goal and observation space.

4.1.3 Defining the action and observation space

Action and observation space are a core component of the RL algorithm. In the following, the mapping of the RL spaces to the GR spaces and the generation of the two spaces will be discussed in more detail.

Observation space

The observation space contains all predicates which describe the environment space. Blocksworld is known as planning problem domain, where an algorithm has to create a plan how to stack and unstack different blocks to reach a certain combination of stacked blocks. For example a PDDLPlanner can be used to create the plan.

The Blocksworld domain without the actions is defined through Listing 4.1. It contains these object types and predicates:

```
1  (define (domain blocksworld)
2    (:types block robot)
3    (:predicates
4      (on ?x - block ?y - block)
5      (ontable ?x - block)
6      (clear ?x - block)
7      (handempty ?x - robot)
8      (handfull ?x - robot)
9      (holding ?x - block)
10     (pickup ?x - block)
11     (putdown ?x - block)
12     (stack ?x - block ?y - block)
13     (unstack ?x - block)
14   )
15 )
```

Listing 4.1: Blocksworld domain

For example, a state of the Blocksworld domain may look like this:
`[on(A,B), onTable(B), clear(A), handempty(robot1)]`, this means we have two blocks A and B and A is stacked on B, the gripper is empty and there is no other block on A so the top of A is clear. The hand of $robot1$ is empty. The predicates and objects are countable thus we can transform the fixed predicate set to a discrete observation space based on numbers.

For creating the observation space, first all possible predicate-object combinations are generated. Assigning a number to all combinations, which corresponds to the position in the vector e.g. here it is the row numbers, leads to the discrete representation. For the situation shown above, we get the following vector in Figure 4.5.

```
on(A,B)
on(B,A)
onTable(A)
onTable(B)
clear(A)
clear(B)
handempty(robot1)
handfull(robot1)
```

$$\begin{pmatrix}1\\0\\0\\1\\1\\0\\1\\0\end{pmatrix}$$

Figure 4.5: Mapping of domain predicates to discrete observation space

Action space

A discrete action space based on goal classes and other goal properties is created similar to the observation space.

In case of the Blocksworld domain, one goal is to stack the blocks in a certain way. There is an initial state given and the goal in the form of the target state. In a Blocksworld domain, the goal would be divided into several smaller goals. These are for example single *towers* that have to be built. The number of stacked blocks gives the complexity of the tower. Thus a *C1-tower* means one block is stacked on another, for example block A is on top of block B and B is laying on the table, then $[A, B]$ is a *C1-tower*.

```
(goal (id Tower1)
      (class TowerC1)
      (params A B ))
```

Listing 4.2: Example of a *C1-tower* goal

The first step is to generate the combination set of goal class with parameters (params), to create the mapping $M : A \rightarrow G$ of the RL action to a goal. The RL agents action space corresponds to the discrete projection of this set.

The set of goal class - params combinations for the goal classes *TowerC1* and *TowerC2* and the blocks A,B,C is:

```
[TowerC1#A#B, TowerC1#A#C, TowerC1#B#A,
TowerC1#B#C, TowerC1#C#A, TowerC1#C#B,
TowerC2#A#B#C, TowerC2#A#C#B, TowerC2#B#A#C,
TowerC2#B#C#A, TowerC2#C#B#A, TowerC2#C#A#B].
```

This combination leads to an action space of the size twelve. The current executable goals, depend on the availability of blocks. A block is *clear*, if no other block is stacked on top of it. If a block is not *clear*, it cannot be stacked on another one.

The action and observation space are defined. Thus, the mapping between the GR situation and the RL observation as well as between the RL action and the GR goal is discussed. Now it is relevant to embed the steps into the execution mode.

This thesis distinguishes between a trained RL model that is in execution mode and an untrained RL model that is in the training process. The trained RL agent in execution mode is used to select the next goal via the predict function. The individual steps of the goal selection process and its integration into the goal life-cycle are explained in the next Section 4.1.4. The training process will be discussed in Section 4.2.4.

4.1.4 Execution mode of the RL Agent

In Section 4.1.1 the procedure for executing a trained agent is described. Based on this, the execution mode of the RL agent is further deliberated. The activity flow in the training mode differs from this and will be described in more detail later in Section 4.2.4. Figure 4.6 starts with the goal reasoning step of formulating goals. Afterwards it is checked if one goal exists that is currently in the state selected, dispatched or committed. The goal can be executed by an acting entity like a robot. The domain checks all acting entities, whether they are actively engaged in any task or are in a waiting state for a new goal assignment. In this case the RL goal selector is called via the blackboard message interface. Then the RL goal selector asserts a fact with this *goal-id* as the next goal to pursue. This fact assertion fulfills the precondition for the *rl-goal-selection rule* in the goal reasoner and the rule is executed. Thus the goal state is changed to `SELECTED`.

Consequently, this leads to the fulfillment of preconditions for other rules in the goal reasoner. Once fulfilled, these rules are applied accordingly. After a new goal is assigned to an active entity, the executability of the other goals is discarded for this entity. The goal life cycle continues as normal.

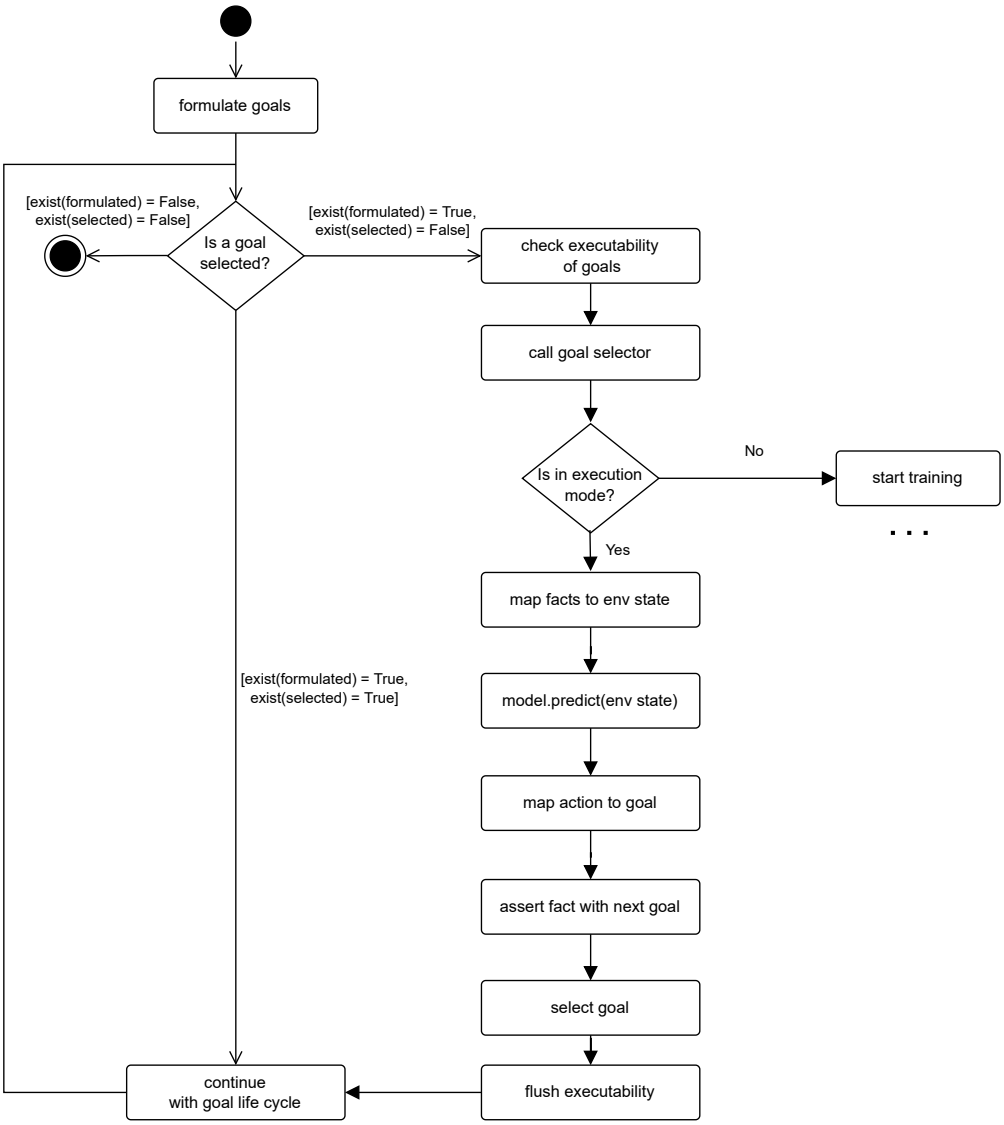


Figure 4.6: Execution flow of the goal selector

For enabling the agent to achieve a well-founded prediction it has to be trained initially. Since it is easier to understand the training process by means of the implementation, the concrete system design with the components and modules is explained and described first. Subsequently, the training process is explained in more detail in Section 4.2.4.

4.2 Implementation in the RCLL domain

The implementation of the goal selector starts with the real world robotic framework and evolves into the RL modules used and the RL framework. The RLbased goal selector is integrated into the multi-threading robotics software framework Fawkes. Figure 4.7 illustrates the relationships between the different modules and is used as a baseline reference throughout this section.

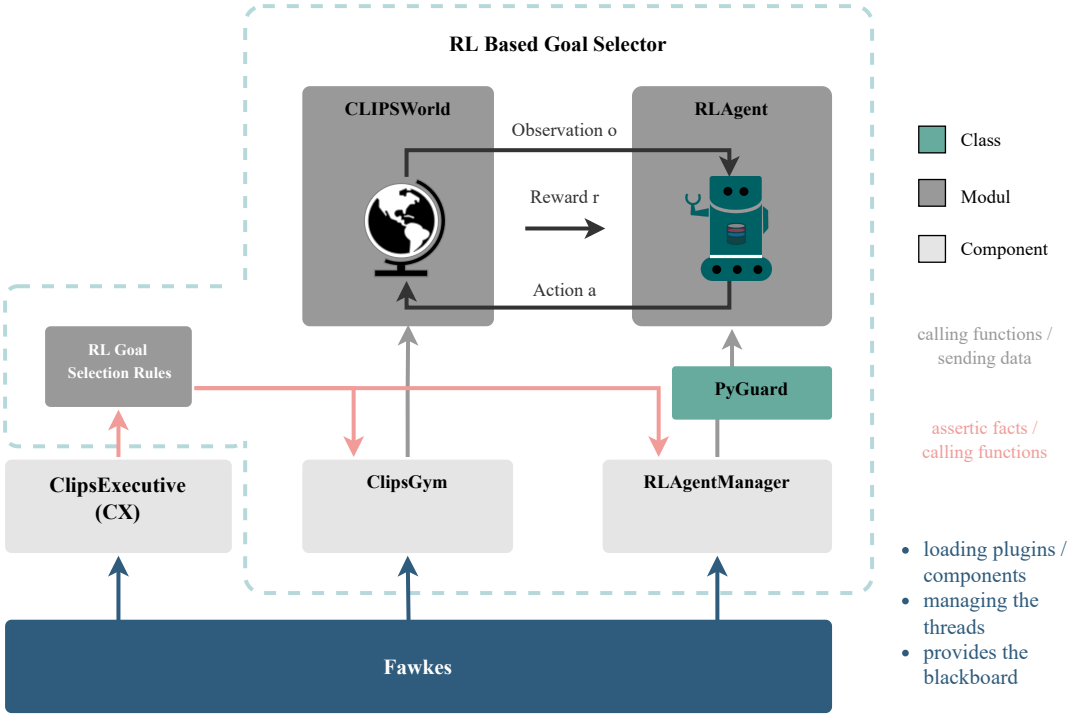


Figure 4.7: Overview of relevant Fawkes plugins and the novel goal selector

We have looked at the abstract building blocks/components of our system from the inside out, we now come to the concrete realization of the implementation. In the following, we will proceed from the outside in.

Top Layer The robots from the Carologistics team use, among other things, the robotic middleware Fawkes . The central agent is integrated into Fawkes and therefore the framework is used in this novel approach as well. Through the Fawkes blackboard message system the central agent communicates with the refbox. For example, machine instructions are sent to the refbox via the blackboard messages.

Middle Layer The central agent is implemented within the CX in C-Language Integrated Production System (CLIPS). Coming from the outer view of the system, the CX is realized as Fawkes Plugin. Through the rules and functions of the CX the goal reasoner is defined along

with e.g. fixed-sequences. The fixed-sequences determine a plan to reach a simple goal, based on the goal class.

Inner Layer On one hand, the proposed goal selector is integrated into the reasoner using various clip rules, which on the other hand call functions of the mapping or the action selection module Figure 4.7. ClipsGym has many more tasks than the mapping shown in Section 4.1. The module is described in more detail in Section 4.2.3.

Center A look closer into the RL goal selector shows that the two modules ClipsGym and RLAgentManager are realized as Fawkes Plugins in C++. So both are on the same system integration level as the CX. These are the interface to the RL modules which are implemented in python. While ClipsWorld (Section 4.2.3) is a single python class, the module on the left, described here as PPO Agent, is composed of several classes. And will be described in more detail in Section 4.2.3.

Before the individual modules are described. The implementationary constraines are dicussed.

Multi-threading C++ and Python integration constrains

Due to the Global Interpreter Lock (GIL), Python threads are unable to run concurrently on multiple CPU cores [45]. This prevents preemptive multi-threading, which occurs when one thread takes over by interrupting another thread. This is in contrast to the idea of the Fawkes main core to handle the threads and their processing times. Therefore, an integration into the Fawkes main core would cause severe limitations for the functionality of the overall system. A solution is to run the python interpreter in an asynchronous thread.

