Rheinisch-Westfälische Technische Hochschule Aachen
Knowledge-based Systems Group
Prof. Gerhard Lakemeyer, Ph.D.

Bachelor's Thesis in Computer Science

Sharing Promises and Requests in Multi-Agent Goal Reasoning for Logistics Robots

Daniel M. Swoboda

RWTH Aachen University
September 2020

Advisor
Till Hofmann, M. Sc.

Supervisors
Prof. Gerhard Lakemeyer, Ph. D.
Prof. Dr. Matthias Jarke

Acknowledgements

I would like to thank my advisor Till Hofmann, who besides providing steady support and guidance, as well as insights and critical feedback during my work, also introduced me to the world of cognitive and logistics robotics at RWTH.

Furthermore, I want to thank Prof. Lakemeyer and Prof. Jarke for supervising this thesis and providing valuable feedback during the proposal presentation.

Additionally, I would like to thank Mostafa Gomaa and Tarik Viehmann for their continuous support and help, introducing me to the debugging processes and workflows in the Fawkes robotics framework and in the CLIPS Executive agent. The conversations over Club Mate in the KBSG lab were incredibly valuable to my work.

In the same sense, I also want to thank all members of Carologistics for their support and insights, that really helped me develop an understanding of the domain and the setup.

Finally, I want to thank my mother, grandfather, grandmother, my friends Leander Behr, Daryl Chiew, Daniel Honies, Jennifer Kim, David Mao, Maxwell McDonnell, Ben Rao, Chris Schnabl, Philip Trauner, and especially Frithjof Petrick for their continued support through my studies and life.

Contents

1 Introduction **7**

2 Background **9**
- 2.1 PDDL . . . 9
- 2.2 CLIPS . . . 9
- 2.3 Goal Reasoning . . . 10
- 2.4 CLIPS Executive . . . 11
- 2.5 Multi-Robot Coordination . . . 13
- 2.6 RoboCup Logistics League . . . 14

3 Related Work **16**
- 3.1 Information Sharing . . . 16
- 3.2 Task Assignment . . . 17
- 3.3 Ad-hoc Teamwork . . . 18
- 3.4 Resource Sharing . . . 19
- 3.5 Other Approaches . . . 19

4 Approach **20**
- 4.1 Semantics of Promises and Requests in Goal Reasoning . . . 20
- 4.2 Implementation in CLIPS . . . 25
- 4.3 Additional Modifications . . . 36

5 Evaluation **43**
- 5.1 Simulation Environment . . . 43
- 5.2 Experiments . . . 44

6 Conclusion **55**
- 6.1 Summary . . . 55
- 6.2 Future Work . . . 56

List of Figures

2.1 The goal lifecycle of the CLIPS Executive 12
2.2 Example production chain of a C2 product in the RCLL. 15

4.1 Required modifications of the CX for the integration of promises located at the modes of the goal lifecylce. 26
4.2 Resource handover between two agents following a request. 35
4.3 Slice of a game, showing the active goals of two agents. 37
4.4 Diagram of the modified goal trees. 38
4.5 Diagram of the goal tree restructuring process. 40

5.1 Scatterplot of the delivery times for a C0 product. 46
5.2 Game diagram of successful cooperation between agents to produce a C0 product. 47
5.3 Game diagram of unsuccessful cooperation between agents to produce a C0 product. 47
5.4 Scatterplot of the delivery times for a C1 product. 48
5.5 Scatterplot of the delivery times for a C2 product. 49
5.6 Scatterplot of the delivery times for a C3 product. 49
5.7 Scatterplots of delivery times for different products with complexity C0 and C3 in 50 full games. 52
5.8 Game graph of a full game highlighting cooperative behavior induced by promises. 54

List of Tables

5.1 The settings for the SkillerSimulation used in the evaluation for modified and unmodified agent. 44
5.2 Results of 20 games with a single C0 order for both the modified and base agent. 45
5.3 Results of 20 games with a single C1 order for both the modified and base agent. 46
5.4 Results of 20 games with a single C2 ordert for both the modified and base agent. 48
5.5 Results of 20 games with a single C3 order for both the modified and base agent. 48
5.6 Performance of baseline and modified agent over 50 games. 51
5.7 Delivery numbers of each possible product for both agents in 50 games. . . 53

List of Listings

4.1 PDDL-action for delivery station preparation. 27
4.2 CLIPS `domain` fact for machine type. 28
4.3 CLIPS `promise` fact for machine-type. 28
4.4 CLIPS `projected-promise` fact for machine-type. 28
4.5 CLIPS rule for a `projected-promise` assertion. 29
4.6 A CLIPS rule that asserts a `projected-promise` for a `promise` fact. . . . 30
4.7 A CLIPS rule that retracts a `projected-promise` for a negative promise-fact. 30
4.8 A CLIPS rule that retracts a `projected-promise` if there is no supporting promise. 30
4.9 Example CLIPS rule for goal formulation. 31
4.10 Example CLIPS rule for goal formulation using `projected-promise` facts. 32
4.11 Structure of a `promise-resource` facts. 33
4.12 Example of modified preconditions matching a `projected-promise` and a `promise-resource` 34
4.13 The structure of a resource request fact. 34
4.14 Adapted `projected-promise` with integrated timing 41
4.15 Example of a timing-based rejection mechanism for promised goals. 42

1 Introduction

Intelligent robots and the services built around them are becoming more common parts of our lives. From production and logistics to the service industry and home robotics, international demand in 2018 was estimated to be almost 8 times as large as in 2002 with more than $400,000$ units shipped [12]. Industrial processes are becoming more digitalized and automated to increase efficiency whilst reducing costs. The connected production systems commonly referred to as Industry 4.0, Smart Factory or Internet of Production require a new approach to logistics in order to cope with increasing demand for flexible manufacturing and highly individualized products. Robotic logistics can play a role in building these modern factories by filling the gaps left by traditional approaches and providing flexible point-to-point on-demand delivery [32, 2].

The RoboCup Logistics League (RCLL) is a competition focusing on the advancement of autonomous and smart logistics robots for Smart Factory settings. The robots are tasked with on-demand assembly of workpieces of differing complexity, requiring the transport of material between production machines. To maximize productivity given constraints of time and dynamic changes in the environment (e.g. machine failures, changing orders), the robots have to implement robust navigation, perception, manipulation and reasoning. Additionally, efficient cooperation of the agents, forming a so called Multi-Robot System (MRS), can lead to further increases in their productivity [34]. All of these factors are relevant in the development of smart factories. However, one of the major challenges is the coordination and execution of strategies between the robots.

Goal Reasoning (GR) is a method for intelligent agents to handle high-level goals in a way that is both reactive to the environment and flexible [1]. Instead of merely reasoning about its actions to reach a certain, fixed goal, a GR agent also reasons about the goals it pursues, modifying the goal or even switching to an entirely new one, if the circumstances call for it [15]. Current GR agents employ a goal refinement strategy called the *goal lifecycle*. It defines several modes, which a goal might reach over its lifetime. Furthermore, the lifecycle specifies how a goal progresses over time and which constraints are to be met before a goal can transition.

The CLIPS Executive (CX) is a multi-robot goal reasoning agent framework that uses a distributed and incremental approach with granular goals, that has been used in the RCLL [15]. It provides the basic techniques necessary to develop a multi-agent GR system, including planner integration, ways to express goals and goal trees and coordination concepts like mutexes and a shared world-model database with shared resources. In the RCLL domain the terms incremental, granular and distributed translate to an approach where the production of a workpiece is split into several smaller incremental steps, each of which can be performed by a different robot. This approach proved to be state-of-the-

art in various competitive settings [16]. Still, the coordination and cooperation concepts in the CX are limited to world-model sharing and conflict avoidance using the provided locking mechanism.

In this thesis, we present two novel extensions to the goal reasoning mechanism in the CX that provide additional possibilities for multi-robot coordination and cooperation, which we call *promises* and *requests*. We use the term promise to describe a form of intention sharing, wherein each agent shares the expected outcome of their current goals. When this information is incorporated into the world model, agents can reason over a near-future world-state and potentially formulate higher-value goals, whose preconditions are not met given only the current state. With requests, we describe two forms of multi-robot-coordination that are akin to task-assignment and resource-sharing approaches. They are techniques through which an agent actively requests a service from another agent, in our case the fulfillment of a goal or the ownership of a resource. We focus on resource requests, which coordinate co-occupation and handover of locked resources. Resource requests are used primarily as a method to make promises more effective, allowing goals to be executed before all resources are acquired. Additionally, we present two adaptions to the CX framework that aid promises: a timing feature integrated in promises, that estimates the execution time of a goal, which we use to reject infeasible promised goals and a modified goal-tree with dynamic restructuring, increasing the lifetime of promises.

The thesis is structured as follows: In Chapter 2 we briefly introduce the Planning Domain Definition Language (PDDL), the C-Language Integrated Production System (CLIPS) and the goal reasoning concept, which are the preliminaries for the CLIPS Executive and our extensions. We then give a brief overview of the multi-robot coordination (MRC) problem and how it is solved in the CX. Finally, we introduce the RoboCup Logistics League (RCLL), which is our evaluation domain. In Chapter 3, to categorize our own approach, we introduce Related Work in the form of a selection of different MRC techniques that have been successfully used in the past. Chapter 4 presents promises and requests in detail, first with a formal definition, then by introducing the specific implementations as extensions of the CX. Also included are detailed descriptions of the adapted goal-tree and the timing feature. In Chapter 5 we present a detailed evaluation of promises and requests in the RCLL domain. We used a simulated RCLL as our testing environment and present the experiments that we conducted to evaluate the effects of promises and resource requests on agent behavior. We highlight how the extended agent performs in comparison to the unmodified agent in different scenarios and evaluate the results. Finally, in Chapter 6, we conclude the thesis with a summary and outlook.

2 Background

In logistics applications, the change to modern, versatile production floors requires the use of flexible robotic transportation systems to keep up with dynamically changing requirements [32]. While single-robot systems are considered to be easier to implement, multi-robot systems have advantages in these settings by providing good spatial distribution, robustness, reliability and flexibility, all of which are considered to be desirable in logistics [34]. To perform tasks in a cooperative manner, coordination between the robot agents is required, the study of which is called multi-robot coordination. In addition to coordination, the agents must also have robust planning, goal execution and execution monitoring [15].

The CLIPS Executive (CX) [24] is a framework for creating agent software built on the principles of goal reasoning and is suitable for usage in a MRS [16]. It integrates planning based on PDDL [22] with task execution and monitoring based on the CLIPS [33] expert system, whilst providing basic robot coordination functionality [24, 16].

2.1 PDDL

Planning can be described as the problem of determining a sequence of actions to reach a certain world-state [26]. Commonly, actions are represented as a set of preconditions and effects (in the form of FOL literals) while the world-state is represented by first order atoms. The Planning Domain Definition Language (PDDL) is a universal planning language used to encode planning tasks, providing an interface that is independent of the actual planner software, allowing for better interchangeability [24]. It separates the problem into two parts, a domain description and a problem description [22]:

- **Domain Description:** defines the domain for which to be planned for, this includes:
  - *Predicates:* Properties of the objects in the domain (either *true* or *false*).
  - *Actions:* Ways of changing the world, including pre-conditions and post-conditions.
- **Problem Description:** the specific problem for which a course of action needs to be planned for. This includes the objects to interact with, the initial state and a description of the desired goal state.

2.2 CLIPS

CLIPS is a rule-based production system. It consists of three main building blocks [24]:

- **Fact Base:** Represents the working memory of the software agent. It stores basic pieces of information in the form of a structured type known as facts, e.g., the information, that a new item is ordered, or that a machine is in the ready state.
- **Knowledge Base:** Includes two types of knowledge:
  - *Procedural Knowledge:* Functions that can be executed by the agent. These can be updates to the Fact Base, functions in the CLIPS language or C++ functions that might interact with other components of a larger system to implement additional functionality, e.g., the function controlling the robot to feed the production machine to fulfill part of an order or an update to the fact base, signalling that the machine is now busy.
  - *Rules:* Rules have a left-hand-side (LHS), which is a set of preconditions and right-hand-side (RHS), where procedures are executed. Preconditions are patterns for desired facts or their negations. If a matching fact exists in the fact base (or no matching fact in the case of a negation), the rule is considered to have “fired“ and thus is added to an agenda, which contains all the activated rules scheduled for execution. Rules from the agenda are subsequently executed by executing the corresponding procedural knowledge, e.g., if the production machine is ready and an order exists, feed it with the required input material.
- **Inference Engine:** Combines working memory and knowledge base in a cycle of fact updates, rule activation and agenda execution that is performed until stability is reached, i.e. no more rules are activated. E.g., a production machine is ready and a new order is created. This knowledge triggers the rule that calls a function to feed the machine and updates to the fact base to reflect that the machine is now busy. Once the called function finishes, the result is used to update the fact base again which might trigger additional rules that control the further production steps until the final product is created.

Based on this system a form of conditional behavior control for intelligent systems can be implemented, as highlighted by the order fulfillment example above. Unlike imperative programming, the program flow is not static and linear but rather dynamic depending on the fact-base and the accordingly activated rules. This way, we can focus on the situations and their corresponding actions, while the inference engine is taking care of the executive part and program flow. This allows for easier adaption of the procedures and better maintainability.