The Python `Py_FinalizeEX()` is prone for memory leaks, especially during flushing buffered data .[1] Those leaks can occur if the Python destructor fails to delete objects and therewith would not clean certain parts of the memory. Thus, creating a new python thread for each goal selection is not feasible. Instead, we can create a python thread and keep it's reference in a parental thread, for example the RLAgentManager, so that the reference remains valid even if the parent thread is interrupted by the fawkes core and is continued later.

The challenge within our environment is that the reference cannot simply be created or deleted in the constructor and destructor of the parent thread. As soon as the scope of the constructor is exited, the reference is invalid due to the scoped interpreter lock of our Python interpreter.

Therefore we have to use a workaround by storing the reference in the PyGuard so that the interpreter scope is only exited when the respective PyGuard object is deleted.

[1] https://docs.python.org/3/glossary.html#term-global-interpreter-lock (accessed 23.01.23)

4.2.1 ClipsGym: Extending Python with C++

ClipsGym is the interface between the GR environment CX and the RL OpenAI Gym environment (ClipsWorld). ClipsGym can be loaded as a python module. The interface provided by ClipsGym can be used to execute C++ functions. That means from python, C++ can be executed. This is a common way to implement python modules. Especially in high performance computing matters, C/C++ is often used. For example, the well-known python libraries Numpy [46] and TensorFlow [2] are implemented in C.

```
1  py::class_<ClipsGymThread>(m, "ClipsGymThread")
2    .def(py::init<>())
3    .def("getInstance", &ClipsGymThread::getInstance, py::
         return_value_policy::reference)
4    .def("step", &ClipsGymThread::step)
5    .def("resetCX", &ClipsGymThread::resetCX)
6    .def("create_rl_env_state_from_facts", &ClipsGymThread::
         create_rl_env_state_from_facts)
7    .def("getAllFormulatedExecutableGoals", &ClipsGymThread::
         getAllFormulatedExecutableGoals)
8    .def("generateActionSpace", &ClipsGymThread::generateActionSpace)
9    .def("generateObservationSpace", &ClipsGymThread::
         generateObservationSpace)
10   .def("assertRlGoalSelectionFact", &ClipsGymThread::
         assertRlGoalSelectionFact)
11   .def("getGoalId", &ClipsGymThread::getGoalId)
12   .def("getRefboxGameTime", &ClipsGymThread::getRefboxGameTime)
13   .def("getRefboxGamePhase", &ClipsGymThread::getRefboxGamePhase)
14   .def("clipsGymSleep", &ClipsGymThread::clipsGymSleep)
15   .def("log", &ClipsGymThread::log);
```

Listing 4.3: Part of the ClipsGym interface for extending python with C++

4.2.2 RLAgentManager and PyGuard: Embedding Python in C++

Embedding Python in a C/C++ application can provide several advantages, such as allowing users to customize the application through Python scripting or utilizing Python's rich libraries to easily implement certain functionalities. The initialization of the Python interpreter in an embedded environment can be achieved by calling `Py_Initialize()` from the main program of the application, which is responsible for communicating with the interpreter when necessary. Due to the GIL, Python threads are unable to run concurrently on multiple central processing unit (CPU) cores [45]. The GIL prevents preemptive multi-threading, which occurs when one thread takes over by interrupting another thread. It is more performing to initialize only one Python thread and keep the reference on this thread, than to create a new thread for each use of Python - in our case for each prediction of the next goal. This is because it involves the following steps: initialize the thread, allocate memory, load the references, create or load the environment

[2] https://www.tensorflow.org/ [47](accessed 27.01.23)

and agent, execute the prediction, and terminate the thread again. Five of the six steps have to be done only once by using the PyGuard which holds the reference.

Since the Python `Py_FinalizeEX()` is prone for memory leaks, especially during flushing buffered data .[3] Those leaks can occur if the Python destructor fails to delete objects and therewith would not clean certain parts of the memory. Therefore, we do not create a new python thread for each goal selection, but keep one that we fall back on when a selection is pending. In our case the class PyGuard handels the initialization of the python interpreter. In the execution mode it also keeps the reference to the ClipsWorld and provides a function to call the python `model.predict` function from C++.

```
PyGuard *
PyGuard::getInstance()
{
  if (py_guard_instance == nullptr) {
    py_guard_instance = new PyGuard();
  }

  return py_guard_instance;
}
```

```
std::string
PyGuard::predict()
{
  std::string goal_str = "";
  try {
    // get current discrete observation
    py::exec("obs = env.getCurrentObs()", py_scope);
    // predict the next action
    py::exec("action, _ = model.predict(obs, action_masks = env.
        action_masks(), deterministic=True)",py_scope);
    py::exec("print(\"Predicted action: \", action)", py_scope);
    // map the action to the next goal to select
    py::exec("goal = env.action_dict[action]", py_scope);
    py::exec("print(\"Predicted goal: \", goal)", py_scope);
    goal_str = py_scope["goal"].cast<std::string>();
  } catch (py::error_already_set &e) {
    py::module::import("traceback").attr("print_exception")
    (e.type(), e.value(), e.trace());

    std::cout <<"Python exception:" << e.what() << std::endl;
  } catch (const std::runtime_error &re) {
    std::cout <<"Python exception:" << re.what() << std::endl;
  } catch (...) {
    PyErr_Print();
    PyErr_Clear();
  }

  return goal_str;
}
```

[3] https://docs.python.org/3/glossary.html#term-global-interpreter-lock (accessed 23.01.23)

4.2.3 Individual modules

As visible in the Figure 4.7 the goal selector consists of six modules: ClipsGym, ClipsWorld, RLAgentManager , RLAgent, RLGoalSelectionRules and the PyGuard. In the following each module is further introduced.

RLAgentManager The RLAgentManager and ClipsGym are the interfaces for coupling the CX with python based RL, these modules are realized as Fawkes plugins. RLAgentManager is responsible for the connection from the CX to the RL agent, administrates the asynchronous thread of the python interpreter for training the RL algorithm and handles the PyGuard, which is further introduced in Section 4.2.3. The RLAgentManager module has two modes *training* and *execution*. For either, training and saving the trained RL agent or for loading a saved agent. The thread has a *RlGoalSelection* blackboard interface which is used in the execution mode to wake up the Fawkes plugin and to start the prediction of the RL agent. A goal selection fact is asserted with the predicted next goal. In the training mode, the thread also passes the configuration values to the embedded python interpreter. For example the loaded environment is configurable, with this the RLAgentManager module could be used in connection with other OpenAI Gym based environments. Before using any Python APIs, including those provided by pybind11, the Python interpreter must be initialized.

The Python API is a set of functions and classes that are part of the Python programming language. It is documented in the ”Python API Reference” documentation, which is available on the Python website.[4]

Pybind11 is a C++ library that allows users to bind C++ classes and functions to Python.[5]

Python Interpreter - PyGuard The PyGuard class handles the python interpreter reference. This can be done using the pybind11 class `scoped_interpreter`, which manages the lifetime of the interpreter. As long as the `scoped_interpreter guard` is alive, python functions can be called. Since we want to load libraries and modules only once in our Python environment, we keep the reference in PyGuard to this interpreter and do not open and close the environment on each function call. On the one hand this saves computation time, on the other hand it is more robust, since the Python destructor might fail for deleting some objects[6] what means some memory wouldn't be cleaned and the next start of a python interpreter would fail.

For training the RL agent the python interpreter runs in a separate asynchronous thread. This way it is achieved that the Python interpreter is active during the whole training and is not terminated when the thread time of the RLAgentManager allocated by the Fawkes core is over. The python runtime environment can always be accessed via the PyGuard, even when the thread is sleeping / has no CPU time. Since the environment continues to run in parallel / persists, variables that

[4] https://docs.python.org/3/c-api/index.html (accessed 06.01.23)
[5] https://pybind11.readthedocs.io/en/stable/ (accessed 06.01.23)
[6] https://docs.python.org/3/c-api/init.html#initializing-and-finalizing-the-interpreter (accessed 06.01.23)

have been initialized before remain present. Thus, there is no reload or new initialization for the ClipsWorld and the previously saved RL agent needed. They can be called directly when another goal selection is pending. For an initialization of the ClipsWorld the ClipsGym plugin must be loaded before.

ClipsWorld ClipsWorld offers a customized reinforcement learning environment which is used by the RLagent. It is inheriting from the OpenAI Gym environment, Therefore it implements the step and reset function as well as the attributes observation and action space. The module uses the python extension of ClipsGym for the step and reset functions.

OpenAI Gym Environment

For modeling the environment, the OpenAI Gym environment serves as a toolkit for developing and comparing reinforcement learning algorithms [48]. The gym package comes with a wide range of environments, commonly used for comparison of different RL algorithms and benchmark tests. A gym environment consists of these functions: [48]

init: defines the action and the observation space

step: determines the observation and the reward based on the given action

reset: restores the initial state of the environment

render: interprets the environment state for example to retrieve an image.

So to create a environment that fits our purpose, we have to implement these functions, this environment is then called a *customized environment*.

ClipsGym This module is the interface between the CX and ClipsWorld. It provides c++ functions for a python call by a pybind11 interface. To use the functionalities, the dynamic link library is loaded as a module in python. The vital functions are the *step* and *resetCX* function. At the same time it is a fawkes plugin with a CLIPSFeature aspect. For example for the generation of the observation space it accesses the clips facts to extract the domain predicates and domain objects, and combines them. The same way a single observation is created which represents the current state of the environment. Therefore the current domain-facts are extracted and returned to ClipsWorld, where the mapping to the discrete observation vector is made. For the goal selection initiated from the step function, ClipsGym asserts a goal selection fact and waits till the goal is evaluated. The indicator for this is the `rl-finished-goal` fact. Furthermore, it implements the extraction of the current executable goals from the knowledge base, which is invoked through the action masking function.

Python Scripts For training and saving the trained reinforcement agent a custom script was created and is loaded by the RLAgentManager module. Training the RL agent means adjusting the policy of the agent thus fitting the weights of the policy and of the critic network. This facilitates to adjust the RL agent.

The class diagram Figure 4.8 shows the relations between the different classes in the system. It illustrates the inheritance relationships between the aspects and the plugin classes in the system.

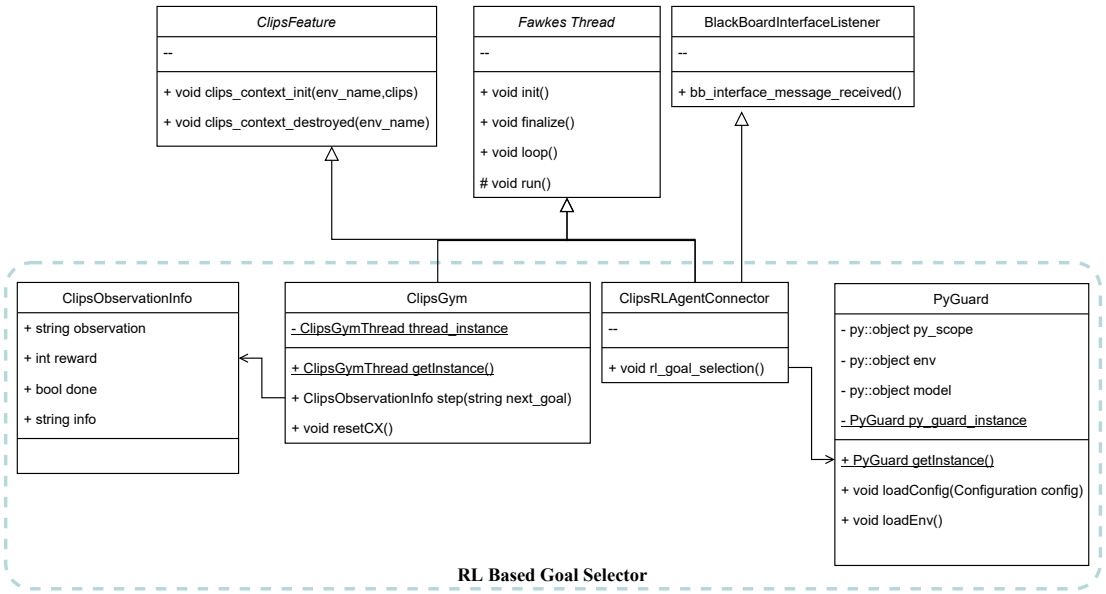


Figure 4.8: Classdiagram of goal selector and basic fawkes plugin

4.2.4 Training process

Subsequently, the training mode of the RL agent with its step function is elaborated in more detail.

First we look at the training from the environment point of view and then from the agent point of view. The agent makes an action and observes the effect of its action and also receives a reward. The environment is responsible for determining the new state from an incoming action and returning the corresponding reward, this is realized in the step function. The step function is called until a final state is reached, then the reset method sets the environment to the initial state, this is illustrated in Figure 4.9. Besides the policy update of the actor, the critic is also updated. In our case the step and reset methods of the ClipsWorld have to interact with the reasoning system shown in Figure 4.11.

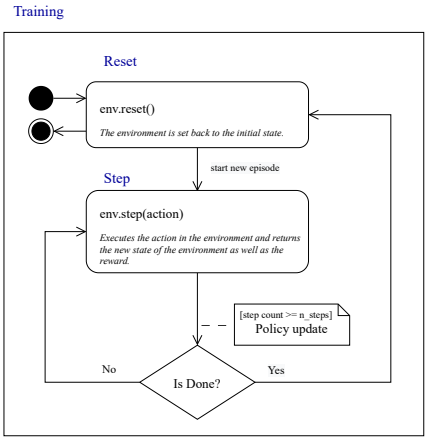


Figure 4.9: The core process of the agent - environment interaction

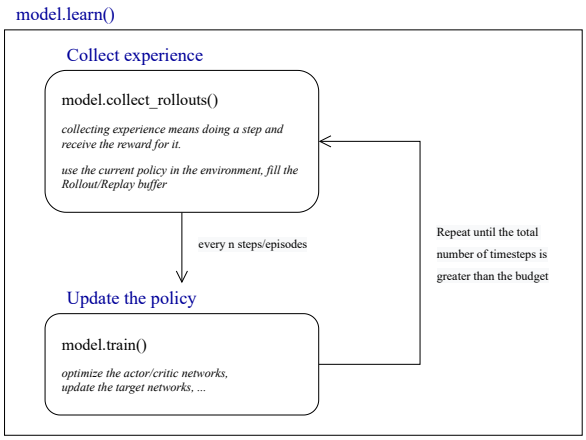


Figure 4.10: The structure of the *model.learn()* method, adapted from [39]

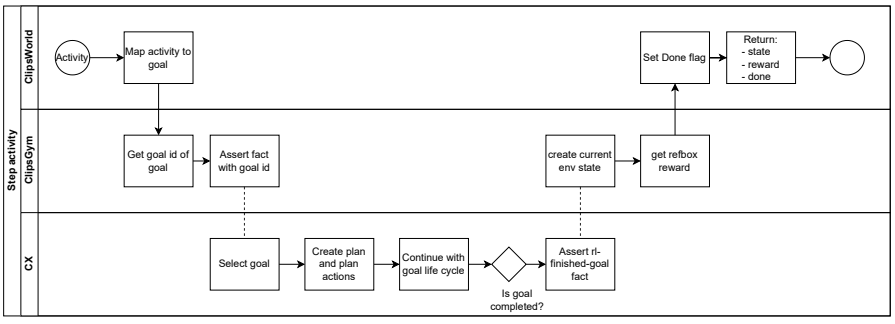


Figure 4.11: The diagram illustrates the step function on different layers

The interaction of the agent with the environment looks like this:

The hyper-parameter n_steps determines the number of step function calls that are called one after the other until a policy update occurs.

After each step function call it is checked if the episode is over, that means in our case if the RCLL game is over. Which ends after 1200 time steps (at speedup 1 this corresponds to seconds). The flag *Done* is used as an indicator for the end of a game.

In total, as many steps are executed until the *total number of steps* is reached. Or a specific callback is given, which may abort the training earlier, if e.g. a certain number of episodes, i.e. games, have been executed.

For training an StableBaseline3 agent the learning function is called. Figure 4.10 represents the internal steps of the framework.

We have the fawkes main thread, the ClipsGym thread and the RLManager thread. ClipsGym and RLManager thread always have a certain time window for the loop function, which must not be exceeded. But now the RL agent must be active during the whole training, so an asynchronous thread is created, which is used to call the Python script to train the agent. This allows ClipsWorld to run asynchronously to the CX. This means that the CX can regulate applications, provided it is not locked. Regardless of what the RL environment is doing.

Integration in the goal tree

Before going deeper into the integration, a small digression about the goals and the goal tree in RCLL is given. To build up an understanding for the goals and goal classes, subsequently the goal classes are used in the action space definition, as described in Section 4.2.5. The next part refers to the goal reasoning implementation of the Carologistics Team [49].

Goals in RCLL Goals in the RCLL are used to handle the different orders and production steps within a game. The Team uses as goal reasoning system the CX, introduced in section 2.1.1. A compound goal in RCLL is for example the production of a $C1$ product. Each production step is a goal. If the step can be split in smaller steps, it's a compound goal. The smallest production steps are simple goals.

Some of the production steps can be done in parallel and some require to be done in sequence. E.g. the cap should not be mounted on a workpiece without a ring. But the prearrangement (ring payment and buffering the cap) can be done in parallel.

Figure 4.12 shows a reduced extract from a goal tree. The Carologistics central RCLL agent approach is based on one central agent, which is responsible for delegating the goals to the robots and also processes the instruction goals. We want to use the goal selector for the production and not for the instruction goals. This is further explained in Section 4.2.5. For integrating the RL based goal selector into the existing Carologistics central agent, the goal selection rules are slightly changed. If a robot is waiting for a goal to pursue, the rl goal selector is called. The goal selector extracts all current executable goals and the predicates of the CX facts. Then filters the goals, based on the goal space and predicts the next goal based on this.

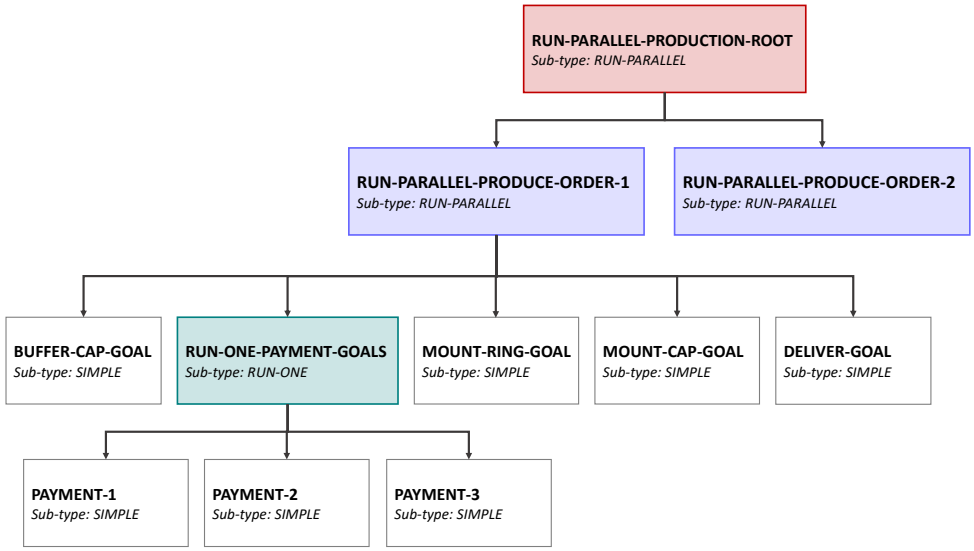


Figure 4.12: An example of a goal tree

Through the filtering process, the goal selector is easier to customize, as only the goal space needs to be adjusted and no other CX rules. The area of application of the goal selector is also larger, as it can be used in parallel with other goal selectors.

4.2.5 Customized action and observation space

In such complex domains as the RCLL, it is not feasible to automatically generate all possible combinations of predicates and objects or goal classes with parameters. Instead, it is more sufficient to manually configure the observation and action space while limiting to the logically possible states. This leads to a feasible trade-off between training time and training performance of our agent.

Observation space

The observation space contains all predicates which describe the environment space.

In case of the RCLL the PDDL domain model can be used as baseline. For instance, the position of the robot, the machine state, order information and production progress can be included in the environment state. Here is a small extract of the domain model predicates, which could be used to describe the observation space:

```
(at ?r - robot ?m - location ?side - mps-side)
(holding ?r - robot ?wp - workpiece)
(mps-type ?m - mps ?t - mps-typename)
(mps-state ?m - mps ?s - mps-statename)
```

```
(order-base-color ?ord - order ?col - base-color)
(order-ring1-color ?ord - order ?col - ring-color)
(order-fulfilled ?ord - order)
(wp-unused ?wp - workpiece)
(wp-at ?wp - workpiece ?m - mps ?side - mps-side)
;production progress
(wp-base-color ?wp - workpiece ?col - base-color)
(wp-ring1-color ?wp - workpiece ?col - ring-color)
(wp-cap-color ?wp - workpiece ?col - cap-color)
```

Action space

Similar to the observation space, we have a discrete action space which is based on goal classes and other goal properties. Using only the goal classes as action space is not enough, as it wouldn't be possible to distinguish between a `Buffer-Cap` goal of order one and a `Buffer-Cap` of order two.

Calculating the action space from a combination of the goal class with all parameters of a goal is not feasible. When examining the goal parameters, it became clear that not all parameters are "game independent". A parameter is game independent, if e.g. when loading another game exactly the same goal with this parameter exists again. And would lead to the same effect. Combining the order number with the goal class is the first suggestion for the action space. But the effect of `Buffer-Cap#order1` has in one game the effect that a black cap is buffered and in the next game the effect of a gray buffered cap. So an agent trained in this way can only be used for a fixed game - with fixed orders. For example, the mps-type (BS or CS) is game independent. The specific MPS, depending on the selected team e.g. for Cyan C-BS on the other hand is not game independent. Likewise, the goal-id, or the workpiece which also has an id generated at runtime, is not game independent. Since the RL agent should not only be trained for a specific game, it should also be applied to another game, not previously trained, an overfitting of the agent must be prevented. By omitting "game dependent" parameters the danger of overfitting can be reduced.

To determine the size of the action space required for RL we need to solve the underlying combinatorial problem. Since several goal parameters can have the same value it is a variation with repetition. The number of relevant goal classes is $|C| = 10$. To calculate the upper bound of our action space, we take the maximum number of parameters $|X| = 7$ and the largest set of values of a parameter $j = 5$. This results in an upper limit for the action space with: $\mathbb{O} = |C| * |Var_j(X)|$ with $10 * 7^5 = 168.070$. Many of these variations do not occur for logical reasons, so we refrain from a built-in automatic goal space generation and pass a list of logically possible goals to the goal selector. More important for an decision is the location of the workpiece and the target location. Furthermore, the cap-color and ring-color are important.

Listing 4.4: Extract of the used action space

```
"BUFFER-CAP#cap-color#CAP_BLACK",
```

```
"BUFFER-CAP#cap-color#CAP_GREY",
"MOUNT-CAP#wp-loc#C-BS",
"MOUNT-CAP#wp-loc#C-CS1",
"MOUNT-CAP#wp-loc#C-CS2",
"MOUNT-CAP#wp-loc#C-RS1",
"MOUNT-CAP#wp-loc#C-RS2",
"MOUNT-CAP#wp-loc#C-SS",
"PAY-FOR-RINGS-WITH-BASE#target-mps#C-RS1",
"PAY-FOR-RINGS-WITH-BASE#target-mps#C-RS2",
"PAY-FOR-RINGS-WITH-CAP-CARRIER#target-mps#C-RS1",
"PAY-FOR-RINGS-WITH-CAP-CARRIER#target-mps#C-RS2",
"PAY-FOR-RINGS-WITH-CARRIER-FROM-SHELF#target-mps#C-RS1",
"PAY-FOR-RINGS-WITH-CARRIER-FROM-SHELF#target-mps#C-RS2",
"MOUNT-RING#ring-color#RING_BLUE",
"MOUNT-RING#ring-color#RING_GREEN",
"MOUNT-RING#ring-color#RING_ORANGE",
"MOUNT-RING#ring-color#RING_YELLOW",
"DELIVER#",
"WAIT-NOTHING-EXECUTABLE#"
```

Even though the action space is very large. In most of the environment states, only a few goals are really executable. For the rest the goals precondition are not fulfilled. Thus an environment state action matrix is a sparse matrix, where most of the matrix elements have a 0 value. To prevent the RL agent from frequently selecting invalid actions - non executable goals, it is possible to use action masking [50]. This causes the agent to select only available actions, which means that only one executable goal becomes selected.

Action Masking - Realization The action masking is used to prevent the RL agent from frequently selecting non executable goals. The `action_masks` is a function implemented in the ClipsWorld and called by the rl agent. To generate the action mask the ClipsWorld module calls the ClipsGym to get a list of executable goals. This list is prepared in the ClipsGym module by accessing the knowledge base of the CX.

4.2.6 Reward function

In the long run, a goal selection is sufficient if the production went through quickly thus if the team produces the most orders and delivers them in time. As this long term reward can not be matched directly to a single goal selection it is important to determine „short term rewards“. The RCLL rulebook lists the scoring points during a production [19, p.21]. As some of the goals map directly to dedicated production steps, the RCLL points can be used as short term reward. For a fast production, often means to perform one goal instead of two. Example: In case of *discard capcarrier* and *get base for ring payment* it might be faster to perform *use base for ring payment* instead.

For the other goals there is the possibility to evolve own rewards. This would be quite similar to dispose priorities to the goals. Besides, the danger is to influence the agent already unconsciously during training, so that it follows a certain strategy, which was implemented by rewards. Therefore, that we have to work with a sparse reward function. In the thesis it has to be further investigated how the parameters of the RL algorithm have to be adapted to handle such a sparse reward.

4.2.7 Reset Process

The implementation of the reset function is special because not only the parameters of the python environment have to be reset, but a reset of the surrounding system refbox is triggered and partly the CX.

Therefore, a multi-stage reset process was developed in the CX. This process is triggered by asserting a `reset-game` fact in the ClipsGym. This happens in the reset method of the ClipsGym which is called by the ClipsWorld after resetting the own parameters. The reset method of the ClipsGym asserts the `reset-game` fact in stage zero. It waits for 4 seconds before it checks the `reset-game` fact stage. If it's not yet finished it waits again. Until either the maximum waiting time is reached or the `reset-finished` fact is asserted.