2.3 Goal Reasoning

Motivated by the human behavior of dynamic goal reprioritization, a similar concept for autonomous agents known as goal reasoning (GR) is defined by Aha the following way [1]:

Definition 1.
Goal reasoning is the study of agents that can deliberate on and self-select their objectives, which is a desirable capability for some applications of deliberative autonomy.

It represents a process conducted by the agent, in which a goal lives through a set of different stages during its lifetime, defined as the goal lifecycle [1]. In any of the phases the pursued goal might change given the internal conditions of the agent and external, environmental conditions such as the data collected from exteroceptive sensors. This approach has been studied extensively under different terminologies for several decades, in an attempt to improve the behavior of intelligent agents, especially in unknown conditions, removing the need for a model for all situations to be encountered. Accordingly, GR agents are most appropriate in complex environments.

Several models of GR processes like Goal-Driven Autonomy (GDA) and goal refinement exist [1]. Goal refinement is considered the more extensive and capable one of these two, being able to model more complex processes and goal constraints. It approaches GR through iterative refinement of goals in the form of a lifecycle by adding constraints, while at the same time providing different strategies for the handling of events and changes during the reasoning process. This multi-stage approach provides a formal structure for agent implementation to enable deliberation and goal adaption.

2.4 CLIPS Executive

The CX is a CLIPS-based agent that implements a goal reasoning mechanism based on the goal refinement approach [25, 1]. Goals represent the core data structure, supplemented by CLIPS-language rules that implement the control of a goal's transformation between goal modes.

The world-model is represented in the form of a PDDL domain description, shared amongst both the executive (which uses CLIPS) and the planner (which uses PDDL) [24]. A parser is used to translate the predicates and actions for the CLIPS environment, where predicates carrying information about the world-state, are translated into CLIPS-facts. If the CX is used in a multi-agent setup, world-facts can be shared using a common MongoDB database between the agents, such that they can reason over the same world-state. Goals are pre-defined for each domain and stored in the CLIPS environment. Each goal comes with a set of preconditions, resources and other meta information like goal-type and goal-parent.

Goals are arranged in trees with leaf nodes representing simple goals that can be achieved by executing a certain set of actions, as determined by the planner via the PDDL interface [25, 16]. The planner component can be exchanged for a plan-database or plan-library, which store predetermined plans for each action. Compound goals are trees of goals consisting of several subgoals, revealed through their expansion. They are iteratively expanded until all simple goals associated with them are achieved. Each goal instance specifies all the relevant aspects required by the executive to achieve the goal, including its required resources and the goal mode, through which a goal's current stage

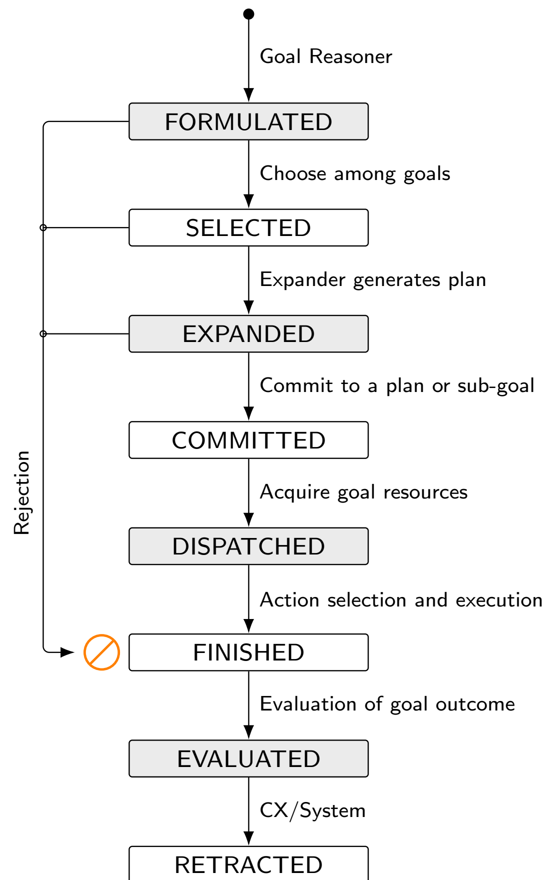


Figure 2.1: The stages, transitions and resolution strategies for goals in the goal refinement based goal lifecycle employed in the CX. A goal enters the lifecycle in formulation and then progresses through specified actions between each stage. Based on Niemueller et al. [25].

in the lifecycle is marked [25]. This mechanism is used by a set of CLIPS rules to guide the transition of the goals between modes according to the goal refinement principle, leading to their eventual fulfillment.

Figure 2.1 depicts the different stages of refinement in the GR model used in the CX. Typically, the agent, when reasoning over actions to perform, would perform the following steps [25]:

Formulated create new goal(s) that are possible to reach, given the actions available to the agent and the current world state. These goals now enter the goal lifecycle.

Selected choose goal(s) to actively pursue, based on set criteria (e.g. resources available, cost metrics, etc.).

Expanded expand goal(s) into subgoal(s) or generate plan(s) for the goal(s) if atomic (i.e. no subgoals).

Committed pick one expansion, acquire resources.

Dispatched execute the committed expansion.

Finished return from plan execution.

Evaluated based on goal specific criteria, evaluate the goal execution and apply changes to the world model.

Retracted remove the goal, free resources acquired.

The more advanced a goal's mode is in respect to the goal lifecycle, the more constraints must be fulfilled for it to be considered. Through this, the agent can assert that the necessary conditions are met step-by-step, considering a greater variety at the beginning, and narrowing down the selection based on the external circumstances. A goal may be rejected before committing to it, based on available resources [25].

To fulfill a simple goal, agents need to perform both execution and planning [24]. Traditionally, it is not uncommon for executives to treat planning software as black-boxes. The CX integrates the planner with the executive using a shared world model which is defined in the planning language PDDL. With this setup any planning software that supports PDDL can be used in combination with the executive. Similar to the common notion, the executive remains the top-most controller, while the planner is called on-demand to generate plans for tasks to be achieved.

2.5 Multi-Robot Coordination

Because of shared resources, logistics MRS need to perform multi-robot coordination (MRC) between the agents to achieve effective collaboration and avoid potentially arising conflicts. Various forms of MRC exist, however, the goal is always to achieve increased robustness, higher quality solutions and faster task completion [7]. Yet, effective coordination remains a big obstacle in the development of MRS, due to dynamic events and limits in communication and mobility, among other reasons.

Coordination approaches are often tailored to the problems of the specific kind of MRS solution employed [34]. MRS can be split into homogenous and heterogenous MRS from a hardware perspective, where homogenous systems make use of only one type of robot. Similarly, they can also be split into cooperative or competitive MRS from a behavior perspective, with the agents of a cooperative system having to work together to increase the utility of the entire system.

In logistics, it is common to have a homogenous and cooperative setup. Based on this, two types of conflicts requiring coordination are most likely to occur [34]: *resource conflicts* and *space conflicts*. Resource conflicts describe situations in which multiple agents need to have access to the same resource, e.g. a production machine. Space conflicts are those situations in which multiple agents want to access the same space at the same time, leading to a colision. It can be argued that a space conflict is a resource

conflict itself. Solutions to these coordination problems can be in the form of predefined protocols, communication, multi-robot planning (focusing on task assignment) or centralized reasoning.

2.5.1 MRC in the CX

The CX supports makes use of a distributed, incremental and granular reasoning strategy. This means that fine-grained goals are used, giving each agent the autonomy to decide locally which goal to pursue next. However, because of limited resources in the world the CX must still provide MRC functionality to prevent the goals that are dispatched on each agent to interfere with one another. A shared database is used to synchronize world-model facts, enabling the following two coordination methods [15, 25]:

- **Locking Actions:** Using a mutex functionality, two actions are implemented: lock and unlock. These actions temporarily grant access to objects by acquiring the mutex, or revoking exclusive access by releasing the mutex. Similarly location-based locks are used to acquire exclusive access to locations.
- **Goal Resource Allocation:** Resources are tied to goals, in order to exclude interfering goals from being simultaneously executed. Mutexes are used to acquire exclusive access to the necessary resources which are then held during the entirety of the execution. Mutexes are requested during the commit phase, meaning that a goal is only dispatched in the case that it acquires all locks successfully. Resources are only released after the finish state has been reached.

Goal resource allocation and the locking actions require cross-agent mutexes to be implemented. In the CX this is achieved using facts in the shared world-model to exchange the necessary information. Mutexes can be acquired using a majority acknowledgement principle. This means that the world-model update that would assign a mutex must be agreed upon by the majority of agents in the environment.

2.6 RoboCup Logistics League

The RoboCup Logisitcs League is part of RoboCup, a series of international robotics competitions aiming to foster AI and intelligent robotics research by providing a benchmark [15]. Originally known for the robotics soccer competitions, it now includes competitions for service robots, rescue missions, logistics (in the form of the RCLL) and junior competitions for student enthusiasts.

RCLL is intended to simulate in-factory logistics in a smart factory setting with dynamic, on-demand orders [15]. Two teams share the same production floor of 14m $\times$ 8m for 17 minutes. Each team uses three robots and a unique set of production machines, called Modular Production Systems (MPS). While the machines are exclusive to each team, their location on the production floor is randomly generated, leading to frequent path crossings between the robots of the two teams, requiring collision detection and

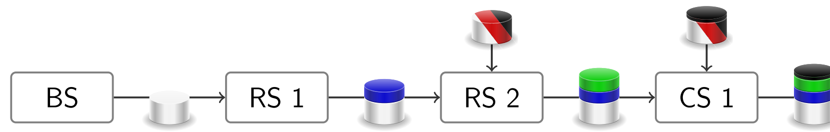


Figure 2.2: A possible production chain for a C2 product. A base is taken from the cap station, fed into two different ring stations, and finally put into the cap station where the cap is mounted. Based on Hofmann et al. [15].

avoidance mechanisms. Uncertainty is generated through unscheduled machine maintenance, unexpected machine behavior due to errors, agents failing and orders being generated at random.

Orders are generated from a centralized, semi-autonomous referee system which also takes care of the score keeping process [15]. The products that can be ordered are modular workpieces consisting of a base, optional rings, and a cap. They are of different complexities: C0 (base+cap), C1 (base+[1 ring]+cap), C2 (base+[2 rings]+cap), C3 (base+[3 rings]+cap), with multiple color options for each of the elements. Accordingly the production steps are: collect base, mount ring(s), mount cap, deliver; with a machine for each of these steps. The robots have to transport the results from each station to the input of the next station in the correct order to create the desired product. Additionally, they also need to prepare the machines to perform these operations. This includes feeding a cap of the desired color into one of the cap stations and feeding additional bases into the ring stations as payment before certain ring colors can be mounted onto a workpiece. Figure 2.2 highlights a possible production chain for a C2 product. First the base is collected from the base station (BS). After the base is fed into ring station 1 (RS1), the first ring is mounted, for which no payment was necessary. For the next ring, a payment of one base (of any color) is required, which is fed into the ring station 2 (RS2), followed by the workpiece. After the second ring is mounted, the product now needs to be outfitted with a cap. Cap station 1 (CS1) is fed with a cap carrier, which contains the cap of the right color. The cap station is takes the cap from the carrier, and the robot can feed the workpiece into the cap station, which mounts the cap, finishing the production cycle.

Over-subscription is one of the main challenges besides workpiece transportation, collision avoidance, sequencing of production steps and coordination betwee the agents [15]. It results from a higher amount of orders being generated in the game, than could possibly be fulfilled, which requires the agents to reason about which orders to pursue based on the reward it might give and other constraints like time left and delivery windows.

GR with the CX was successfully used to implement a distributed, multi-agent goal reasoning approach for the 2019 RCLL competition. The chosen approach made use of a pre-computed plan database using the same PDDL interface that would normally be used for an on-line planner. Through this measure, runtime overhead can be avoided since the scenario of the RCLL is narrow enough to support a plan library. Goal resource allocation was used as the main way of coordination between the agents.

3 Related Work

In the development of MRS, the coordination between the agents plays an extensive role [34]. We introduce existing approaches for MRC and roughly classify them based on perceived core properties in this section. However, the complex nature of multi-robot systems often leads to combined approaches that fit into multiple categories.

Existing literature identifies several layers for multi-robot coordination [9], including task allocation [21] and perception [15, 28]. The coordination within these layers is achieved using a variety of different techniques, such as market approaches (mostly for task-allocation) [7] and communicative approaches (mostly for perceptive information) [15].

3.1 Information Sharing

One set of approaches found in literature is built on the communication of world states or perceptive inputs to the other agents, allowing them to incooperate this information into their own planning. These approaches include those focused on sharing agent intentions [17, 11, 27, 29], and those focused on sharing their world perception [16].

3.1.1 Intention Sharing

Intention sharing approaches have in common, that agents communicate their intended actions to the rest of the robots in the MRS. They might provide this information just as a means of allowing other agents in the same MRS to accommodate for their actions. In other cases, intentions are used to coordinate common actions amongst the agents. Holvoet and Valckenaers [17] use the marking of shared resources to communicate intentions and coordinated exclusive access. Grant et al. [11] introduce a logic to formulate intentions that is used to build sub-tasks for effective coordination. Sarratt and Jhala [29] introduce the concept of sharing intentions to allow cooperation with unknown teammates.