This multi-stage process consists of the following stages:

1. **[STAGE-0: Change refbox phase to POST_GAME]** the stage is started by the ClipsGym and leads to a refbox phase change if it not already happened
2. **[STAGE-1: Change refbox phase to SETUP]** additionally the refbox state is set to paused. Generally a refbox phase and state change can be initialized through a blackboard message. After an incoming Blackboard message confirms the phase and state change of the refbox, the fact `(domain-facts-loaded)` is deleted. This is the precondition for the next stage.
3. **[STAGE-2: DELETING]** In this stage compound goals counting root goals and simple goals are retracted. Just like the game specific facts: plan, plan-action, order-meta, refbox-agent-tasks, rl-goal-selection.
4. **[STAGE-3: DOMAIN-WM-FLUSH]** The domain specific world model facts are retracted. After the retraction all facts are saved to a file, for debugging reasons.
5. **[STAGE-4: Change REFBOX STATE]** The refbox state is set to running and the inital facts are loaded.
6. **[STAGE-5: Change REFBOX PHASE TO PRODUCTION]** the reset is almost completed, the initial state before the game start is restored. Therefore, now with the change of the refbox phase to production the game can be restarted.

7. **[STAGE-6: FINISHED]**The reset process is completed when the refbox has successfully changed phase. Since the step function is executed next, it is also important here that the first goals are generated before the reset process ends. So that a goal selection decision can be made.

4.2.8 Integration of goal selection rules

In the context of the Carologistics Team, the central agent refers to the collaboration of robots based on the master-slave paradigm. Therefore, there is a high-level central master agent and one to three subordinate robots. The three robots do not choose their own goals. Instead the underlying concept is that the three robots execute production goals, while a central agent has the responsibility of assigning goals and is in charge of the machine instruction goals.

The goal selection of the RCLL central agent works in the following way:

- For each incoming order a separate goal tree, so called order production tree is generated.
- The selection is carried out on the basis of priorities. A distinction is made between the goal sub-types. For the sub-types *root*, *run-all* and *run-parallel* independent priority based selection criterion exist.
- It selects the root of an order production tree if it has the highest priority and is not interfering with another currently selected order.
- When the production root is not already in the selection criterion list, its added to it
- goal reasoner remove non executable goals from selection criterion list
- If there is no achieve goal in the state SELECTED, EXPANDED or COMMITTED the selection across types is applied. This selection merges the selection criterion list into a list of choices and the goal with the highest priority is identified.
- Then the corresponding agent is determined. In case of an actual robot the executability is flushed, which means removing all other assignments of that robot and the waiting status.

The RL goal selection is integrated in the goal reasoner for the simple production goals. Thus it decides which production step should be conducted next. Based on the parameters the production step goal can be matched to an order. For the machine instruction goals which are carried out by the master agent are selected by the same selection mechanisms as with the central agent from the Carologistics team.

5 Evaluation

In order to assess the capabilities of the novel goal selector, the evaluation of the system is split into two parts - the evaluation of the training process and the evaluation of the execution process. The latter includes the comparison of the RL goal selection with the goal selection from the central agent of the Carologistics team.

This experiment is designed to show the ability of training the reinforcement learning agent for the goal selection within the Fawkes robotic setup. Therefore, it provides the possibility of training the agent in the simulation environment first. Later on training can be continued in the real world and the trained agent can ultimately be utilized in the desired environment without changing the framework.

Before the specific setup is discussed, the common settings of the training and execution mode experiments are presented.

5.1 Experimental Setup

To perform the evaluation in the following, the Fawkes framework with the CX is used. As discussed, the structure of Fawkes is similar to a Belief-Desire-Intention agent. The bottom layer serves as a driver for the sensor data. The middle layer maps an action to a skill and translates the concrete skill, like *move*, to the low-level actuator control for e.g. the motor control for the wheels. The Skiller plugin is an essential plugin within Fawkes that maps the skill to be executed to the Lua-based behavior engine. For training and testing the goal selector the skiller simulator, which is located in the midlayer, is used. it is thereby possible to save computational time and resources compared to the Gazebo simulation or even a real-world run.

5.1.1 Game setup

To create reproducible training and games, an RCLL game was generated with the refbox which was used for the training and for the evaluation of the agent in execution mode. The RL agent is now at first trained and tested on a fixed dedicated game. This means that the field layout is fixed as well as the sequence of the given orders. The goal selector is initially only used in a game with a single robot, so that the selection can be clearly assigned and analysis is facilitated later on. In order to be able to perform the same number of tasks during a game with one robot as with three robots, the speed of the actions performed by this robot is tripled in the simulation.

The following tasks must be completed in the generated game:

We train and evaluate the agent on one generated RCLL game. Below the order number and the complexity the delivery window of the order is illustrated in the Figure 5.11. The order for

a game overtime is not included, as this does not appear during training in our setup. The key points of the game are:

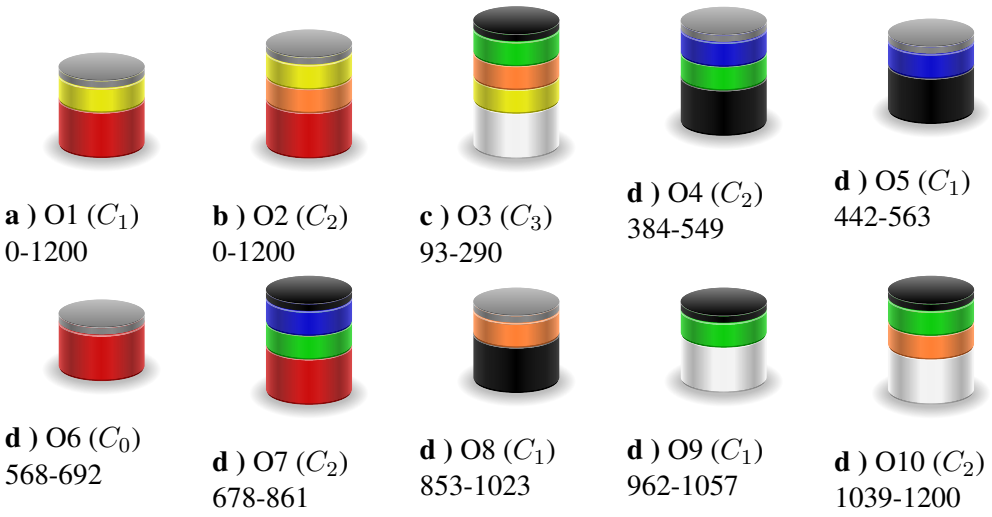


Figure 5.11: Products and order configuration of the specific RCLL game.

Figure 5.11 illustrate the single orders with their complexity and the delivery time window. This game contains one $C3$ order, four $C2$, four $C1$ orders and one $C0$ order. In total, 825 points can be achieved in the game, without overtime. It is unrealistic for an agent to achieve this number of points in a normal game. The corresponding playing field is illustrated in the appendix.

Besides this one game, which is used for training another benchmark set of 10 games is generated.

5.1.2 Speedup

When speeding up, a distinction must be made between the speedup of individual actions, for which the skiller simulator or execution time estimator is responsible, and the speedup of the RCLL game, for which the refbox is responsible.

Execution time estimator The simulator not only recreated the effect of an action but also the time required for an action to conduct. For instance, if a robot is moving fom the front of the home zone to a MPS, it will require a certain amount of time. Depending on the game setup, the MPS's location may differ, and the time may change accordingly. This time is simulated and controlled through time steps. Speed 1 represents normal time while a speedup of 2 halves the execution time of actions.

Refbox In addition to accelerating the individual action times performed by a robot, the environment can also be accelerated in the simulation. This means that the refbox can also run with a speedup factor. The game duration of 1200 time steps remains the same. At speed 1, the time steps correspond to seconds, i.e. the 20 minutes that an RCLL game requires. At Speed 2, one time step corresponds to only half a real second, so only 10 minutes pass until an RCLL game is finished.

Speed Configuration

	Speed 2	Speed 4
Refbox	2	4
Execution-time-estimator	6	12
ClipsGym	4	8

Speed 4 means that the refbox performs the actions at 4 times the normal speed.
In the goal selector is used for a robot. In order to produce a similar amount with one robot as with three robots, it is necessary to let this one robot perform the actions faster. This can be achieved by setting the execution time estimator in the simulation. Thus it is set to $4 * 3 = 12$.

Figure 5.12: Speed Configuration

By speeding up the ClipsGym, the minimum sleep time of the thread before checking whether the robot has finished its action can be reduced.

The central agent played several different RCLL games in the simulation to test the speedup factor. It turned out that depending on the speedup, the agent can score different numbers of points in a game. Table 5.1 summarizes the findings of these runs. Therefore the experiments are conducted with different speed settings. The Table 5.12 provides exact values of the individual speedup variants utilized in the experiments.

	Speed 1	Speed 2	Speed 4
N	4	6	6
Missing	2	0	0
Mean	723	310	117
Median	776	370	92.5
Stdev	126	173	82.4
Minimum	535	4	31
Maximum	806	477	226

Table 5.1: Total game points of the central agent with refbox speedup 2

Since a speedup of 1 is not feasible for training the RL agent, fewer speedup tests were carried out. In the game with a total score of four points, an MPS downtime, i.e. the MPS was BROKEN

and led to consequential errors. More detailed evaluations of these games can be found in the appendix.

In general, Fawkes is started with the central agent. Through a configuration flag of the RLAgentManager, the RL based goal selection can be switched on and off. This configurations includes whether to start the agent in training or execution mode. In the conducted experiments, the metrics were collected accordingly to a fixed and a variable game and with a speedup factor of 2 and 4.

5.1.3 Training the RL agent

To evaluate the training process, an untrained RL agent is trained several times on a fixed game. The loaded game is introduced in Section 5.1.1.

The RL agent trains for 300 steps. This corresponds to about 10 RCLL games with a speedup of four. For the analysis of the training process the received game score during the training period is compared. The agent with the best training performance is used for further evaluation with the execution mode. In total six independent training runs are conducted to obtain a trained agent. Important for the training is the configuration of the RL algorithm and policy. Figure 5.13 illustrates the rewards the RL based goal selector agent reached during the training.

Algorithm and Policy configuration

The experiments are conducted with the MaskablePPO algorithm from StableBaseline3 . The MaskableActorCriticPolicy was used as the policy. The actor is a Neural-Network (NN) which represents the policy function with two hidden layers of size 64×64. In the same way, an NN of the same size was used for the Critic.

Thus there are $270 \times 64 + 64 \times 64 + 64 \times 39 = 23.872$ weights for one network. For both networks, the weights are adjusted during the policy update. The table 5.2 shows a summary of the relevant configuration. An extended table is provided in the appendix 7.1.

A detailed list of the parameters can be found in Table 7.1. The discount factor was chosen particularly high because the future total reward is rather important for a long-term strategy. It is more relevant than the immediate reward. This applies especially in our RCLL context as we have a sparse reward, where some goals are not directly rewarded but only when the product has been built and delivered. Or rather, there are a lot of points in the finishing and distribution of a finished product compared to some of the previous production steps.

Overall, the evaluation of the novel goal selector involves a well-designed training setup. With a fixed game setup for training it is realistic to obtain a training effect on this fix game despite a low number of total time steps.

	Parameter	Value
RL Policy MaskableActorCriticPolicy	$learning\_rate$	0.0003
	discount factorγ	0.99
	policy function network	$[64 \times 64]$ two hidden layers with the size of 64
	value function network	$[64 \times 64]$ two hidden layers with the size of 64
RL Training MaskablePPO	Observation space size	270
	Action space size	39
	n_steps	3 number of steps after which a policy update is triggered
	$max\_episodes$	10
	$total\_timesteps$	1200
	$save\_freq$	100 using a CheckpointCallback for saving the RL agent every 100 steps while training

Table 5.2: Parameters for the training and experiments

5.1.4 Executing RL agent

For conducting the experiments in the execution mode, the configuration of the RLAgentManager is adjusted to:

- Activate or deactivate the RL based goal selection; Deactivation means that the central agent is executed as baseline for the experiments with its handcrafted goal selection.
- The training mode is set to *false*, thus the RL goal selection is started in the execution mode.
- Setting the model reference to either load the trained or untrained RL model.

An untrained reference model is an agent where no policy update has taken place. This random agent is generated through loading the setup and stopping the training after two step function calls. As trained reference model an model which is trained for 300 steps is used, which was

obtained as result in Section 5.2.1. The evaluation of the execution process shows, the ability of using the trained agent in a game to make instant decisions.

5.2 Visualization of Results

5.2.1 Training results

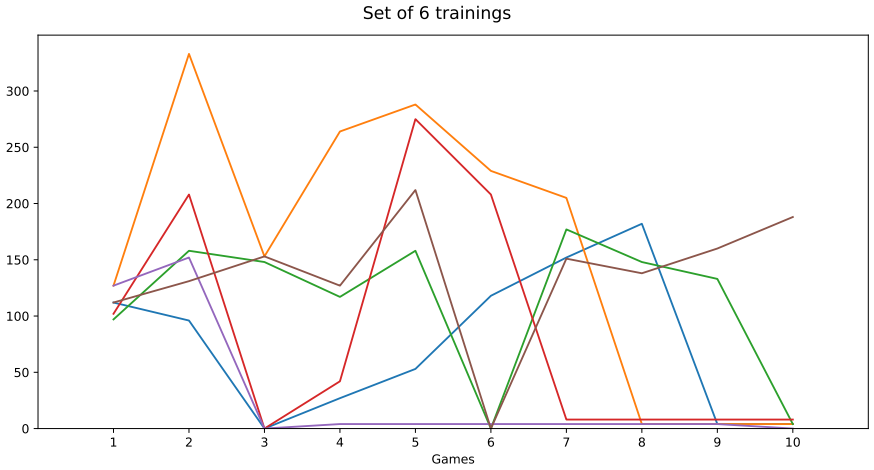
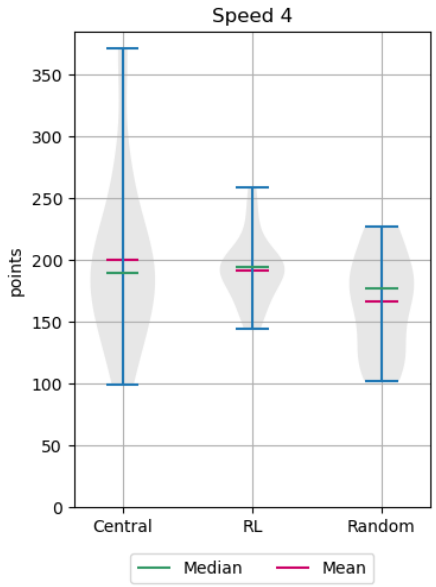


Figure 5.13: Results of six trainings for ten games of the RL based goal selector, each color represents another training run

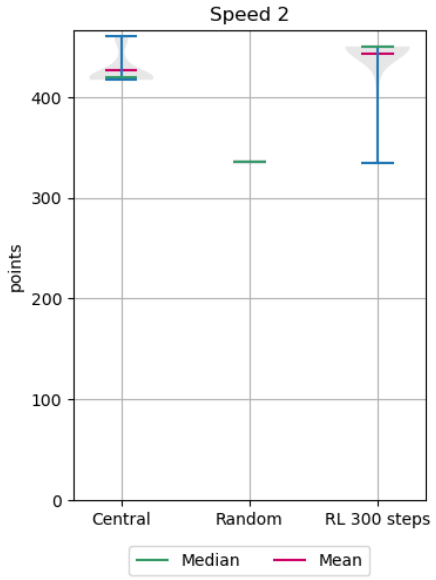
5.2.2 Execution results

Total Score



Total points			
	Central	Random	RL
N	25	25	25
Mean	200	166	192
Median	189	177	194
Stdev	67.3	39.9	26.9
Minimum	99	102	144
Maximum	371	227	259

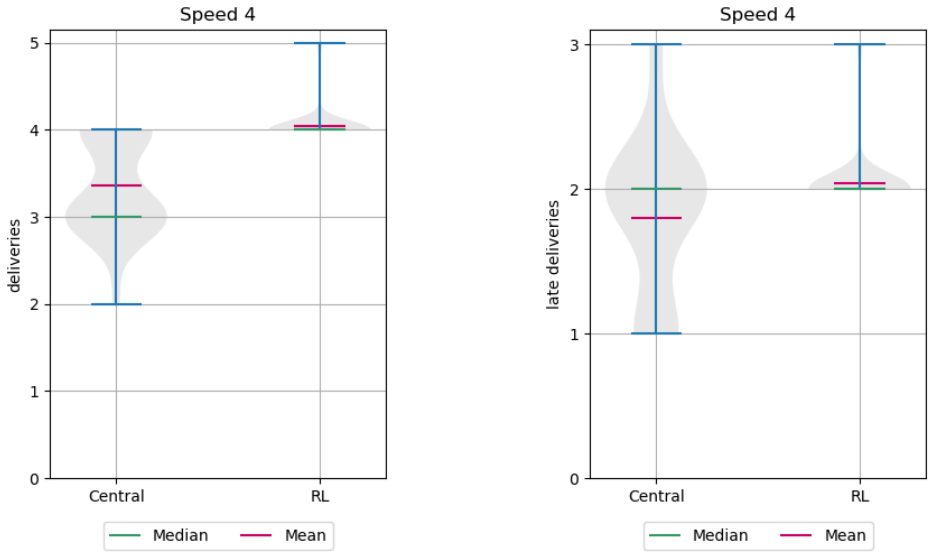
Figure 5.14: RCLL scoring statistics: Playing 25x *Game1* with speedup 4



Total points			
	Central	Random	RL
N	25	25	25
Mean	427	336	443
Median	419	336	450
Stdev	14.8	0.00	24.4
Minimum	417	336	335
Maximum	460	336	450

Figure 5.15: RCLL scoring statistics: Playing 25x *Game1* with speedup 2

Deliveries and Late Deliveries



d) Deliveries **d)** Late deliveries

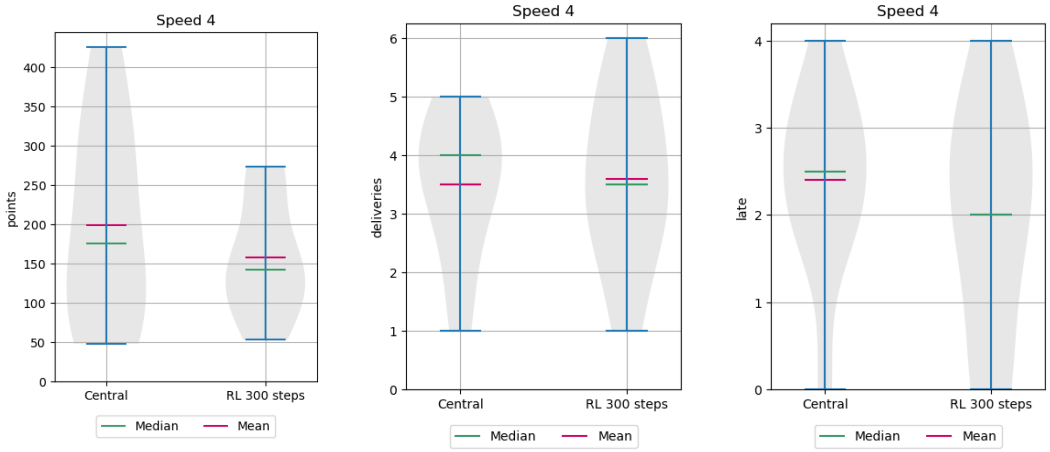
Figure 5.18: RCLL deliveries statistics of 25x playing Game 1 with speedup 4

	Central		Random		RL	
	deliveries	late deliveries	deliveries	late deliveries	deliveries	late deliveries
N	25	25	25	25	25	25
Mean	3.36	1.80	4.00	3.00	4.04	2.04
Median	3	2	4	3	4	2
Stdev	0.569	0.577	0.00	0.00	0.200	0.200
Minimum	2	1	4	3	4	2
Maximum	4	3	4	3	5	3

Table 5.3: RCLL deliveries statistics: Playing 25x *Game1* with speedup 4

	Central		Random		RL	
	deliveries	late deliveries	deliveries	late deliveries	deliveries	late deliveries
N	25	25	25	25	25	25
Mean	6.00	4.00	6.00	2.00	6.88	3.96
Median	6	4	6	2	7	4
Stdev	0.00	0.00	0.00	0.00	0.440	0.200
Minimum	6	4	6	2	5	3
Maximum	6	4	6	2	7	4

Table 5.4: RCLL deliveries statistics: Playing 25x *Game1* with speedup 2



d) Points **d)** Deliveries **d)** Late deliveries

Figure 5.22: RCLL statistics of playing 10 different games with speed 4

Different Games

Delivered Orders

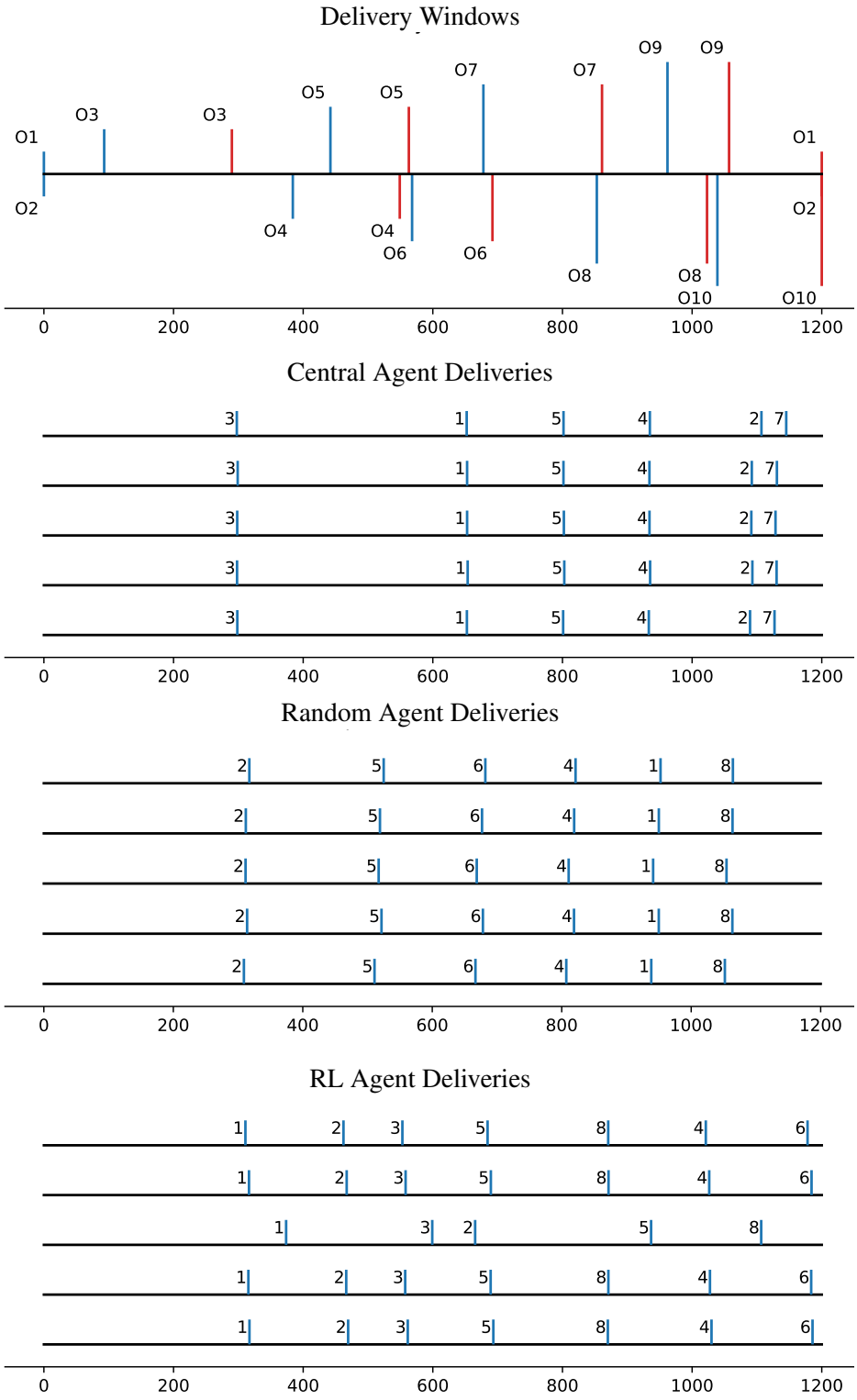


Figure 5.23: Five RCLL games played by the central and the random agent with speedup 2

6 Discussion

The main part of the work was the implementation of the goal selector and the proof of concept in the sense that the RL algorithm can be trained and executed with the architecture design. The functioning integration into the robotic execution framework Fawkes is of great importance. The use case in the Fawkes framework and for the RCLL setup is very specific and therefore there is no comparable work. The thesis is an experimental approach, so the discussion is based on the comparison with the existing central agent of the Carologistics team and no further applications outside the context are analyzed.

6.1 Comparison to the central agent

Total Score The results show that the RL agents scores in average 192 on the game with speed 4 it is trained for. That is a little less than the central agent achieved on average. Since the RL agent standard deviation is lower than of the central agent, the RL agent more often scores around 192 points in a match. The disadvantage is that the RL agent has no upward outlier with many points. One advantage is that it is more stable in achieving the expected reward. Especially as the minimum of the RL agent is significantly higher than central agent, it makes sense to use the RL agent in this game. The RL agent also performs very well at speed 2. Since mean and median are larger here than with the central agent. The standard deviation is greater for the RL agent, as only 335 points were achieved in one game. If you look at the graph, you can see that this low number of points rarely occurs. Significant improvements are achieved by training the agent. Even with only 300 steps of training the agent achieves clearly more points than without training.

Deliveries The analysis of the deliveries and late deliveries hardly differ for both agents, mainly because the count of orders in a RCLL game is small. As visible in Figure 5.18 the RL agent assemblies more deliveries with less late deliveries on average. A possible reason for this behavior is that the RL agent chooses different orders. It seams that the RL agent is more likely to choose orders with less complexity to produce a bigger amount of orders.

Therefore the delivered orders and their sequence is more interesting. Although the agents score similar points in the RCLL game, different products are produced.

Figure 5.23 illustrates that the delivery windows of the produced orders are quite reproducible for a specific game on speedup 2. Only the RL Agent shows a run with a time shift backwards. A possible reason could be that it may have taken longer than usual to start the RCLL game with

the goal selection. The figure pictures also that, the set of delivered orders of the agent differes. A reason for this is, that the central agent prioritizes order of $C3$ complexity thus it start with the production of O3 first. Another difference of the agents is, that the RL agent delivers around seven orders, while the central agent and the random agent deliver six orders. A reason for this could be, that the RL agent learned to produce as much orders as possible in the given time. And never really explored the production of a $C3$.

Figure 7.5 shows in case of a higher speedup the decision of which order to produce differs. And it shows that through the runs of one speedup similar decisions about which order produce are made. The central agent usually produces the orders in the following order: $O3, O2, O4, [O10]$, where order 10 is not always delivered. While the RL agent follows a different strategy and builds the following orders in the order:$O1, O5, O2, O7$. Order four of the central agent and O5 and O7 of the RL are always late deliveries.The variance of the delivery point of an order is bigger at Speed4. Which leads sometimes to late deliveries. Thus, the total game score alone is not sufficiently informative.

Based on these results the conclusion is, that the RL agent is comparable to the central agent regarding the game score.