3.1.2 World Model Sharing

World model sharing approaches achieve coordination by actively sharing their perception of the world fully or in parts, allowing other agents to incorporate this information. One such example would be Hofmann et al. [16] with the CX for the RCLL. Here, a shared world model is used to establish a distributed locking mechanism as well as to exchange crucial information on the world state, e.g. machine state or workpiece position.

3.2 Task Assignment

Another approach to coordination in multi-robot systems is task assignment, or task allocation [21]. Compared to the communicative approaches, that are meant to share parts of the inner processing of an agent, task assignment applies a different perspective. In this approach the tasks that need to be performed to achieve the goals of the system, are assigned to the agents directly and explicitly, giving each agent a unique role.

Tasks can be narrow, avoiding most forms of coordination (elemental or simple tasks); can be cooperative, requiring agents to perform sub-tasks of a bigger tasks simultaneously (compound or complex tasks); or can be vague definitions of roles that the agents try to fulfill according to certain constraints but with no explicit assignment on a task level [7, 21, 18]. Task assignment can be achieved centrally or decentralized, with mixed forms like sub-contracting, where one agent assigns parts of his own task to other agents, as an example of locally centralized approaches [21].

3.2.1 Organizational Structuring

Organizational structuring for task assignment usually relies on a centralized approach, with a master coordinating the other agents [30]. It's the master's task, having full autonomy on the task assignment, to split the main objective into sub-tasks and assign them to the agents directly. In this setting, the executing agents are mere workers that fulfill the given task. It is considered the simplest coordinating approach for MRS.

3.2.2 Sub-contracting

Sub-contracting is a form of locally centralized task-allocation. Grant et al. [11] describe a logic for intentions that explicitly models sub-contracting, meaning that one agent needs to assign parts of its tasks to other agents in order to fulfill the main objective. Sub-contracting shares similarities with centralized approaches for task assignment. However, it can occur recursively, meaning that agents that took over sub-tasks can further split them and allocate them to different agents. [30]

3.2.3 Market-based Approaches

Market-based multi-robot coordination is based on the idea of self-interested individuals trying to maximize their own profit in a market setting [7]. Usually a team is given an objective that can not be achieved by a single agent in a sufficient manner. However, these objectives can be decomposed into sub-tasks that can be achieved by the agents. Each robot features a unique utility function measuring the costs of its resource usage when trying to fulfill a task, based on its implementation and hardware constraints.

To assign the tasks, auctions are most commonly used [7]. In those, the coordinating agent announces a task or a set of tasks and the agents submit their corresponding costs based on their utility function. The agent with the lowest cost will usually be assigned the task. However, there are some variants that make use of task packages and other

optimization methods to further increase productivity. Coordination using auctions can also occur on the planning level, Hertle et al. [13] who use temporal planning to first create an abstract plan, the parts of which are then assigned to agents in an auction process, after which they perform the detailed planning locally.

3.2.4 Competition-based Approaches

Competition-based approaches are similar in nature to market-based multi-robot coordination, with slight differences in the actual allocation methods. Jin et al. [19] describe an approach that is based on control theory, where one task is assigned to one agent without a central coordinator. In their setting a group of robots is tasked with tracking a robot that is not a member of that group. Similar to hunting behavior only the fittest agent (i.e. the one closest to the target) is tasked with moving towards the target. The fitness of the agents is calculated in a distributed manner, with each agent calculating all scores.

3.2.5 Role Assignment

Role assignment approaches like Iocchi et al. [18] and Vail et al. [31] are focusing on assigning more general role descriptions instead of detailed tasks. The agents then need to pick tasks contributing to the overarching goal of their role in the context of the MRS. These approaches split the overall objective into main roles needed to fulfill it, e.g., a soccer team that requires several positions (like goalkeeper, attacker, defender) to be successful. It can assign the roles efficiently to the agents based on, e.g. utility functions, incorporating their positions and physical capabilities (if the system is heterogenous). The agents on the other hand can make use of agent autonomy over their specified role, enabling distinct implementations of role behavior.

3.3 Ad-hoc Teamwork

Ad-hoc teamwork refers to agents that only share limited information, or might be completely unkown to each other, working together to achieve a common objective. Agents performing ad-hoc teamwork must sufficiently interpret the behavior of teammates and adapt their own in order to achieve the common goal [4, 20].

3.3.1 Behavior Prediction

When encountering unseen teammates, cooperation is impeded by a number of facts. Agents do not have a model of the agent encountered [4] or might not even know how to communicate directly [29]. One approach to solve this, is to use the models of seen agents, acquired through forms of machine learning, to predict the behavior of the teammates through observed behavior. This leads to hybrid models generated from the existing models, tailored to the observed behavior of the unknown teammate. A result of this approach is that the teamwork improves over time as more agent behavior is observed.

Examples for these approaches are [4, 3], in which a group of robot agents is tasked with capturing a prey robot. Robots plan locally and need to coordinate their movement to avoid collisions, they use models of the other agents to anticipate teammate behavior in their own planning.

3.4 Resource Sharing

Another form of coordination uses access limitations to resources to coordinate behavior of agents in a MRS to avoid congestion and inefficient waiting times [8, 25]. Common amongst the introduced approaches for coordinated resource access is the use of local planning.

3.4.1 One-time Assignment

Dolgov et al. [8] introduce a system based on Markov decision processes (MDP) for one-time resource assignment. In this system, the resources are only free during the initialization of the system and then assigned to the respective agents for the entirety of operation. The coordinating agent requests the value of a resource bundle from each agent, assuming optimal policy can be followed and assigns resource bundles to the agents in a way that maximizes the global utility towards reaching the objective.

3.4.2 Locking

Exclusive access to resources can also be used as a form of coordination. Both spatial blocking and waiting for resources can be mitigated using locking mechanisms. Niemueller et al. [25] describe a democratic locking mechanism, based on majority agreement with local planning. The majority agreement avoids situations in which egoistic agent behavior disrupts the operation of the other agents, while the locking mechanism avoids congestion. Agents coordinate democratically which resources to access, and tasks are rejected if exclusive access can not be acquired.

3.5 Other Approaches

The introduced approaches only give a limited view on the entirety of the large field of MRC. We summarized related work on MRC with a particular focus on attributes that are shared with the newly proposed methods of promises and requests. Other approaches to MRC include control theoretic approaches as presented by [5], hybrid approaches using more than one method [25], or plan sharing [23].

4 Approach

The aim of this thesis is to integrate two forms of multi-agent cooperation and coordination into an existing goal reasoning agent in the context of logistics robotics, with specific application in the RCLL domain. We want to extend the CX with a combination of intention and resource sharing mechanisms, which we refer to as promises and requests, based on their similarities with human interactions.

The general concept of promises and requests that we suggest encompasses two communication techniques between the agents of a multi-agent system. Promises are meant to communicate the intent of an agent to all agents in the system in the form of goal effects, while requests represent a form of coordination in which one agent works for the other agent. We will see that, promises are related to the information sharing approaches introduced in Section 3.1, while requests are akin to task assignment (3.2) and resource sharing (3.4) approaches.

With our implementation as an extension of the CLIPS Executive, we employ the techniques in more specific use-cases within the RCLL domain. In the context of the CX a promise is the effect of an action that was not executed yet, while requests are limited to a special form we call resource requests, which refers to a method for co-occupation of symbolic resources and subsequential handover.

4.1 Semantics of Promises and Requests in Goal Reasoning

First, we want to define the semantics of promises and requests in the form of a formal model in the context of multi-agent goal reasoning. In the model we present, they are a powerful, yet simple extension to the existing reasoning mechanism. Since some features that we need are reasonable to describe in the CX's specific variety of GR, we built the extension on a model of it.

4.1.1 Goal Reasoning Semantics in the CX

In the CX every goal is associated with a set of preconditions at each of the stages within the goal lifecycle. Additionally goals have one or multiple possible plans, consisting of several actions, of which each has its own set of preconditions. We use a GR formalism akin to the one defined by Aha [1] and Cox [6] adapted to the CX's approach to describe the environment and goal formulation in a world-state oriented view with:

- D, the finite set of propositions that we use to represent the world-state. We reason over D under the closed-world assumption.

- $S := \mathbb{P}(D)$, the set of possible states.
- A, a set of actions $a = (c^+, c^-, e^+, e^-)$ that can be performed in the world. Actions are, similar to STRIPS operators [10], quadruples of finite sets of positive and negative preconditions as well as positive and negative effects, with $c^+ \subseteq D$, $c^- \subseteq D$, $e^+ \subseteq D$ and $e^- \subseteq D$. For the preconditions to be satisfied in situation s, it most hold that: $s \models \bigwedge_{p \in c^+} p$ and $s \models \bigwedge_{p \in c^-} \neg p$, where $\models$ refers to entailment under the CWA.
- E, a set of exogenous actions $E \subseteq A$.
- $\text{Del} : A \rightarrow \mathbb{P}(D)$, the function that returns the negative effects of an action or an exogenous effect.
- $\text{Add} : A \rightarrow \mathbb{P}(D)$, the function that returns the positive effects of an action or an exogenous effect.
- $\gamma : S \times A \rightarrow S, \gamma(s, a) \mapsto s \setminus \text{Del}(a) \cup \text{Add}(a)$, a state transition function, which applies positive and negative effects on the state.
- R is the set of unique resources $r \in R$ in the environment that can be held by one agent at a time.
- $M_\Sigma = (S, A, E, R, \gamma)$ the environment, common to all agents.

We then define the goals of our agent in the environment as the set G, such that for each $g \in G$, $g = (\pi, c, m, \rho)$. Here,

- $\pi = \langle a_1, a_2, ..., a_n \rangle$ is a plan for a goal defined through a sequence of actions $a_i \in A$. In our domain there is one fixed plan for each goal.
- c are the formulation preconditions of the goal. In our domain, these are FOL formulas over D.
- m is the mode of the goal as it progresses through the goal reasoning process.
- ρ is the set of resources $r \in \rho$ that the agent needs to hold, to actively work on a goal.

A goal is only formulated if the world state satisfies its preconditions, i.e. for a goal g with preconditions c to be formulated in world-state s, $s \models c$ must hold. Therefore the space of goals that are formulated in the state s is $G_{\text{form}}(s) = \{g \in G \mid s \models c \in g\}$. The goal reasoner then selects one of the goals from $G_{\text{form}}(s)$ based on a priority function, tries to acquire the resources and eventually executes and evaluates the associated plan. There is no explicit representation of the other agents or their intentions in this model, instead their actions are perceived as exogenous effects in the world.

4.1.2 Promise Semantics in the CX

We use the term promise to describe a form of short-term look-ahead from the current world-state. Using promises, we want to influence an agent's goal formulation to enable it to incorporate the currently disptached goals of itself and other agents. The effects of goals can be viewed as an implicit representation of the intentions of an agent. Therefore we model promises as the effects of the plans that are currently executed by the agents.

Defining Promises

Our aim is to allow agents to incorporate other agents' intentions in their goal formulation process through promises, before these effects could be applied by executing the corresponding actions. Since goal formulation is based on the current world-state, we need to apply effects to the world-state s before execution is completed, which allows us to formulate on potential future world-states.

Formally we can denote the promises p_g for a goal g the following way:

$$p_g = (p_g^+, p_g^-)$$

with $p_g^+ \subseteq \bigcup_{a \in \pi} e_a^+$ and $p_g^- \subseteq \bigcup_{a \in \pi} e_a^-$, which is a tuple of the subsets of the positive and negative effects of all actions of the plan for the goal. Since the effects of the sequential actions of a plan might cancel each other out, p_g^+ and p_g^- need to be determined by applying the effects of the action of a plan sequentially, using a technique like effect chaining as described by Hofmann et al. [14]. The resulting promise effects are akin to macro effects in macro planning.

Incorporating Promises

We explored two ways of incorporating promises in the reasoning process. Both of them adapt the world-state s, but differ in the way they handle negative goal effects:

- **Extending** s (the permissive approach): Extending the environment M_Σ with a set P of promised facts is a simple way of introducing promises into the existing goal reasoning formalism. We view P as the set of the promises of the currently dispatched goals G_{disp}, but limit ourselves to the positive facts, since we reason under the closed-world assumption:

  $$P = \bigcup_{g \in G_{\text{disp}}} p_g^+.$$

  Under this extension, a precondition set c for a goal g holds for the current state s and promises P if $s \cup P \models c$. Since $s \subseteq (s \cup P)$, and under the assumption that there are more positive than negative preconditions, the space of formulated goals $G_{\text{form}}(s)$ is most likely increased. This approach does not negate any true statements that we currently know about the world, as it only increases the amount of positive literals we reason on.