Our approach has a higher mean of total game point in a fixed game setup where it is trained for. Additionally it still achieves comparable results on different games, where it is not trained for. Therefore investing more time in training the RL agent would possible lead to better results on different games as well.

In view of the count of deliveries as metric for the evaluation the RL agent on different games. The RL agent has a slightly higher mean, which is better as it means the RL agent produces more orders than the central agent during a game. Furthermore, the mean of the late deliveries is lower, thus it is more likely that the RL agent delivers on time.

The results demonstrate that it is possible to train an RL agent within the Fawkes framework and that the agent's performance improves within the first few games. However, it should be noted that our RL agent was only trained for 300 steps and we expect that hyperparameter tuning and the use of learning and exploration functions could significantly improve its training.

Related work have trained RL agents for many more steps. For example, [34] trained over 1M iterations the CNN with a final inner product layer of four outputs for deciding between four action classes. In comparison, the action space of the novel RL goal selector is eight times bigger. This highlights the future for even better performance after a longer training. The potential of the RL agents as goal selector in the goal reasoning context is demonstrated.

6.2 Limitations

6.2.1 Training the RL agent

StableBaseline3 provides a lot of possibilities to optimize the training of the RL agent for obtaining a well performing model. One downside of the current setup is that the RL agent can only be trained on one environment instance which prevents parallelized learning with vec_env in our architecture. The GIL prohibits the tensorflow monitoring while training. This means that a live check of the current training progress is not possible. As a consequence, the analysis of the training metrics have to take place after the training has ended.

Applying hyperparameter tuning is no integral part of this thesis. Tuning these hyperparameters can be done through a process called hyperparameter optimization, which requires multiple training sessions of the RL agent with different hyperparameter settings. Afterwards we can select the particular hyperparameters that lead to the best performance of our RL agent. StableBaseline3 provides for example an Optuna [51] integration for hyperparameter tuning. However, integrating Optuna into Fawkes is not straight forward applicable and requires extensive analysis, because launching different environments means in our case to launch multiple Fawkes instances and multiple refbox instances in parallel. This again leads to difficulties with the GIL of our current architecture.

6.2.2 Refbox

In the provided central agent setup, unexpected MPS downtimes sometimes occur, for example by executing an instruction action several times in a row. This has the consequence that no more points are scored in the rest of this game, causing a detrimental effect on the training of the RL agent: The game must be interrupted early and the reset is started in order to avoid training the wrong behavior. The MongoDB, which is required for loading a certain game from the refbox, crashes after about 25 games. These are further reasons why no detailed training can be carried out.

6.2.3 Timing

In a real robot system, many complex calculations take place in parallel. As a result, there are small but present delays between messages, actions and tasks. These delays are more noticeable when the robot actions themselves take less time due to setting a speed up. Therefore we can not set any desired speedup factor as too high values would cause too much timing problems, but have to rely on longer training.

In conclusion, there is a significant overhead when training on a complex real execution framework with real dynamics which are for example given through the refbox.

Within the framework of the Master's thesis, only a limited amount of time was set aside for training and evaluating the RL agent. Since an RCLL game requires at least 5 minutes, because a higher speedup factor leads to unreliable results. Due to the thesis time scope, the refbox issues and the limited reliable speedup, only six RL agents are obtained that could be used for evaluation. Each of these agents was trained up to 300 steps. Training only this few number of games is insufficient to produce good results. The more training, the better the models, as long as there is no overfitting.

The experiments on the benchmark set of games were also carried out in a very small number. Therefore, a statistical statement is only possible to a limited extent. Furthermore, it was not investigated exactly how many orders are produced in parallel.

6.3 Future work

As mentioned in the discussion, there are many related approaches where the RL agent has been trained longer. It is not possible to pick a certain number of total time steps that always leads to a sufficient training of the agent, but the size of the NN used in the policy can serve as a guide. In the case of a network with 270 observations as input, and two hidden fully connected layers of size 64 and 39 actions as output for example, there are 23872 weights. Consequently, it is conceivable that after around $23k$ steps a successfully trained agent is reached.

Since the PPO agent does not update the policy after every call of the step function, the number of steps performed and the total number of observations made during this time are important parameters. Therefore, it is suggested to first improve the training of the RL agent by hyperparamter tuning.For the learning rate in particular, hyperparameter tuning should be carried out. Alternatively, the learning rate could also be described by a function.

It is important that the agent explores more at the beginning During the ongoing training process the exploration rate should be reduced. With increasing number of completed games or increasing diversity of delivered orders the exploration rate should be reduced further on. This way, the agent will produce various orders and receive the respective feedback instead of only processing those orders which give the highest rewards. Starting off with a low exploration factor could lead to the agent only building a C2 as he learned that this will lead to the highest points, while missing to learn that a C3 would be even better as he never built or finished to deliver one.

To avoid overfitting the agent to a specific game, the agent should be trained with different games. For example, a benchmark set of 25 games can be generated. In this case, the resetting process and the refbox interface needs small adjustments to make it configurable to load a new game while training.

Furthermore, it is conceivable to create different RL-goal-selectors which vary by the chosen RL algorithm. In this case it would be interesting to investigate if there are differences in the delivered products or in the order of their delivery. As long as an algorithm is chosen that can handle a discrete action space, the algorithm is easily interchangeable with the architecture presented in this approach.

Beyond the parameters for the policy optimization and the algorithm, analyzing and improving the parameters for the ClipsGym can have an impact on the time that elapses between goal selections. Thus it is an important factor for the overall performance of the agent. An alternative to the integration of the goal selector into the reasoner is to introduce a new *RL goal* as root, compound note, which triggers the RL based goal selection for its child goal. This would lead to a flat goal tree, which is easier to maintain. In our approach a separate production tree is created for each order. However, the handling of several parallel goal trees must be reconsidered for a *RL goal*.

As we have seen the agent performs well in this fixed game scenario. Consequently, further research is necessary to determine the effectiveness in a broader range of scenarios.

7 Conclusion

The goal of this thesis was to develop a RL based goal selection which overcomes the weaknesses of a handcrafted priority based selection, that is not generalizable and very domain specific. Therefore, an architecture design of how to integrate the goal selector into a goal reasoning system was provided. This implied the integration of the goal selector in a robotic execution framework.

GR agents were employed for complex decision making processes like those given in the RoboCup Logistics League context. The goal selector is trained and tested in the RCLL domain with the settings of the Carologistics Team. This setting provides a high dynamically changing environment, where robots are manufacturing autonomously incoming orders. The decision of which production step is reasonable to proceed next is important for the long therm strategy of the game. For this decision the novel goal selector is applied.

It replaces the hand-crafted selection criteria and can facilitate the implementation of the goal tree, as no compound goals are needed for the selection anymore. Thus the conceptual idea of applying RL on a goal level is introduced. The action space of the RL agent corresponds to a set of possible goals in the domain. Currently executable goals are used for the action masking process.This thesis describes a development for integrating the RL based goal selection into the robotic execution framework Fawkes. And more specific into the reasoning component of the CX. Connecting the execution framework with RL entails connecting C++ with Python. Thus with relying on the Pybind 11 library two new Fawkes plugins are developed which are using on one hand embedding python in C++ and on the other hand extending python by providing a python loadable package. In the evaluation the novel goal selector is compared against the Carologistics' central agent goal handcrafted goal selection. The approach is evaluated based on the total game score and the count of deliveries in the RCLL game. The RL agent had a higher mean of total game point in a fixed game setup where it is trained for than the central agent.

Additionally it still achieved comparable results on different games, although it was not trained for them. Therefore, investing more time in training the RL agent will facilitate reaching better results on different games as well.

The approach highlights the integration of reinforcement learning in the current goal decision of a robot in the real world. Ideally,it enables the robot now to continue training in the real world to make better decisions. As RL is already used for action decisions it is suggestive to adapt it further on for goal selection. The knowledge gained will also be used to derive insights for the optimization of robots in industrial applications and thus further expand the efficiency and suitable range of applications for highly automated robots in our future day-to-day life.

Bibliography

[1] Nina Kutzbach and Christopher Dr. Müller. *World Robotics 2022 – Service Robots*. VDMA Services GmbH, Lyoner Str. 18, 60528 Frankfurt, 2022. ISBN 978-3-8163-0753-2.

[2] Nina Kutzbach and Christopher Dr. Müller. *World Robotics 2022 – Industrial Robots*. VDMA Services GmbH, Lyoner Str. 18, 60528 Frankfurt, 2022. ISBN 978-3-8163-0752-5. URL https://ifr.org/img/worldrobotics/Executive_Summary_WR_Industrial_Robots_2022.pdf.

[3] David W. Aha. Goal reasoning: Foundations, emerging applications, and prospects. *AI Magazine*, 39(2):3–24, 2018. ISSN 2371-9621. doi: 10.1609/aimag.v39i2.2800. URL http://incompleteideas.net/book/RLbook2020.pdf.

[4] Stuart Jonathan Russell and Peter Norvig. Artificial intelligence: A modern approach. *Prentice Hall, Englewood Cliffs, NJ*, 2010, 2010.

[5] Mark Roberts, Swaroop Vattam, Ronald Alford, Beth Auslander, Justin Karneeb, Matthew Molineaux, Thomas Apker, Mark Wilson, James Mcmahon, and David Aha. Iterative goal refinement for robotics. URL https://www.researchgate.net/publication/301564088_Iterative_Goal_Refinement_for_Robotics.

[6] Robert M. Wygant. Clips — a powerful development and delivery expert system tool. *Computers & Industrial Engineering*, (1):546–549, 1989. URL https://www.sciencedirect.com/science/article/pii/0360835289901216.

[7] Tim Niemueller, Till Hofmann, and Gerhard Lakemeyer. Goal reasoning in the clips executive for integrated planning and execution. *Proceedings of the International Conference on Automated Planning and Scheduling*, 29:754–763, 2019. ISSN 2334-0843. URL https://ojs.aaai.org/index.php/ICAPS/article/view/3544.

[8] Tim Niemueller, Till Hofmann, and Gerhard Lakemeyer. Clips-based execution for pddl planners. *Proceedings of the 2nd Workshop on Integrated Planning, Acting and Execution (ICAPS intEx)*, 2018. URL https://kbsg.rwth-aachen.de/~hofmann/papers/clips-exec-pddl.pdf.

[9] Malik Ghallab, Adele Howe, Craig Knoblock, Drew McDermott, Ashwin Ram, Manuela Veloso, Daniel Weld, and David Wilkins. Pddl— the planning domain definition language. URL https://www.csee.umbc.edu/courses/671/fall12/hw/hw6/pddl1.2.pdf.

[10] Jay Powell, Matthew Molineaux, and David W. Aha. Active and interactive discovery of goal selection knowledge. URL https://www.aaai.org/ocs/index.php/flairs/flairs11/paper/viewpaper/2613.

[11] David Silver. Lectures on reinforcement learning, 2021-01-11. URL https://www.davidsilver.uk/teaching/.

[12] Richard S. Sutton and Andrew G. Barto. *Reinforcement Learning: An Introduction*. MIT Press, Cambridge, 2 edition, 2018. URL http://incompleteideas.net/book/the-book.html.

[13] OpenAI. Part 2: Kinds of rl algorithms — spinning up documentation, 2018. URL https://spinningup.openai.com/en/latest/spinningup/rl_intro2.html.

[14] Thomas Degris, Martha White, and Richard S. Sutton. *Off-Policy Actor-Critic*. 2013.

[15] Csaba Szepesvári. Algorithms for reinforcement learning. *Synthesis Lectures on Artificial Intelligence and Machine Learning*, 2009. URL https://sites.ualberta.ca/~szepesva/papers/RLAlgsInMDPs.pdf.

[16] Shengyi Huang and Santiago Ontañón. A closer look at invalid action masking in policy gradient algorithms. URL https://arxiv.org/pdf/2006.14171.pdf.

[17] Tim Niemueller, Daniel Ewert, Sebastian Reuter, Alexander Ferrein, Sabina Jeschke, and Gerhard Lakemeyer. Robocup logistics league sponsored by festo: A competitive factory automation testbed. In Sabina Jeschke, Ingrid Isenhardt, Frank Hees, and Klaus Henning, editors, *Automation, Communication and Cybernetics in Science and Engineering 2015/2016*, pages 605–618. Springer International Publishing, Cham, 2016. ISBN 978-3-319-42619-8. doi: 10.1007/978-3-319-42620-4_45. URL https://kbsg.rwth-aachen.de/papers/llsf-testbed-rc2013.pdf.

[18] RoboCup Federation. Robocup logistics league, 2016. URL https://www.robocup.org/leagues/17.

[19] Vincent Coelen, Till Hofmann, Tarik Viehmann, Tim Niemueller, Christian Deppe, Mostafa Gomaa, Ulrich Karras, Alain Rohr, Thomas Ulz, Lukas Knoflach, Kohout Peter, Sven Imhof, and Daniel Swoboda. Robocup robocup logistics league rules and regulations. URL https://github.com/robocup-logistics/rcll-rulebook.

[20] Till Hofmann, Sebastian Eltester, Tarik Viehmann, Nicolas Limpert, Victor Mataré, Alexander Ferrein, and Gerhard Lakemeyer. The carologistics robocup logistics team 2020. URL https://kbsg.rwth-aachen.de/~hofmann/papers/carologistics-2020-tdp.pdf.

[21] Tim Niemueller, Alexander Ferrein, Daniel Beck, and Gerhard Lakemeyer. Design principles of the component-based robot software framework fawkes. *in*

Proceedings of the International Conference on Simulation, Modeling, and Programming for Autonomous Robots (SIMPAR) (2010), 2010. doi: 10.1007/978-3-642-17319-6-29. URL https://kbsg.rwth-aachen.de/papers/fawkes-design-principles-simpar2010.pdf.