- **Modifying** s (the strict approach): We extend the environment with a set P of promised facts again, but view P is the set of both positive and negative facts, the following way:

$$P = (\bigcup_{g \in G_{\text{disp}}} p_g^+, \bigcup_{g \in G_{\text{disp}}} p_g^-).$$

  Instead of simply looking at the union of s and P, we rather treat the promised facts like the effects of an already executed action, adapting s in a similar manner. This leads to a new projected world-state s', that is akin to the expected state, after all currently dispatched goals are achieved by the involved agents:

$$s' = s \setminus P^- \cup P^+.$$

  We then view $G_{\text{form}}(s')$ not as an extended set of formulated goals, but rather as the set of goals that would be formulated at a future time-step in the execution, assuming that all goals are successfully reached.

4.1.3 Request Semantics in the CX

Within the CX, there are two different abstract objects an agent can possibly request: a goal or a resource. In the case of a goal request, one agent subtasks a goal to another agent, to reduce or distribute the workload of the first agent. Since we formulate all possible goals in our goal reasoning approach, and use decentralized reasoning, this form of requests was not further investigated as part of this thesis. In the case of a resource request, one agent requests ownership of one or more resources from a second agent that is currently owning them.

Defining Resource Requests

Resource requests communicate the intention to take over the ownership of already held resources from another agent in the system. The resource-allocation process is not an integral part of goal reasoning, but rather a synchronization technique used by the CX to coordinate the agents and can be omitted for goal reasoning in a single-agent system. Because time is not represented in the GR semantics we used but helps when modeling explicit multi-agent interaction (like resource-locking) we will switch to a new model that has an explicit representation of time. We will build this model as an extension of the previous one and let:

- $\alpha = \{\alpha_1, \alpha_2, ...\}$ be the agents acting in our environment,
- $S_t : t \to S$ be the function that maps from time t_i to the state at that time $s \in S$,
- $G_{\text{disp}} : t \times \alpha \to G$ be the function that gives us the dispatched goal for given time t_i and agent α_j. We restrict ourselves to one active goal per agent in this model.

We can then get a mapping from the time t_c to the resource allocation by utilizing that we know that a goal g needs to have its resources $\rho \in g$ allocated to be dispatched:

$$R_t(t_c) = \bigcup_{\alpha_i \in \alpha} \bigcup_{g \in G_{\text{disp}}(t_c, \alpha_i)} \bigcup_{\rho \in g} \rho$$

Based on this, we can define a lenient version of resource requests, where a request is nothing more than a shared tuple $(R_{\text{req}}, \alpha_i)$ of a set of resources $R_{\text{req}} \subseteq R_t(t_c)$ and the requesting agent α_i.

Incorporating Resource Requests

We formulate requests in a goal-oriented way, where the authorities for creating requests are goals. For each agent $\alpha_i \in \alpha$, the resource request at the time t_c for a formulated goal $g \in G_{\text{form}}(S_t(t_c))$, is the set $R_{\text{req}} = R_t(t_c) \cap \rho_g$. This means that the agent is requesting all resources that are currently held by other agents, which its formulated goals require.

However, this approach has limits because it blocks the goal from progressing further in the goal reasoning process until the resources can actually be acquired. This reduces performance and can, depending on the implementation, lead to goal-failure. To circumvent this, we replace the resources that were requested with placeholder resources, that indicate that these resources are not held yet:

$$\rho'(\rho) = \rho \setminus R_{\text{req}} \cup \{r_{\text{mod}} \mid \; r \in R_{\text{req}} \text{ and } r_{\text{mod}} \notin R_{\text{req}}\},$$

where for each r we replace it with the resource r_{mod}. This enables the goal to keep track of the still missing resources and allows it to move ahead in the goal reasoning process, and get dispatched even before the resources are acquired. By using the modified resources, we also ensure that no one can request resources from the requesting agent. Finally, since we use one set of modified resources to with a bijective mapping for all resources, no additional goal can proceed that is attempting to request the same resources, as these are already held, avoiding increasingly complex request dependencies.

4.1.4 Formulation with Promises and Resource Requests

We can now combine promises and resource requests in the form of an extended goal formulation mechanism on the projected world state s' with coupled resource replacement. First we need to determine the currently active promises P_{tot} for all agents, which we get by computing the unions of the positive and negative promises for all goals over all agents:

$$P^+_{\alpha_i} \subseteq \bigcup_{g \in G_{\text{disp}}(t_c, \alpha_i)} p^+_g,$$

$$P^-_{\alpha_i} \subseteq \bigcup_{g \in G_{\text{disp}}(t_c, \alpha_i)} p^-_g,$$

$$P_{\text{tot}} = (\bigcup_{\alpha_i \in \alpha} P^+_{\alpha_i}, \bigcup_{\alpha_i \in \alpha} P^-_{\alpha_i}).$$

We can compute $P^+_{\alpha_i}$ and $P^-_{\alpha_i}$ using effect chaining again, akin to p^+_g and p^-_g. Then, we can determine the projected world-state s' using the strict approach: $s' = s \setminus P^-_{\text{tot}} \cup P^+_{\text{tot}}$. We now have the base set of goals that we can formulate once we apply the promises $G_{\text{form}}(s')$, for which we need to replace the resources of those goals, that require a resource that is already held by an agent. Therefore, we create requests for the missing resources, for each goal $g \in G_{\text{form}}(s')$ with resources ρ: $(\{r \mid r \in R_t(t_c) \text{ and } r \in \rho\}, \alpha_i)$. Finally, we can compute our new set of goals with the replaced resources ρ' as defined above:

$$G'_{\text{form}}(s') = \{(\pi, c, m, \rho'(\rho)) \mid (\pi, c, m, \rho) \in G_{\text{form}}(s')\}$$

A selection function can now choose one of the goals, including the promised goals, based on a priority function.

4.2 Implementation in CLIPS

We implement promises and requests as extensions of the CLIPS Executive agent, using the CLIPS production system language. Since the base-agent needs to be modified for every domain it is used in, we focused our implementation to a certain extend on the specific version of the CX for our evaluative domain, the RCLL. However, most of our concepts are just modifications of the common behavior of the CX, thus it is applicable in other domains as well, without requiring vast adaptions beyond the domain-specific parts.

4.2.1 Promise Implementation

Promises are intended to improve the decision making ability of the CX goal reasoner by allowing one-goal look-ahead across all agents. As discussed before this is achieved by modifying the world-state as if the effects of the active goals' plans have been already applied to the world. To integrate promises into the CX, several phases of the goal reasoning process had to be adapted and amended.

We can take two different stances that help us pick apart the different changes we have to make when implementing promises. Figure 4.1 locates the changes that are required for the integration of promises into the goal reasoning process at the different stages of the goal lifecycle. From promising goal's (a goal which creates promises) position, we need to focus on promise assertion, promise integration and clean-up:

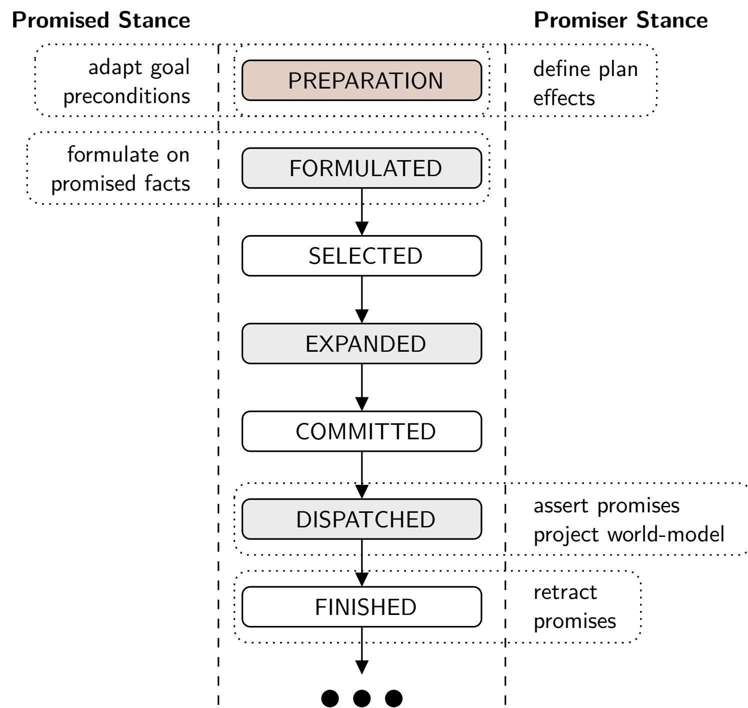


Figure 4.1: Modifications of the CX for promise integration located at their corresponding modes in the (incomplete) goal lifecycle. The modification from the promised goal's stance are located left, while the modifications from the promising goal's stance are located right of the lifecycle.

1. Preparation: Each goal has one or several plans that can either be fixed or generated on demand. They need to be analyzed and the plan effects need to be determined for each goal and made accessible, such that they can be used to assert promises.

2. Assertion/World-Model Update: Once a goal has been dispatched, we now know which effects the goal will have. The effects need to be integrated into the world-model, such that they become accessible to the other agents. This corresponds with the projected state s' given the current state and the promises.

3. Clean-Up: After the plan of a goal is executed, it reaches the finished state. The actual effects of the plan are now applied to the world-model, thus the promises need to be retracted, potential inconsistencies in the projected state need to be resolved.

From the point-of-view of a promised goal (which is a goal that is formulated because of promises) it is relevant how to react to promises in the world-model and how to handle the edge-cases that are the result of being based on a future state of the world:

1. Preparative: The preconditions for goal formulation need to be adapted in such

a way, that they can also be fulfilled with promise-facts. That means to change the preconditions from being based on the world-state s to being based on the projected state s'.

2. Formulation: We need to track which promises were used as preconditions to eventually acquire the resources using resource requests.

Translating Plan-Effects to Promises

Integrating promises into the CX starts with determining the plan-effects of the plans of each goal and translating them into facts we can reason on. In the CX goal plans can either be generated on-line using a planner and the PDDL-based domain description, by using a plan database, or by having fixed plans for each goal. In either case the possible actions of the agents would be described in the form of a PDDL-action. Except for the addition of parameters, it is equivalent to the formal definition of action preconditions we used before.

Listing 4.1: An example PDDL-action for preparing a RCLL delivery station.

```
1 (:action prepare-ds
2   :parameters (?m - mps ?ord - order)
3   :precondition (and (mps-type ?m DS) (mps-state ?m IDLE) (locked ?m))
4   :effect (and (not (mps-state ?m IDLE)) (mps-state ?m PREPARED))
5 )
```

Listing 4.1 shows an example of a typical action in the RCLL domain. This specific action requires that the Modular Production System (MPS) is in state `IDLE`, that it is of type delivery station `DS` and that the machine is locked by the agent. The effect of the agent is that the MPS is now in state `PREPARED`.

In the case of on-line planning or even for bigger plan-databases, the only effective way to generate the promises from PDDL-actions is to parse the plan-actions after the goal has been expanded and a specific plan is selected. Since our domain has a limited set of fixed plans, the promises have been hand-crafted by assessing the relevant effects from each action of a fixed plan. Then, for each goal, a CLIPS-rule was created, that asserts the promises when the corresponding goal gets dispatched.

The CX stores all the relevant facts describing the world-model in facts of the class `wm-fact`, a subtype of which is used to store the `domain` facts, which are shared among the PDDL description and the CLIPS environment. For the precondition `mps-type` of the PDDL action in Listing 4.1, a corresponding `wm-fact` is depicted in Listing 4.2. Since the world-model of the CX reasons under the CWA, all facts represent positive literals.

Based on the existing domain-facts, a new type of world-model facts, called promise-facts, was added that represents the elements of the set P of our formal model. For each domain-fact, the corresponding `promise` fact is of the form depicted in Listing 4.3.

Listing 4.2: The domain-fact representing that the mps-type of the machine C-DS is DS (delivery station).

```
1 (wm-fact (key domain fact mps-type args? m C-DS t DS) (value TRUE))
```

Listing 4.3: The promise-fact representing that the mps-type of the machine C-DS is DS (delivery station).

```
1 (wm-fact (key promise <robot> <goal-id> fact mps-type
2                args? m C-DS t DS)
3          (value <TRUE/FALSE>))
```

The general structure is identical to the respective domain fact. However, since these facts are not unique (as they represent specific effects of a plan, not the world-model itself), the slots robot and goal-id were added, which allow unique identification of the promising agent, and support promise fact management. Since we reason under the CWA, domain facts only represent the atoms that describe the current known truths in the world. promise facts however, describe effects or changes to the world and thus can be positive, or negative just like in our formal model. We represent their truth value using the value slot.

The lifespan of promise facts is limited to the time their goal is in the dispatched state in the goal reasoning process. Once all actions are executed, i.e. the goal leaves dispatched, the promise facts are retracted. Both assertion and retraction are replicated using the existing shared world model mechanism which is built on top of a common MongoDB database.