[22] Tim Niemueller, Daniel Ewert, Sebastian Reuter, Alexander Ferrein, Sabina Jeschke, and Gerhard Lakemeyer. Robocup logistics league sponsored by festo: A competitive factory automation testbed. In *Automation, Communication and Cybernetics in Science and Engineering 2015/2016*, pages 605–618. Springer, Cham, 2016. doi: 10.1007/978-3-319-42620-4_45. URL https://link.springer.com/chapter/10.1007/978-3-319-42620-4_45.

[23] Tim Niemueller, Sebastian Zug, Sven Schneider, and Ulrich Karras. Knowledge-based instrumentation and control for competitive industry-inspired robotic domains. *KI - Künstliche Intelligenz*, 30(3-4):289–299, 2016. ISSN 1610-1987. doi: 10.1007/s13218-016-0438-8.

[24] Andrea Dittadi, Thomas Bolander, and Ole Winther. Learning to plan from raw data in grid-based games. *GCAI*, (55):54–67, 2018. doi: 10.29007/s8jk.

[25] Volodymyr Mnih, Koray Kavukcuoglu, David Silver, Alex Graves, Ioannis Antonoglou, Daan Wierstra, and Martin Riedmiller. Playing atari with deep reinforcement learning. URL https://arxiv.org/pdf/1312.5602.pdf.

[26] Mustafa Mukadam, Akansel Cosgun, Alireza Nakhaei, and Kikuo Fujimura. Tactical-decision-making-for-lane-changing-with-deep-reinforcement-learning. *31st Conference on Neural Information Processing Systems (NIPS 2017), Long Beach, CA, USA.*, 2017. URL https://www.researchgate.net/profile/Mustafa-Mukadam/publication/333658431_Tactical_Decision_Making_for_Lane_Changing_with_Deep_Reinforcement_Learning/links/5cfaa3b5a6fdccd1308a5de1/Tactical-Decision-Making-for-Lane-Changing-with-Deep-Reinforcement-Learni
pdf.

[27] Kutluhan Erol, James Hendler, and Dana S. Nau. Htn planning: Complexity and expressivity. *AAAI*, (94):1123–1128, 1994.

[28] Jinning Li, Chen Tang, Masayoshi Tomizuka, and Wei Zhan. Hierarchical planning through goal-conditioned offline reinforcement learning. *IEEE Robotics and Automation Letters*, 7(4):10216–10223, 2022. doi: 10.1109/lra.2022.3190100.

[29] Benjamin Eysenbach, Ruslan Salakhutdinov, and Sergey Levine. Search on the replay buffer: Bridging planning and reinforcement learning. *arXiv e-prints*, page arXiv:1906.05253, 2019. doi: 10.48550/arXiv.1906.05253.

[30] David Hutchison, Takeo Kanade, Josef Kittler, Jon M. Kleinberg, Friedemann Mattern, John C. Mitchell, Moni Naor, Oscar Nierstrasz, C. Pandu Rangan, Bernhard Steffen,

Madhu Sudan, Demetri Terzopoulos, Doug Tygar, Moshe Y. Vardi, Gerhard Weikum, Sarah Jane Delany, and Santiago Ontañón, editors. *Case-Based Reasoning Research and Development*, volume 7969. Springer Berlin Heidelberg, Berlin, Heidelberg, 2013. ISBN 978-3-642-39055-5. doi: 10.1007/978-3-642-39056-2. URL https://link.springer.com/content/pdf/10.1007%2F978-3-642-39056-2.pdf.

[31] Suraj Nair and Chelsea Finn. *Hierarchical Foresight: Self-Supervised Learning of Long-Horizon Tasks via Visual Subgoal Generation*. arXiv, 2019. doi: 10.48550/arXiv.1909.05829.

[32] M. Wilson and David Aha. A goal reasoning model for autonomous underwater vehicles. URL https://sravya-kondrakunta.github.io/9thgoal-reasoning-workshop/papers/paper_7.pdf.

[33] Bruce L. Golden, Larry Levy, and Rakesh Vohra. The orienteering problem. *Naval Research Logistics (NRL)*, 34(3):307–318, 1987. ISSN 1520-6750. doi: 10.1002/1520-6750(198706)34:3⟨307::AID-NAV3220340302⟩3.0.CO;2-D.

[34] David Bonanno, Mark Roberts, Leslie Smith, and David W. Aha. *Selecting Subgoals using Deep Learning in Minecraft: A Preliminary Report: IJCAI Workshop on Deep Learning*. 2016.

[35] Alex Krizhevsky, Ilya Sutskever, and Geoffrey E. Hinton. Imagenet classification with deep convolutional neural networks. *Communications of the ACM*, 60(6):84–90, 2017. ISSN 0001-0782. doi: 10.1145/3065386.

[36] Carlos Núñez-Molina, Ignacio Vellido, Vladislav Nikolov-Vasilev, Raúl Pérez, and Juan Fdez-Olivares. A proposal to integrate deep q-learning with automated planning to improve the performance of a planning-based agent. In Enrique Alba, Gabriel Luque, Francisco Chicano, Carlos Cotta, David Camacho, Manuel Ojeda-Aciego, Susana Montes, Alicia Troncoso, José Riquelme, and Rodrigo Gil-Merino, editors, *Advances in Artificial Intelligence*, volume 12882, pages 23–32. Springer International Publishing, Cham, 2021. ISBN 978-3-030-85712-7. doi: 10.1007/978-3-030-85713-4_3.

[37] Jay Young and Nick Hawes. Evolutionary learning of goal priorities in a real-time strategy game. *Proceedings of the AAAI Conference on Artificial Intelligence and Interactive Digital Entertainment*, 8(1):87–92, 2012. ISSN 2334-0924. doi: 10.1609/aiide.v8i1.12503.

[38] Junkyu Lee, Michael Katz, Don Joven Agravante, Miao Liu, Tim Klinger, Murray Campbell, Gerald Tesauro, and Shirin Sohrabi. *AI Planning Annotation in Reinforcement Learning: Options and Beyond*. 2021. URL https://prl-theworkshop.github.io/prl2021/papers/prl2021_paper_36.pdf.

[39] A. Raffin, A. Hill, M. Ernestus, A. Gleave, A. Kanervisto, and N. and Dormann. Stable-baselines3 docs - reliable reinforcement learning implementations — stable baselines3 1.2.1a2 documentation, 2020. URL https://stable-baselines3.readthedocs.io/en/master/.

[40] John Schulman, Filip Wolski, Prafulla Dhariwal, Alec Radford, and Oleg Klimov. Proximal policy optimization algorithms. URL https://arxiv.org/pdf/1707.06347.

[41] Tom Silver and Rohan Chitnis. *PDDLGym: Gym Environments from PDDL Problems*. 2020.

[42] Timothy P. Lillicrap, Jonathan J. Hunt, Alexander Pritzel, Nicolas Heess, Tom Erez, Yuval Tassa, David Silver, and Daan Wierstra. *Continuous control with deep reinforcement learning*. 2015.

[43] Deepanshu Mehta. State-of-the-art reinforcement learning algorithms. *International Journal of Engineering Research and*, V8(12), 2020. ISSN 2278-0181. doi: 10.17577/ijertv8is120332.

[44] Volodymyr Mnih, Adrià Puigdomènech Badia, Mehdi Mirza, Alex Graves, Timothy P. Lillicrap, Tim Harley, David Silver, and Koray Kavukcuoglu. Asynchronous methods for deep reinforcement learning. *ICML*, 2016. URL https://arxiv.org/pdf/1602.01783.

[45] Brett Slatkin. *Effective Python (2nd Editiion): 90 SPECIFIC WAYS TO WRITE BETTER PYTHON*. Pearson Education, Inc., 2020. ISBN 978-0-13-485398-7. URL https://dl.mrinal.xyz/ebooks/Effective%20Python%20%282nd%20Editiion%29.pdf.

[46] Charles R. Harris, K. Jarrod Millman, Stéfan J. van der Walt, Ralf Gommers, Pauli Virtanen, David Cournapeau, Eric Wieser, Julian Taylor, Sebastian Berg, Nathaniel J. Smith, Robert Kern, Matti Picus, Stephan Hoyer, Marten H. van Kerkwijk, Matthew Brett, Allan Haldane, Jaime Fernández Del Río, Mark Wiebe, Pearu Peterson, Pierre Gérard-Marchant, Kevin Sheppard, Tyler Reddy, Warren Weckesser, Hameer Abbasi, Christoph Gohlke, and Travis E. Oliphant. Array programming with numpy. *Nature*, 585(7825):357–362, 2020. doi: 10.1038/s41586-020-2649-2.

[47] TensorFlow Developers. Tensorflow, 2023.

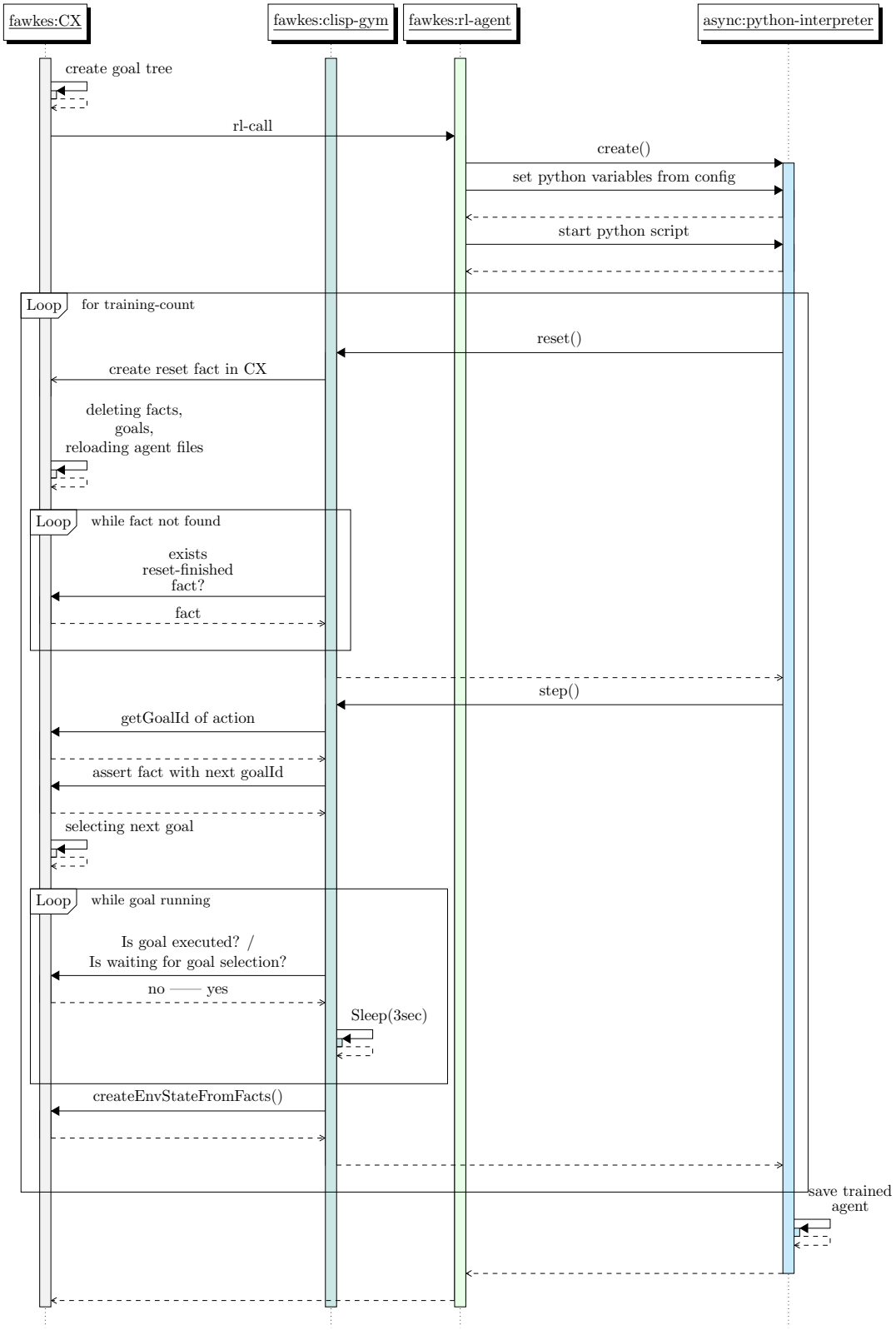
[48] Greg Brockman, Vicki Cheung, Ludwig Pettersson, Jonas Schneider, John Schulman, Jie Tang, and Wojciech Zaremba. Openai gym: A toolkit for developing and comparing reinforcement learning algorithms. URL https://github.com/openai/gym.

[49] 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 Artificial Intelligence*, pages 80–91. SCITEPRESS - Science and Technology Publications, 2021. ISBN 978-989-758-484-8. doi: 10.5220/0010252600800091. URL https://kbsg.rwth-aachen.de/~hofmann/papers/icaart21-goal-reasoning-rcll.pdf.

[50] Mathieu Seurin, Florian Strub, Philippe Preux, and Olivier Pietquin. Don't do what doesn't matter: Intrinsic motivation with action usefulness. In Maria Gini and Zhi-Hua Zhou, editors, *Proceedings of the Thirtieth International Joint Conference on Artificial Intelligence*, pages 2950–2956, California, 2021. International Joint Conferences on Artificial Intelligence Organization. ISBN 978-0-9992411-9-6. doi: 10.24963/ijcai.2021/406. URL `https://www.ijcai.org/proceedings/2021/0406.pdf`.