Creating and Maintaining the Projected World Model

After promise facts have been asserted following a goal entering the dispatched mode, they need to be merged with the domain facts to enable the other agents to formulate new goals based on them. We follow the strict approach introduced in Section 4.1.2 and build the new, projected world model akin to s', by applying the promised effects, both positive and negative, onto the current world model. Since we do not want to modify our fact base of domain facts, which still should hold only the actual truths of our world, but rather want to construct a new one based on the the domain facts and the promises, we use a second new type of world model facts called projected-promise, an example of which is given in Listing 4.4. These new facts take the general structure

Listing 4.4: The projected-promise representing that the mps-type of the machine C-DS is DS (delivery station).

```
1 (wm-fact (key projected-promise <robot> <goal-id> fact mps-type
2                args? m C-DS t DS)
3          (value TRUE))
```

of the promise facts, including the unique identifiers and will replace the domain facts in the goal preconditions. To create the projected world model s' we need to:

- replicate and update the current set of domain facts into a set of projected-promise facts,
- apply the current positive, and negative promises on the projected-promise facts,
- retract effects of retracted promises.

While the promise facts are shared, we compute our projected world model locally, in order to reduce traffic and avoid synchronization of duplicate facts. The first step in creating the projected world model is to replicate the current domain factset into the projected-promise fact set. To accomplish this task, a simple CLIPS rule can be defined. Listing 4.5 creates a copy of each domain fact, for which a projected-promise

Listing 4.5: A CLIPS rule that asserts a projected-promise for a domain fact.

```
1 (defrule assert-projected-promise-from-domain-fact
2   (wm-fact (key domain fact $?args))
3   (not (wm-fact (key projected-promise NO-PROMISE LOCAL fact $?args)))
4   (not (wm-fact (key promise ?gid ?r fact $?args) (value FALSE)))
5   =>
6   (assert
7     (wm-fact (key projected-promise NO-PROMISE LOCAL fact $?args))
8   )
9 )
```

does not yet exist. In place of the robot and goal identifiers (goal-id), we use the symbols NO-PROMISE and LOCAL to indicate the origin from a domain fact. Note that the third precondition of the rule requires that no negative promise for the same fact exists.

The next step is to merge the current set of positive promises with the translated domain facts. To do this, a similar CLIPS rule can be used, as shown in Listing 4.6. The rule fires, whenever there is a promise without a corresponding projected-promise. Once again, there is a precondition that prevents formulation if a negative promise for the same fact exists.

Eventually, we need to (temporarily) remove projected-promises for which a negative promise exists in order to apply all promised effects correctly. The first step, the prevention of assertion of new projected-promises if a negative promise exists, is already handled by the preconditions in the rules 4.6 and 4.5. The final case left to handle is the retraction of existing projected-promises, when a negative promise for the same fact is asserted, for which we define a rule as depicted in Listing 4.7.

Note that once the negative promises for a fact have been retracted and there are still promises or domain facts, the projected-promise needs to be reasserted. With our rule design, this is done automatically, once the negative promise is retracted.

Listing 4.6: A CLIPS rule that asserts a `projected-promise` for a `promise` fact.

```
1 (defrule assert-projected-promise-from-domain-fact
2   (wm-fact (key promise ?gid ?r fact $?args))
3   (not (wm-fact (key projected-promise ?gid ?r fact $?args)))
4   (not (wm-fact (key promise ?gid2 ?r2 fact $?args) (value FALSE)))
5   =>
6   (assert
7     (wm-fact (key projected-promise ?gid ?r fact $?args))
8   )
9 )
```

Listing 4.7: A CLIPS rule that retracts a `projected-promise` for a negative promise-fact.

```
1 (defrule retract-projected-promise-temporarily-from-negative-promise
2   (wm-fact (key promise ?gid ?r fact $?args) (value FALSE))
3   ?pm <- (wm-fact (key projected-promise ?gid ?r fact $?args))
4   =>
5   (retract ?pm)
6 )
```

Finally we need to retract `projected-promise`s that have no supporting promises or `domain` facts. For this we can define two rules, one for each case. The rule for the promises case is given in Listing 4.8. It checks if there is a `projected-promise` with goal and robot identifiers of a promise, but no corresponding promise.

Listing 4.8: A CLIPS rule that retracts a `projected-promise` if there is no supporting promise.

```
1 (defrule retract-projected-promise-from-promise
2   ?pm <- (wm-fact
3       (key projected-promise ?gid&~NO-PROMISE ?robot&~LOCAL fact $?args))
4   (not (wm-fact (key promise ?gid ?robot fact $?args) (value TRUE)))
5   =>
6   (retract ?pm)
7 )
```

This approach might still lead to ”duplicates” in the `projected-promise` set, because multiple promises or a promise and a `domain` fact might exist for the same fact, which is possible because of the unique identifiers. This, however, does not cause any problems since the only side effect is that a goal might be formulated multiple times, once for each `projected-promise` that matches.

Adaption of Goal Formulations

Listing 4.9: Simplified example of a goal formulation rule for the RCLL domain, akin to those found in the CX. Here a goal of the class `GOAL-CLASS` with a parent of the class `GOAL-CLASS-PARENT` is formulated, if there is a formulated possible parent and corresponding facts of type `meta-fact1`, `refbox-fact` and `domain-fact1`. Additionally there must be no `domain`-fact of type `domain-fact2`.

```
1  (defrule goal-example
2    (goal (class <GOAL-CLASS-PARENT>) (id ?parent-id) (mode FORMULATED))
3    (wm-fact (key order meta <meta-fact1> args? <args>))
4    (wm-fact (key refbox <refbox-fact>) (value <value>))
5    (wm-fact (key domain fact <domain-fact1> args? <args>))
6    (not (wm-fact (key domain fact <domain-fact2> args? <args>)))
7    =>
8    (bind ?id (sym-cat <GOAL-CLASS>- (gensym*)))
9    (bind ?req-res (create$ resource1 resource2))
10   (assert (goal (id ?id) (class <GOAL-CLASS>)
11                 (sub-type SIMPLE)
12                 (priority ?*PRIORITY-<GOAL-CLASS>*)
13                 (parent ?parent-id))
14                 (params ....)
15                 (meta ....)
16                 (required-resources ?req-res)
17           )
18   )
19 )
```

Now that we have established projected world model, that incorporates our positive and negative promises, we need to modify the goal formulation preconditions to use the `projected-promise`s instead of the `domain` facts. A typical goal-formulation rule for the RCLL domain in the CX is similar to the example given in Listing 4.9, which shows the general structure of a typical goal formulation rule. Besides the `domain` facts that we use to assess that the world is in a state in which our goal makes sense to be formulated, we can find other kinds of facts in the preconditions. These aid us by supplying additional information (like refbox facts, which we can use to access data like the current game time or state) or by providing more granular control of the production flow (like order facts which hold information about the orders, which are not part of the world model). However, since our changes are on the world model level, these do not concern us.

With the `domain` facts already merged with the promises in the form of the `projected-promise`s, changing the goal formulation to reason on the projected world model now only requires the replacement of the `domain` facts in the formulation rule preconditions. For the above case, a possible replacement is depicted in Listing 4.10. Besides replacing the fact type, we need to add variables for the goal-id and the robot-id, such that the pattern

Listing 4.10: Excerpt of a goal-formulation rule where the `domain` fact preconditions have been replaced by `projected-promises`. The dots indicate unmodified parts of the rule.

```
1  (defrule goal-example
2    ....
3    (wm-fact (key projected-promise ?gid1 ?rid1
4              fact <domain-fact1> args? <args>))
5    (not (wm-fact (key projected-promise ?gid2 ?rid2
6                   fact <domain-fact2> args? <args>)))
7    =>
8    (bind ?id (sym-cat <GOAL-CLASS>- (gensym*)))
9    (bind ?req-res (create$ resource1 resource2))
10   (assert (goal (id ?id) (class <GOAL-CLASS>)
11   ....
12 )
```

matching is functional. At this point the decision might be made to bind preconditions to the same source (i.e. the same goal-id and robot-id across multiple preconditions) to avoid certain inconsistencies that can occur in the projected model. Additionally, goal-id and robot-id can be used to determine if the goal fired on promises or `domain` facts, which will be used later on. Finally, we need to consider which formulation rules we want to modify, in the case of the RCLL we will focus on the production rules, which control the robots behavior during the production phase of the game.

4.2.2 Request Implementation

Resoure-requests allow us to have a controlled resource handover from one agent to another, or more specifically from one goal to another, which we can use to improve the promise behavior, and to allow a promised goal to be dispatched before the promising goal is finished, if it holds a resource that the promised goal requires.

Resources in the CX are symbolic representations of objects in the world, like a location on the field, a workpiece, or a MPS that are used for inter-agent synchronization. By locking a resource, the agent can block all goals of other agents from advancing in the goal lifecycle that also require the resource. The lock state of all resources is shared between agents, and locks are only granted once all agents are informed of the lock request. A typical flow of acquiring and releasing resources for a goal during its lifetime is:

1. The agent formulates a goal, each goal has a specific set of resources that are required for proper operation. In this example, the goal requires the resource `C-RS1-INPUT`, which is the input for the first ring station of team CYAN.
2. When the goal is committed, the agent tires to acquire its associated resources int the form of a lock request, which is replicated to the other agents. If the

lock request succeeds, the resources are acquired by the agent, and all agents are informed of this.

3. Once the resources (or in this case resource) are acquired, the goal can be dispatched and thus plan execution can be started.

4. After execution and goal evaluation, all resources are completely released one by one, regardless of a potential successor goal. Each agent can then claim the now freed resources again.

Because resources are completely released, once the goal finishes, this creates a problem for promised goals (i.e. goals that were formulated based on promises). A promised goal that requires locked resources can neither be dispatched until all resources are owned, nor is it guaranteed to get the resources once they are released, since another goal might try to request the exact same resource. To overcome this, we use resource requests in the CX, which allows us to have an orderly handover of resource ownership from the promising goal (i.e. the goal that created the `promise` facts) to the promised goal. Integrating resource requests requires changes to the goal formulation process and to the locking mechansims.

Communicating Resource Ownership

The CX allows multiple goal trees to exist, which can be executed in parallel. Since each of the leaf goals might hold resources and emit promises, we need to supply additional information to the agents, in order for them to know which resources are associated with which goal, as we only want them to request resources from the promise emitter. While requesting the resources of other goals is technically possible and thus would allow early execution even of a non-promised goal, this is not a behavior that we intend.

To supply the required information, we introduce a new set of wm-fact facts as shown in Listing 4.11, referred to as `promise-resource`. These facts are asserted/retracted at the same time the promise-facts are asserted/retracted. The `promise-resource` facts

Listing 4.11: Structure of a `promise-resource` facts.

```
1 (wm-fact (key promise-resource fact args?
2               robot <robot-id> goal <goal-id>)
3          (values <resource-symbols>) (type SYMBOL) (is-list TRUE))
```

use the robot-id and goal-id for identification, just like `promise` facts and `projected-promise` facts. The values slot contains all the resources that the emitting goal owns. We want to match each promise in the precondition with a `promise-resource` fact. However, since we do not use the `promise` facts directly in the preconditions, but rather the merged `projected-promise` facts, which can also come from `domain` facts, we need to assert blank `promise-resources` with `goal-id NO-PROMISE` and `robot-id LOCAL` on each robot to match with `domain` fact based `projected-promises`.

Requesting Resources

Having the required information about resource ownership to perform requests, we implement a request process that is based on two mechanisms: shared wm facts are used to control a multi-step handover process; and resource replacement. Both of these steps are akin to the formal model we established in Section 4.1, with the shared facts being an implementation of the tuple $(R_{\text{req}}, \alpha_i)$ and the resource replacement, being based on the computation of the resource set ρ'.

The resource replacement is part of the formulation process. A goal that is formulated based on promised facts uses the `projected-promise` preconditions slots for goal-id and robot-id to match a corresponding promise-resource. For each `projected-promise` precondition that establishes a new set of goal-id and robot-id variables, we add a new promise-resource precondition, that uses these same variables, as shown in Listing 4.12.

Listing 4.12: Modified preconditions for goal formulation including the `promise-resource` fact that is matched to the `projected-promise`.

```
1  (wm-fact (key promise-resource fact args? robot ?rid1 goal ?gid1)
2           (values ?resources1))
3  (wm-fact (key projected-promise ?gid1 ?rid1
4            fact <domain-fact1> args? <args>))
```

If a `projected-promise` that is based on a `domain` fact matches, then the placeholder promise-resource will be used, which has an empty value field. If a `projected-promise` that is based on a `promise` fact matches, then the placeholder will contain the resources of the goal that also created the promise.

Next, we need to perform the replacement of the resources to enable early dispatching of promises goals as introduced in the formal model. We use a method akin to the one established in section 4.1, replacing the resources with ones of a modified copy set with an additional prefix added to each resource symbol. We implemented this by comparing the resources variables that we get from the promise-resource facts and the resources that are required, replacing each match with a new symbol that has the prefix `PROMISE-`. For the resource `C-RS1-INPUT`, the result would be `PROMISE-C-RS1-INPUT`.

After we replaced the resources that we request from a promising agent, we now need to create the actual requests. In our implementation, the request tuples are represented by shared wm facts of the following structure, as shown in Listing 4.13.

Listing 4.13: The structure of a resource request fact.

```
1  (wm-fact (key request-resource fact args?
2                  robot <robot-id> requester <goal-id>
3                  requested <goal-id> resource <resource>)
4           (is-list TRUE) (values <true/false> <true/false>))
```

The slots `robot` and `requester` identify the agent that asserted the request, while `requested` and `resource` identify the promising goal and the resource we want to acquire from it. The slot `values` is used to mark the state of the handover process, with the first slot indicating whether the request is active (i.e. the requesting goal has been

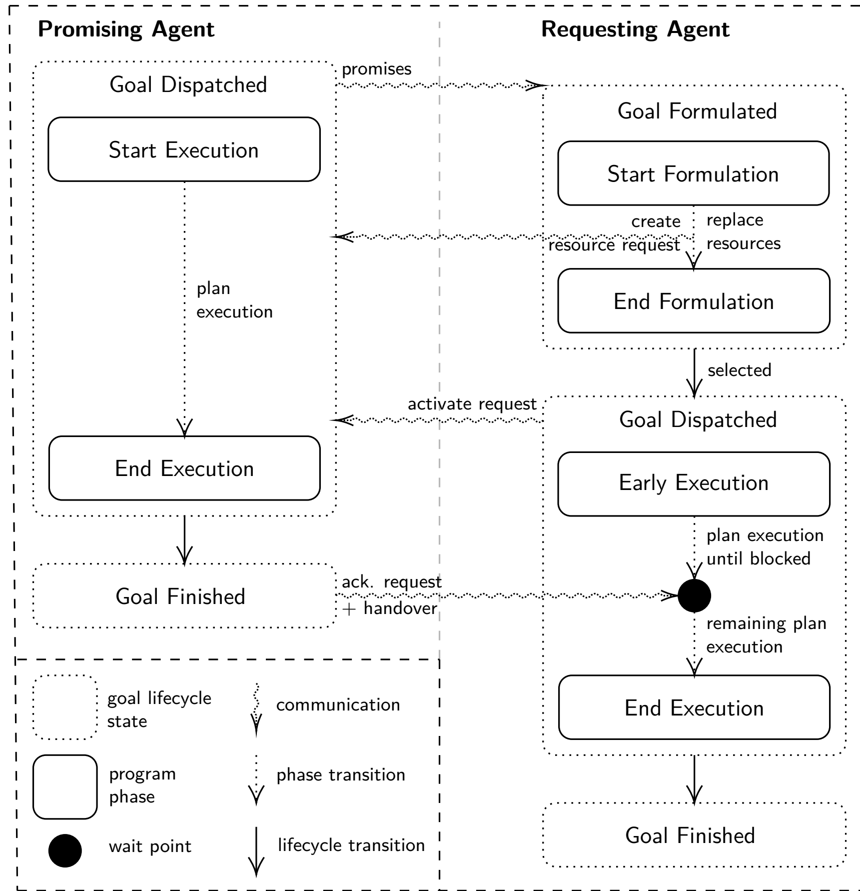


Figure 4.2: Diagram of the handover procedure of requested resources over the lifetime of two goals. The promising agent causes the formulation of the goal on the requesting agent's side. When the promised goal is dispatched, it activates the request. The promising agent acknowledges the request and hands the resources over once the goal is finished.

dispatched), and the second slot indicating the state of the request acknowledgement (i.e. if the promising goal is ready to hand over the resource).

Figure 4.2 depicts the request and handover process between two agents on the example of one resource. The requesting agent is formulating a goal based on the promises emitted from the promising agent. It then performs the resource replacement for the resources that it wants to acquire from the second agent and asserts a request-resource fact. When the request is initially asserted, it will be of the form `(values false false)`, which means that the request is neither active (first slot) nor acknowledged (second slot). Once the requesting agent has acquired all the necessary resources (blocking new requests of the same resource) and dispatches the goal, the request will be set to active (first slot is set to `true`), which blocks other agents from claiming the original resource. When the promising goal is finished, it will start releasing its resources, acknowledging each requested resource. It might have to wait at some point, when the required resources are

not available yet. After the request is acknowledged, the requesting agent will attempt to lock the resource. Because the agent has an active request, it is guaranteed to acquire the resource, which completes the handover. Should the agent wait, it can now resume execution. Finally, when the requesting goal is finished, the resource gets released.

Problems and Limitations of this Approach

While this implementation of resource requests is simple and has proven its practical effectiveness, there are known problems and insufficiencies. One general shortcoming of early dispatching is that there is potential for conflicts during plan execution, because now two goals might actively try to work with the same resource. Since every step of the plan has its own set of locking actions independently of the goal resource locks, usually this problem is mitigated. However, it might still lead to timeouts on both the requesting goal's side (when the requested goal is too slow) or on the requested side (when the requesting goal is too fast). Overall, we expect early dispatching of goals with higher priorities to be more valuable than not dispatching them at all.

Another shortcoming is the lack of chained requests, i.e. once a requesting goal has acquired all resources, it is capable of taking requests for the newly acquired resources. Since the goal is holding the resources with the prefix until it is finished, it is blocking any such requests. The mechanism could, however, be extended by releasing a prefixed resource as soon as the actual one is acquired.

Finally, there is a potential for short-lived deadlocks in this approach. While the request blocks any other goal from acquiring the requested resources, a delay in the synchronization could cause a third agent to acquire the resources from the requested agent before the request arrives. This is possible, because the requesting agent only locks the resources with the prefix. Thus, until the request is synchronized, the resources are unprotected. Potentially, this leads to a time-out and could cause the requesting goal to fail. This behavior was not detected in the actual implementation.

4.3 Additional Modifications

After we implemented both promises and resource requests and integrated them into the CX, we needed to perform additional modifications to improve promise behavior and mitigate unwanted side effects. First, the design of the goal tree led to short promise lifetime, which greatly reduced their impact on agent behavior, thus it was modified to maximize the effects. Secondly, promises were enhanced by adding a timing feature, which automatically rejects goals, if the effects of the promising goal are expected to be applied too far in the future.

4.3.1 Goal-tree Redesign

For the RCLL domain, six distinct goal trees are used to handle a variety of different parallel processes. Of the six trees, two are of special interest to us: Maintain-Production

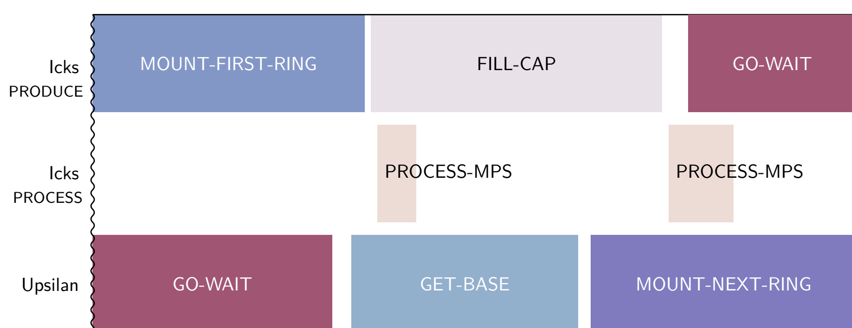


Figure 4.3: Bar chart representing the active goals of two agents (Icks and Upsilan) over time in a part of a game. For agent Icks, the goals of the production and maintain-mps (or process) trees are included, while for Upsilan only the production tree goals are shown.

and Maintain-MPS-Handling. Each is responsible for one of the two most important steps of the production process:

- the **Maintain-Production** tree is responsible for all goals that are part of the physical assembly process. This includes carrying products between the production machines, filling ring stations and preparing cap-stations or delivering finished products. All of these goals are the ones that we want to affect using promises, allowing agents to perform production in a more parallelized fashion. Here we will call these goals production goals.
- the **Maintain-MPS-Handling** tree is responsible for sending commands to the production machines to prepare them for handling the workpieces and to start the actual process the machine is responsible for. Here we will call these goals process goals. A goal in this tree always has an associated goal in the Maintain-Production tree that led to its formulation.

This split into two distinct goal trees allows the agent to perform both tasks in parallel, which is possible since the process goals do not require physical interaction, but rather are solely network based communcation tasks. Practically, this means, for example, that an agent that already fed a product into a machine, does not need to wait for the instructions to the machine to complete, but instead can already start working on a new production goal.

The problem of this split, is that the effects that would enable the next production step of a product (for example, the fact that the product is now at the output and assembled) are only applied after the process goal is completed, which also means, that the promises for these facts are only emitted by a process goal. However, the lifetime of a process goal is usually very short, leading to a reduced chance of overlap with agents formulating new goals, thus making use of the promises. Additionally, the process goal is often dispatched too late to create promises for the same agent, since there is usually already a new production goal dispatched at this point.

One example for both of these effects is highlighted in Figure 4.3: Agent Icks is dispatches a mount-first-ring goal to start production of a C2 product. It takes a base

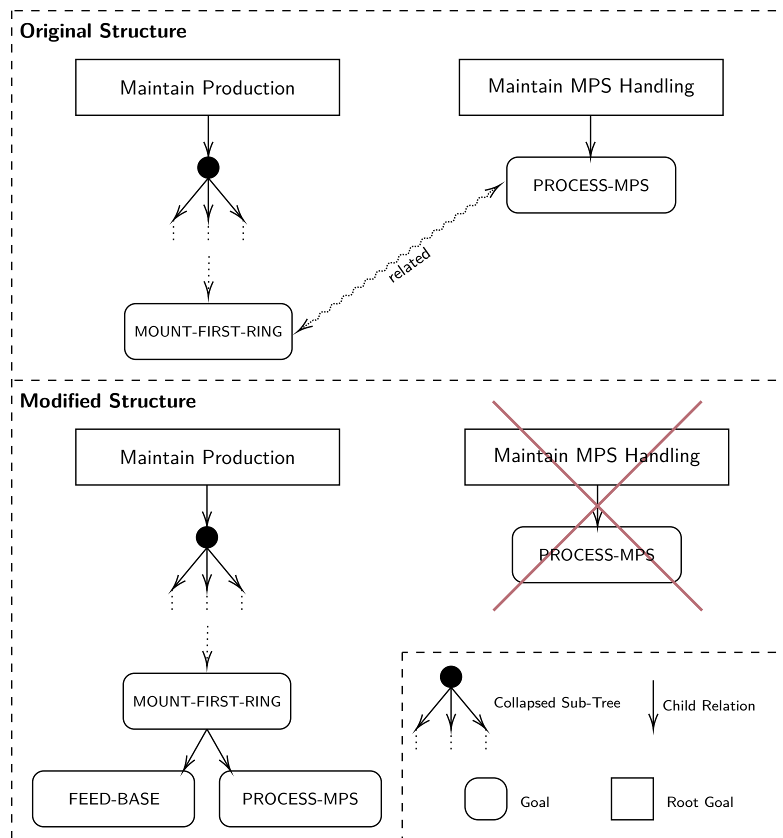


Figure 4.4: Diagram of the goal tree modification. "Maintain MPS Handling" tree and "Maintain Production" tree are combined with the children being placed ander a new common goal parent goal. The goal feed-base takes over the role of the original mount-first-ring.

from the base-station and brings it to a ring station, which mounts the first ring. Once the agent is done feeding the base into the machine, it formulates and dispatches the corresponding process goal. At the same time, it also formulates a new production goal, since the previous one finished. Because the process goal is not dispatched yet, there are no promises yet to use in the formulation. Instead it will pick the next one available, which is not optimal. On the other hand, the second agent Upsilon, has formulated a new goal just before the mount-first-ring goal finished, and thus can not take advantage of the promises of the process goal as well, even though it could have already started the MNR goal if the promises would have existed.

If any of those agents had access to the promises earlier, they could've formulated the mount-next-ring to directly continue the production, decreasing overall production time for the product.

Dynamic Restructuring of the Goal Tree

To mitigate these problems, we redesigned the goal tree to allow for both long-living promises, as well as parallelization when executing the plan of the process goal. In a first step, we removed the Maintain-MPS-Handling goal tree. Instead we combine all the production goals that had an associated process goal by bundling them both under one parent goal in the production goal tree, as highlighted in Figure 4.4. Then we reevaluated the promise-facts for the combined goals, such that the effects of both sub-goals would be considered to be the promises of the new parent goal. This increases the lifetime of the most relevant promises significantly, but does remove parallelization. The agent would now perform both actions after another starting with the production goal, followed by the process goal and would only start with the next production goal once both are finished.

We reintroduce parallelization by performing a dynamic restructuring operation on the goal tree, which we call goal lifting, an example of which is depcited in Figure 4.5. First we wait for the production goal (Feed-Base) to finish, to keep serial execution of production goals. After the goal is finished, we "lift" the parent (mount-first-ring) out of the production tree, such that it becomes its own root goal. The now childless former parent of mount-first-ring now restarts the goal lifecycle by formulating a new set of goals. At the same time, the goal reasoner starts selecting and dispatching the process goal of the lifted parent in parallel, since these are now distinct goal trees. This means that both the lifetime of the promises is extended, as well as that the agents can now formulate on their own promises.

4.3.2 Timing of Promises

Another shortcoming of plain promises is that they are generally time agnostic. This means that the promiser does not take into account, that some of the promised facts might only be true at the very end of the goal-plan execution. Similarly, promised goals are not considering when, during their own plan execution, they need a promise to be true to perform a certain action. For example, a goal might promise a fact that is the effect of one of its last operations, on which another agent might formulate a goal, soon after the promising goal has been dispatched. The promised goal, however, might need this fact to be true to execute one of its first plan actions, leading to long waiting times (until the preconditions are fulfilled) and potential timeout and thus goal failure. This problem is amplified if the goal lifetime is long, which causes potentially long overlaps between promising and promised goal.