[51] Takuya Akiba, Shotaro Sano, Toshihiko Yanase, Takeru Ohta, and Masanori Koyama. Optuna. In Ankur Teredesai, editor, *Proceedings of the 25th ACM SIGKDD International Conference on Knowledge Discovery & Data Mining*, ACM Digital Library, pages 2623–2631, New York,NY,United States, 2019. Association for Computing Machinery. ISBN 9781450362016. doi: 10.1145/3292500.3330701.

Appendix


fawkes:CX
fawkes:clisp-gym
fawkes:rl-agent
async:python-interpreter
create goal tree
rl-call
create()
set python variables from config
start python script
Loop
for training-count
reset()
create reset fact in CX
deleting facts,
goals,
reloading agent files
Loop
while fact not found
exists
reset-finished
fact?
fact
step()
getGoalId of action
assert fact with next goalId
selecting next goal
Loop
while goal running
Is goal executed? /
Is waiting for goal selection?
no —— yes
Sleep(3sec)
createEnvStateFromFacts()
save trained
agent

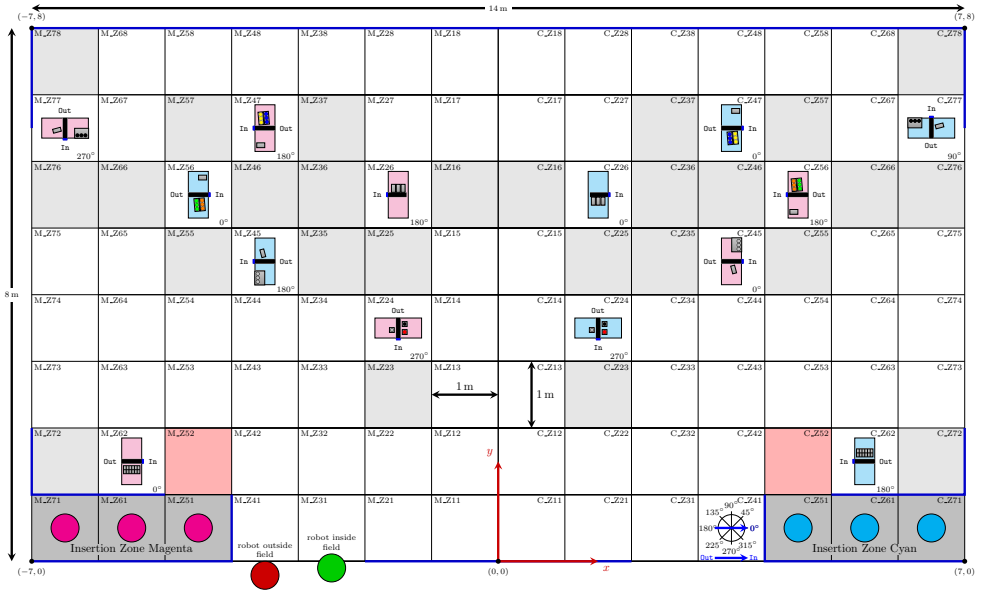


Figure 7.1: Visualization of the game field used for the experiments

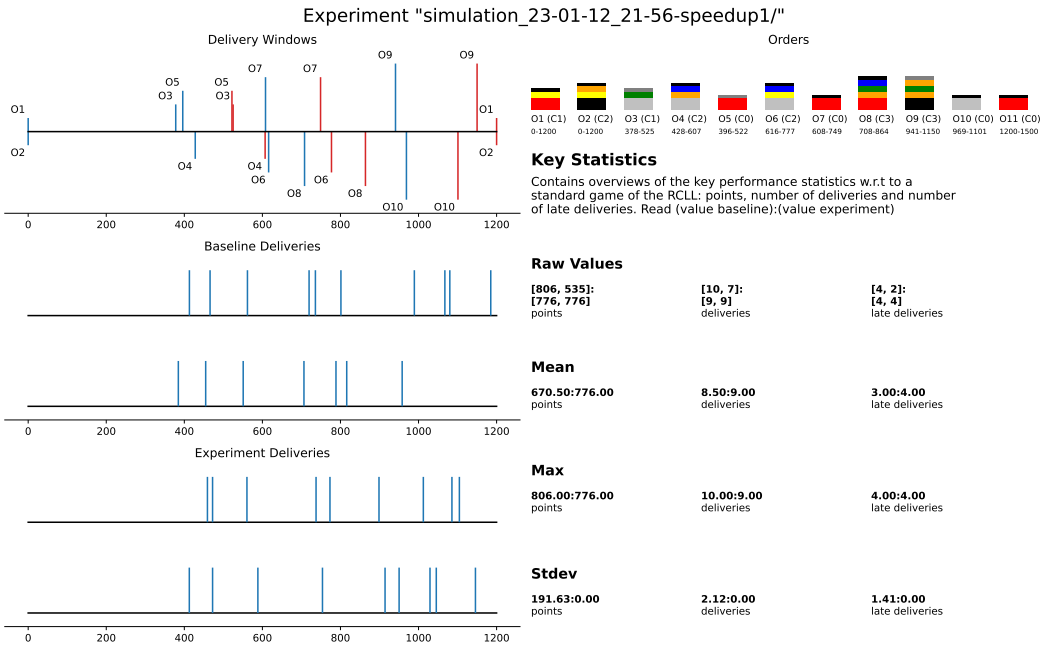


Figure 7.2: Four RCLL games played by the central agent with speedup 1

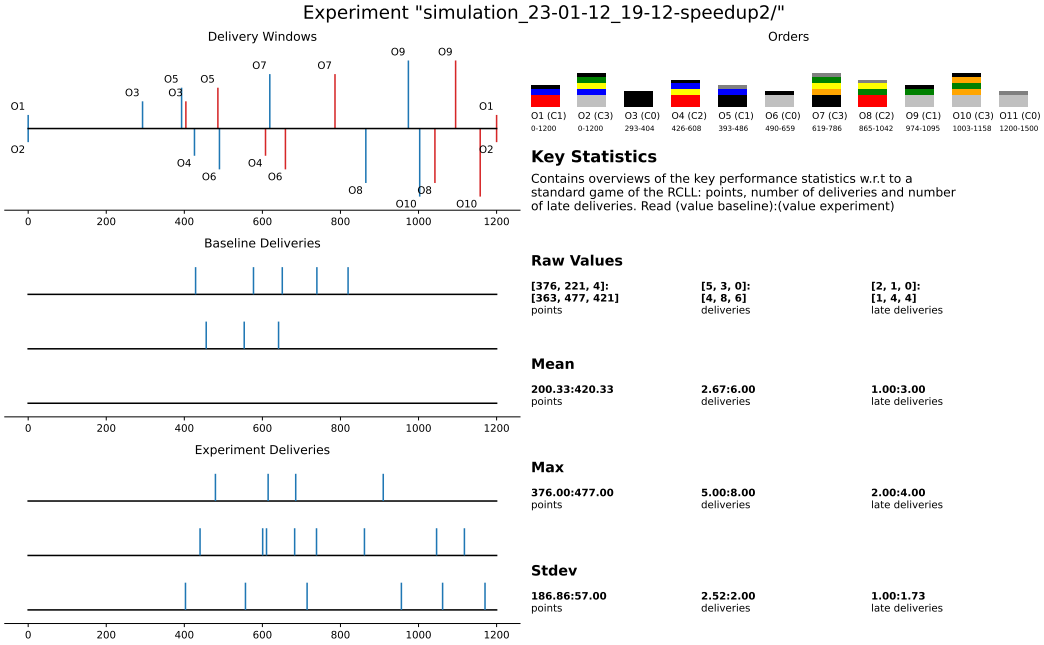


Figure 7.3: Six RCLL games played by the central agent with speedup 2

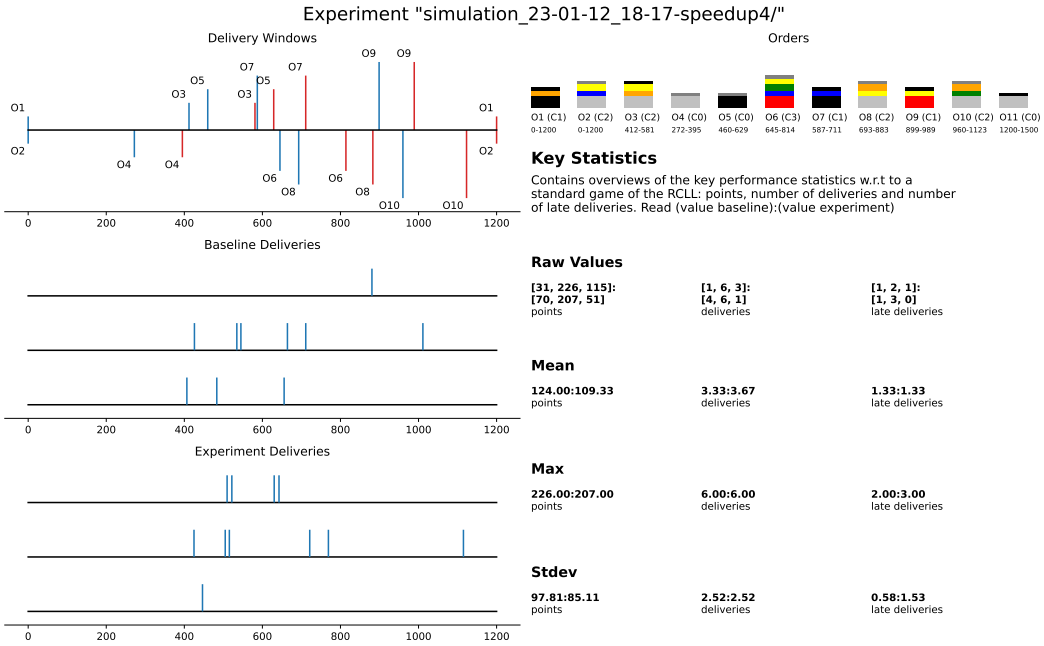


Figure 7.4: Six RCLL games played by the central agent with speedup 4

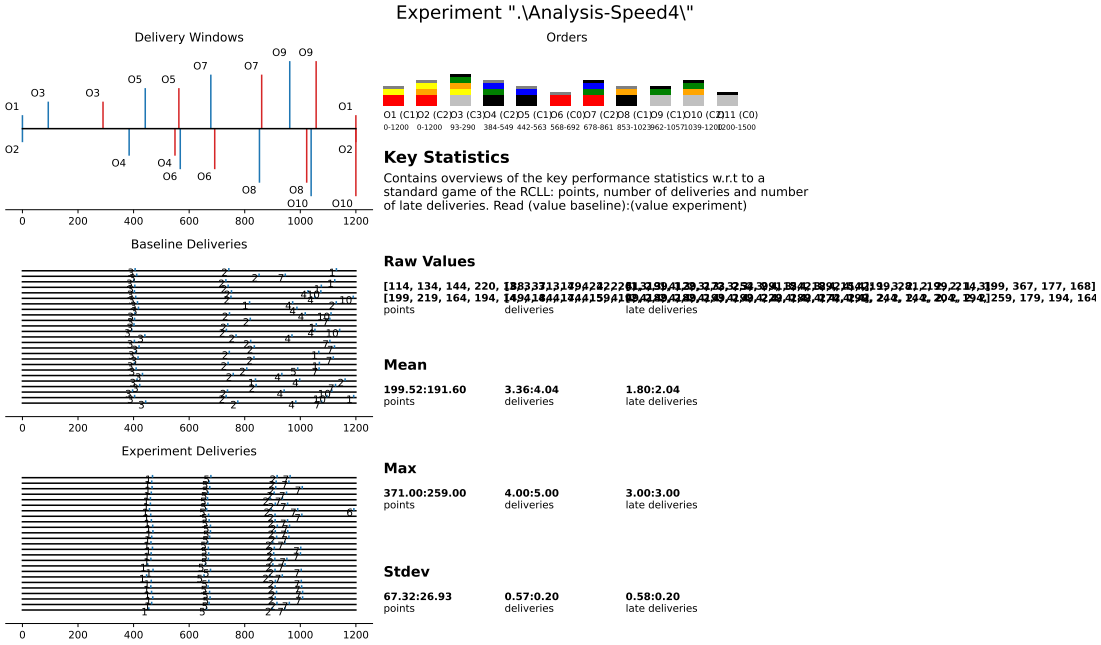


Figure 7.5: Six RCLL games played by the central agent with speedup 4

	Parameter	Value
RL Policy MaskableActorCriticPolicy	$learning\_rate$	0.0003
	discount factorγ	0.99
	gae_lambda	0.95
	ent_coef	0.0
	vf_coef	0.5
	max_grad_norm	0.5
	$batch\_size$	64
	policy function network	$[64x64]$ two hidden layers with the size of 64
	value function network	$[64x64]$ two hidden layers with the size of 64
RL Training MaskablePPO	Observation space size	270
	Action space size	39
	n_steps	3 number of steps after which a policy update is triggered
	$seed$	42
	$verbose$	1
	$max\_episodes$	10
	$total\_timesteps$	1200
	$save\_freq$	100 using a CheckpointCallback for saving the RL agent every 100 steps while training
Interface params clips-gym	env: speed	4.0 / 8.0
	step: max_time	60.0
	step: wait_time	6.0
	resetCX: max_time	120.0
	resetCX: wait_time	24.0
rl-agent-manager	rl-agent: active	true
	rl-agent: training-mode	true
RCLL params	execution-time-estimator:	
	static: speed	6 / 12
	navgraph: speed	3 / 6
	refbox: speed	2.0 / 4.0

Table 7.1: Extension of params