To decrease the impact of these issues, we added a simple timing concept that detects mismatches between the approximate runtime of the promising goal and the point in time when the promised goal will require the promised effects to be true.

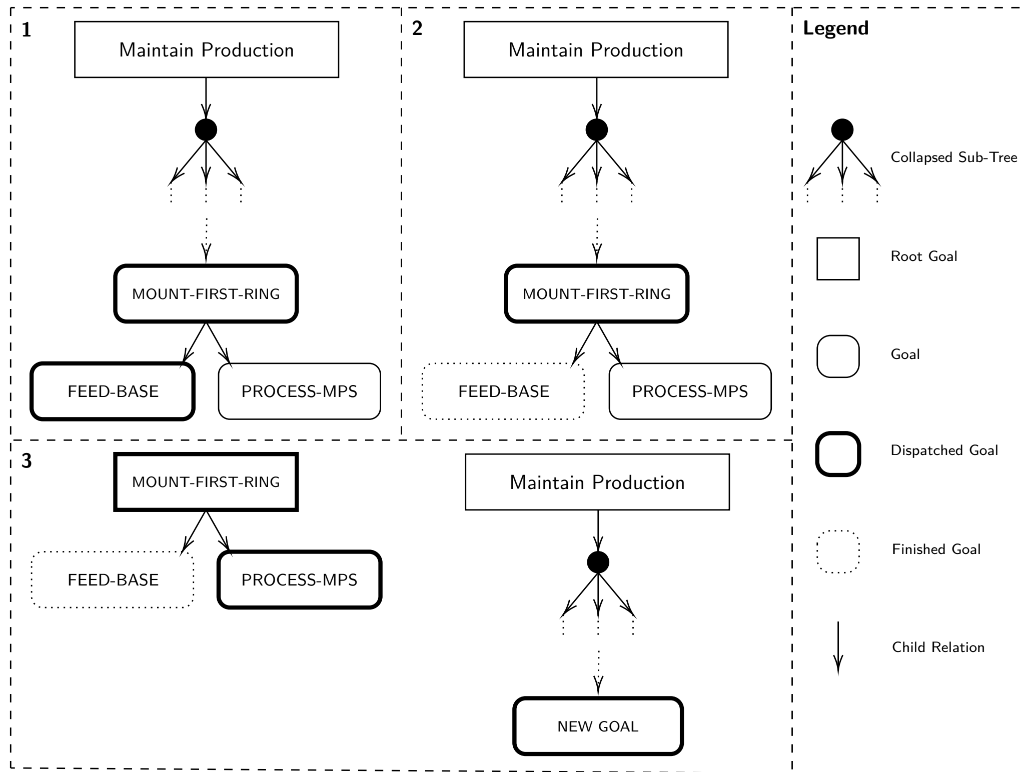


Figure 4.5: Diagram of a goal through the dynamic restructuring process. 1 - A mount-first-ring goal and its child goal feed-base are dispatched. 2 - feed-base is finished, the goal reasoner now will select process-MPS and restructure the goal tree. 3 - Because there is only the production goal left, the mount-first-ring goal is lifted out of the production tree, process-MPS is dispatched and the production tree formulates and dispatches a new goal.

Promise Timing in the CX

Our approach has two phases: First, we calculate the approximate time of a promising goal's plan based on historic times and extend the `promise` facts with this information. Second, we parse the plan of a promised goal and check for each precondition which actions preceed its first appearance. Based on the two times, we determine if there is a mismatch and reject if necessary.

In order to integrate the timing information, we first need to adapt the promises and `projected-promises`. This is simply accomplished by adding the time as the last entry of the argument list of both fact types, as shown for the `projected-promise` case in Listing 4.14. The `projected-promise` translation rules only need to be adapted for the `domain` fact case, here we will need to add the static value 0, as `domain` facts are already true statements about the world when they are asserted. We then need to specify a function that calculates the time of a goal based on its plan, for which we parse the plan actions and determine the sum of all actions of the plan. For the time calculation we chose to use static times for machine operations and workpiece operations based on

Listing 4.14: Adapted projected-promise with integrated timing

```
1 (wm-fact (key projected-promise ?gid1 ?rid1 fact <domain-fact1>
2           args? <args> time ?t1))
```

recorded real world data and to estimate travel time using euclidian distane and an appropriate average speed.

Having a time estimate, we need to add a mechanism that rejects the goal if there is a time mismatch. Now that we have an estimate for the time when a promise will be true, we can now search for mismatches between the estimate for the promiser goal and the time when the fact will be needed by the promised goal. Since we can not do this during goal formulation, as the plan does not exist at this point, we need to carry the time information for all the `projected-promise`s in the goal structure until we have a plan. This is achieved by using the `meta` slot of the goal structure (as depicted in Listing 4.9), which is empty for prodution goals. We chose to manually edit the goal assertion in our implementation, because of the limited scope. For each `projected-promise` we will take the fact name and the time variable for this fact and add them to the meta slot.

Finally we need to add a rule, that checks the time, once a plan for the goal exists, which is after the expansion mode has been reached. This rule, listed in Listing 4.15, first parses the meta slot and splits the time information and the fact-name into separate variables. It then calls the function `calculate-required-time` for each of the facts, which parses the plan, tracks all actions ahead of the first occurence of the fact in an action precondition and then uses the same method to calculate the time for the resulting sub plan that is also used for the goal plans. Finally, we compare the result with the actual promised time from the meta slot. If the promised-time is chronologically after the calculated wait-time, we reject the goal. Since we use the constant 0 for `domain` facts, there can never be a mismatch between such a fact and the estimated plan time.

By modifying the time-estimates we have influence over goal rejections, and can adapt promise “aggressiveness“. However, this obviously does not completely resolve the issue of having potential plan conflicts, especially if goals take unexpectedly long or fail. Nonetheless, it is an adequate technique to reduce the amount of conflicts.

Listing 4.15: A simplified version of the rule that is used to reject goals with a timing mismatch. All given predicates are checked for their occurence, then their promised time is compared with the time they are expected to be needed.

```
(defrule fast-reject-goals-on-promise-time-mismatch
  ?g <- (goal (id ?id) (mode EXPANDED) (meta $?predicates))
  (wm-fact (key refbox game-time) (values ?curr ?any-value))
  =>
    (bind ?ind 1)
    (while (<= ?ind (length ?predicates))
      ;bind the arguments
      (bind ?predicate (nth$ ?ind ?predicates))
      (bind ?promised-time (nth$ (+ 1 ?ind) ?predicates))
      ;determine the predicate wait-time
      (bind ?wait-time (calculate-required-time
                          ?id
                          (nth$ ?ind ?predicates)
                      ))
      ;compare the wait-time with the given promise-time
      (if (not (promised-in ?curr ?wait-time ?promised-time))
        then
        (modify ?g (mode FINISHED) (outcome REJECTED))
        (bind ?ind (length ?predicates))
      )
      ;increase the index
      (bind ?ind (+ ?ind 2))
    )
)
```

5 Evaluation

In order to understand the implications of promises and resource requests on agent behavior, we performed a series of experiments in the RCLL domain using a simulated environment. In the following section, we will introduce the setup that we used as well as the different settings and variables that can affect the outcome, followed by detailed descriptions of the experiments themselves, their results and interpretations of the results.

5.1 Simulation Environment

To perform the evaluation, we used the SkillerSimulation environment for the CLIPS Executive, which is a simple simulation of RCLL games. It is a non-graphical and non-physical simulation, replacing all the real world interactions of the agent with simple mockups on the skill level. This means that the interface, through which the agent would normally control the robot's hardware (the skill), instead calls a mockup function that returns the expected data of a successful call of the skill after a set amount of time.

The mockup skills have configurable but static times for all interactions except driving, which is determined using euclidian distance and a configurable but static driving speed. This is different from a game in the real world or a more sophisticated graphical/physical simulation in which most skills have variable time, such as travel time that is dependent on the actual path and obstacles along the way. Skills in the SkillerSimulation also can not fail – in stark contrast to the real world – where interactions with the workpieces regularly fail.

While this simplistic simulation is not a perfect representation of a real world game, it is a good approximation of a best-case scenario. We therefore concluded that an agent that performs well in the real world should also perform well in the SkillerSimulation, given that most faults are caused by hardware interferences (such as a failed gripping action) and unexpected events like blocked paths. This kind of setup also has several distinct advantages over more complicated simulations and real world testing, like speeding up simulation games, automating large test series, bypassing the need for manual machine placement and robot setup, quick adaptions to the agent and game settings, and reducing necessary computing performance. This allowed us to regularly create large samples of over 50 games without manual supervision.

5.1.1 Settings

We configured the simulation with the same set of base settings for each of our evaluation games to ensure comparability even between different experiments, choosing parameters

Table 5.1: The settings for the SkillerSimulation used in the evaluation for modified and unmodified agent.

Setting	Value	Description
Driving		
driving	0.5m/s	driving speed used to calculate travel times
enter-field	12.68s	enter the game-field from the starting position
Skills		
discard	8.25s	remove a workpiece from the game
get-shelf	24.19s	get a workpiece from the shelf of a machine
put-slide	23.49s	put a base into the slide of a ring station as payment
get	20.94s	get a workpiece from a machine output
put	19.24s	put a workpiece into the machine output
Simulation		
speed-up	4x	amount by which the simulated time is sped-up compared to the real time

to be as close to real world games as possible. For interactions with the machines and workpieces we used the mean of data collected in the 2019 RoboCup season as listed in Table 5.1, while we approximated the speed of the robot at a constant $0.5m/s$.

Our baseline agent is an unmodified version of the standard RCLL agent as it was used in the 2019 season. We ran all the evaluation games on average workstation PCs running Fedora 32. Each experiment was performed on each agent either 20 or 50 times, depending on its expected variability.

5.2 Experiments

The experiments we ran can be split into two categories: single orders, in which three robots only need to produce one product of a certain complexity, and full games. With the former we tried to understand agent behavior in detail and see how the different production steps of increasingly more complex products are affected by promises and resource requests whilst eliminating side effects from other orders. This allowed us to see if cooperative behavior has improved, and if our desired effects occur. The second set of experiments are meant to help us understand if our extension is actually an improvement of the overall agent performance in a real game, or if this requires more modifications to the agent.

5.2.1 Single Orders

In this Section we compare the behavior of the modified agent and the base agent in a simplified RCLL scenario where the robots must produce a single product of a fixed complexity. We performed 20 runs per agent per complexity starting with the simplest

Table 5.2: Results of 20 games with a single C0 order for both the modified and base agent.

Category	Value	Baseline	w/ Promises
Points	mean	36	36
	median	36	36
Delivery Time	mean	241.72s	210.17s
	median	242.00s	205.00s
	stdev	1.12	31.18
Promise Usage	dispatched	0	6.1

product (C0) up to the most complex one (C3). For this experiment we fixed the following parameters:

- Field Layout: Each machine was at the same fixed position for each experiment.
- Machine Configuration: Each ring station was responsible for the same colors.
- Ring Payment: Each ring color had the same price (amount of bases the machine needs to be fed before it can mount a ring of the color)
- Order Configuration: Each order had the same color configuration and the same delivery window (we chose the entire game, i.e. 00:00-17:00)

Through these experiments, we wanted to find out how the earliest possible delivery for a product of a given complexity is affected by promises, without any side effects from other orders influencing the agents' actions. We would expect earlier delivery for the products when promises are used, since the robot should be able to perform tasks ahead of time that are more valuable for the production and thus decrease overall production time.

Single Product - C0

In the first experiment we had the robots produce one C0 product, i.e., a base with a cap. The results are shown in Table 5.2, where we can see that both the base agent as well as our modified agent managed to produce the product in every run. We can also see that our modified agent was significantly faster, delivering the product more than 30s or 13% earlier for both mean and median, with little standard deviation for both agent versions. We therefore conclude that our agent is a consistent improvement over the base agent in this scenario. The delivery times for each game are highlighted in Figure 5.1, where we can see that each agent forms a tight cluster around their median value.

Usually, in this scenario, we witnessed positive behavior in the form of produce-C0 goals (which acquire the base and bring it to the cap-station) or deliver goals being dispatched earlier, thanks to the promises of other agents. Figure 5.2 shows how the

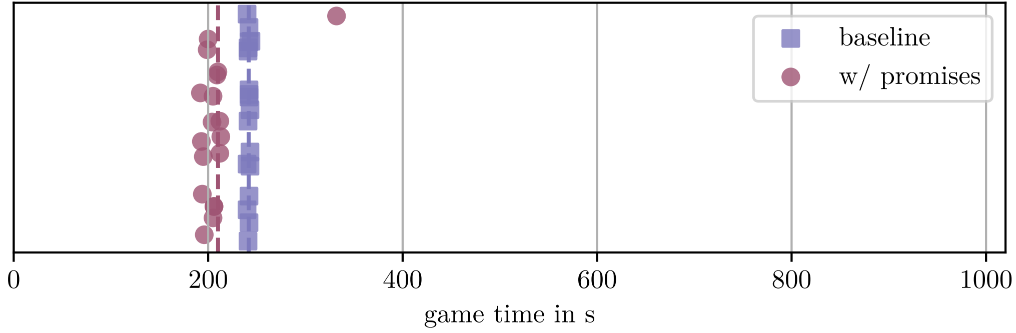


Figure 5.1: Delivery times for a C0 product plotted over game time with random scattering on the y-axis. Red dots represent our extension, purple cubes represent the baseline. The vertical lines mark the mean for each set.

Table 5.3: Results of 20 games with a single C1 order for both the modified and base agent.

Category	Value	Baseline	w/ Promises
Points	mean	48.45	45.9
	median	51	51
Delivery Time	mean	311.73s	309.44s
	median	308.00s	303.00s
	stdev	11.05	19.59
Promise Usage	dispatched	0	2.15

promise of a fill-Cap goal triggered a produce goal at the beginning of the production, leading to desirable cooperative behavior.

We also see that there was an outlier in the modified agent that far exceeded the average time for our product delivery. When we look at the course of the game, depicted in Figure 5.3, we see that one of the agents tried to produce the product ahead of time, based on the promises it was given, but this action ultimately failed, leading to a delay in the entire production.

Single Product - C1 to C3

When we increase the product complexity to include one ring (C1), we can see a shift in the results, as shown in Table 5.3. Both the baseline and the modified approach have very close mean and median values for the delivery time. We also see that the delivery times do not form two distinct clusters anymore, but rather form one coherent cluster, as depicted in Figure 5.4.

The results for complexity C2 and C3 are very similar to the results for C1, with only one major cluster being formed by the delivery times of both agents. In Table 5.4

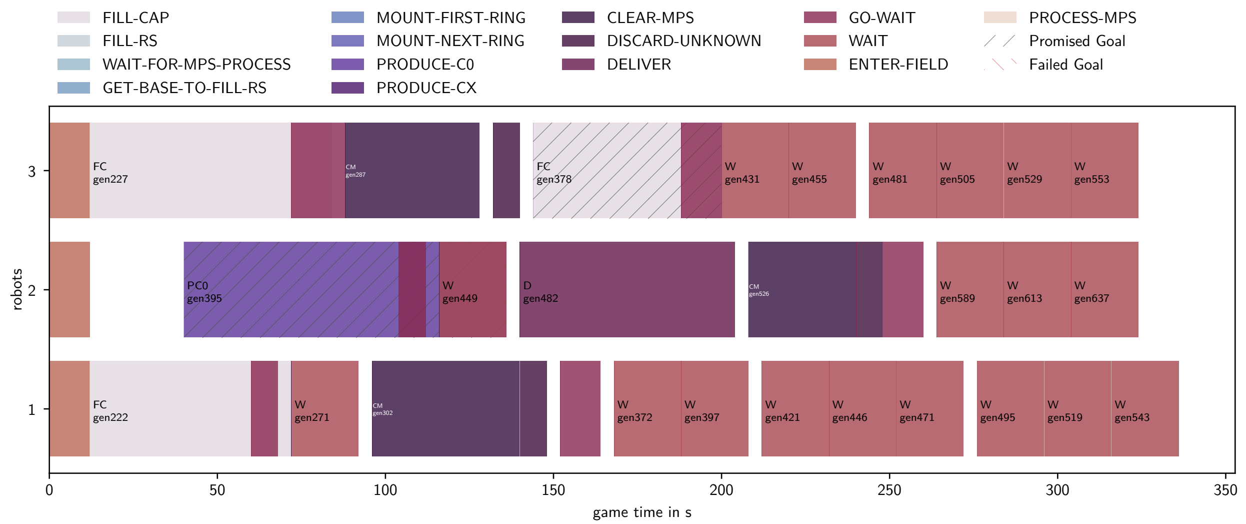


Figure 5.2: Part of a game diagram showing C0-production with successful promise-induced cooperation. A fill-cap goal caused earlier dispatching of a produce-C0 goal, while the required cap-station was still being prepared, leading to earlier delivery of the finished product.

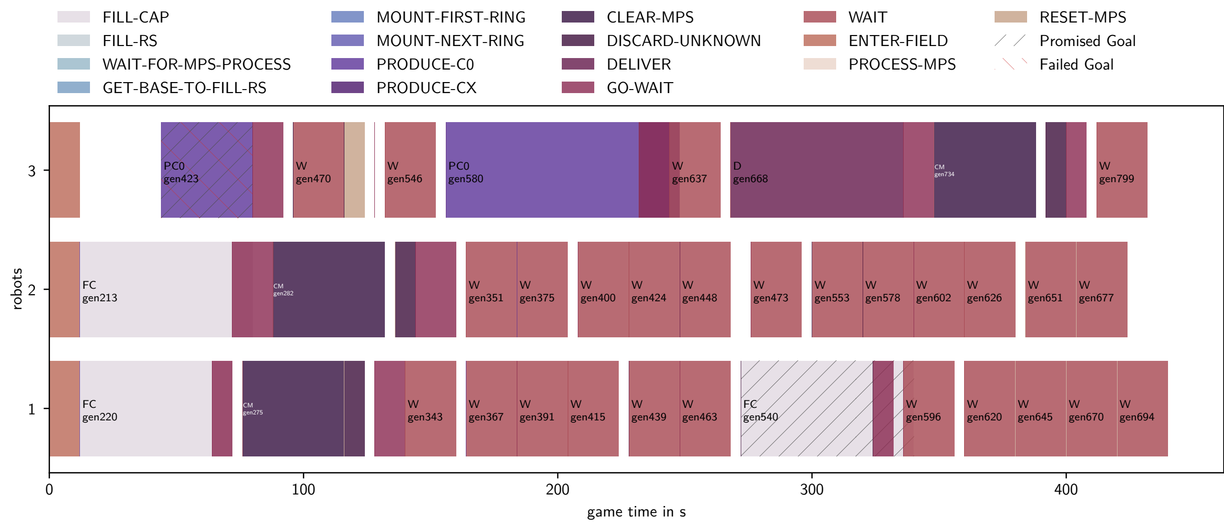


Figure 5.3: Part of a game diagram showing C0-production with unsuccessful promise-induced cooperation. A fill-cap goal caused earlier dispatching of a produce-C0 goal. However, the C0 product failed, causing a delay to the production process, leading to late delivery.

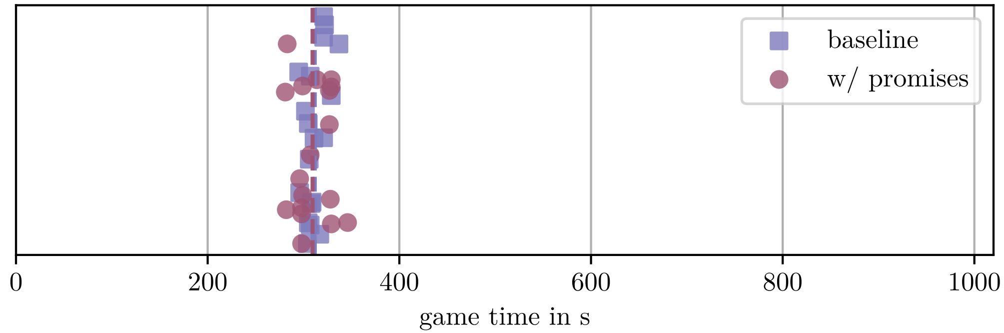


Figure 5.4: Delivery times for a C1 product plotted over game time with random scattering on the y-axis. Red dots represent our extension, purple dots represent the baseline. The vertical lines mark the median for the sets.

Table 5.4: Results of 20 games with a single C2 ordert for both the modified and base agent.

Category	Value	Baseline	w/ Promises
Points	mean	85.5	90.25
	median	95	95
Delivery Time	mean	522.78s	554.36s
	median	520.00s	527.00s
	stdev	14.30	72.47
Promise Usage	dispatched	0	3.45

Table 5.5: Results of 20 games with a single C3 order for both the modified and base agent.

Category	Value	Baseline	w/ Promises
Points	mean	133.45	149.15
	median	157	157
Delivery Time	mean	603.47s	634.37s
	median	603.00s	621.00s
	stdev	14.23	47.88
Promise Usage	dispatched	0	5.4

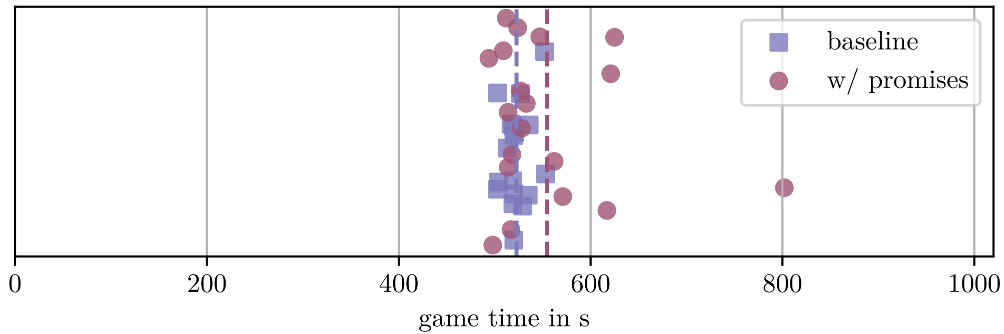


Figure 5.5: Delivery times for a C2 product plotted over game time with random scattering on the y-axis. Red dots represent our extension, purple dots represent the baseline. The vertical lines mark the mean for the sets.

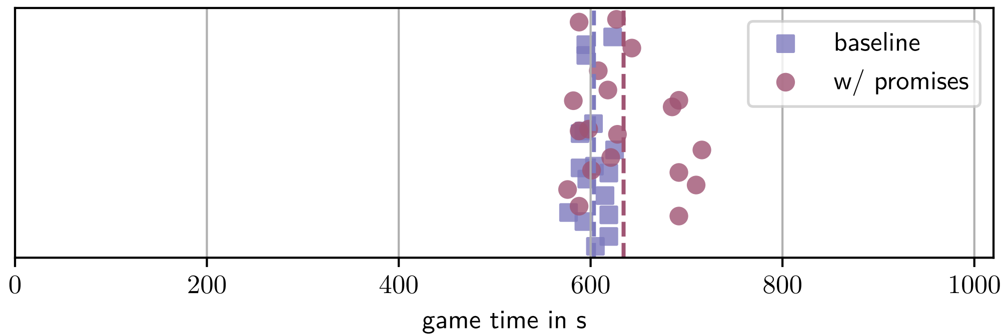


Figure 5.6: Delivery times for a C3 product plotted over game time with random scattering on the y-axis. Red dots represent our extension, purple cubes represent the baseline. The vertical lines mark the mean for the sets.

and 5.5 we see that for our key indicators, mean and median, the base agent performed better. In the case of C3, the base agent manages to deliver the product on average 5% or 30s earlier.

In Figures 5.5 and 5.6, we can see that we have distinct outliers in the delivery times of the modified agent in larger quantities. By analyzing agent behavior in the outlier cases we found a common cause responsible for the longer production times. When an agent dispatches a promised goal (one that used promises in the preconditions of its formulation), the goal can have an increased run-time compared to a non-promised one of the same kind, since the goal might have to wait for resources to be freed. When agents dispatched multiple promised goals that were not as essential to the production process itself, they ended up using more time to finish these goals, which delayed the production process and therefore caused the later delivery times. The outliers represent games where this behavior occured exceptionally often. We also found that these same phenomena

occured, albeit in lesser quantity, in almost all games for the C1 - C3 products, a possible explanation for the worse performance compared to the base agent.

Discussion

We saw that the biggest advantage of promises can be found in low-tier products. While a C0 product is produced significantly faster using promises compared to the baseline, this effect even reverts on the higher tier products like C3. We can see that longer goal time of less important goals, which can be caused by promises, can lead to significant hits in performance, thus making the modified agent a worse choice in the case of high-complexity products, which require more steps. However, when we filter for these outlier cases, we can see equal or slightly better performance even for the higher complexity products.

Therefore, it might be a more viable option to limit promises to only a specific set of goals instead of all goals as this would serve to combine the positive effects of promises, making important production steps faster, whilst mitigating the witnessed negative effects.

5.2.2 Full Games

Our second experiment is concerned with analyzing the performance of promises and requests in a regular game setting as compared to the baseline agent. By running 50 simulated games, we attempt to cover a high-range of randomly generated game configurations per agent, the results of which we analyzed to determine which agent performs better in general, and how their behavior in different situations differed. Based on the results of our previous experiment we did expect roughly equal performance, depending on the strength of the effect of the delays for C2/C3 products in actual games.

Description and Settings

During a RCLL tournament, the game parameters can change several times, with the positions of machines, delivery times, etc. being randomly generated. It is therefore important to cover a wide range of different configurations to get representative results of the agent performance. We therefore set up a large-scale test with the same basic agent configuration that was used in the first experiment, but instead of using fixed machine positions and restricted orders, we had full games of 8 orders with different, random delivery windows and different machine positions for each run, using the same randomization that is also used during tournaments. For strategic reasons, the agent is usually set to not produce a C1 workpiece, a setting we kept for both agents.

Overall, we did 50 simulated games per agent reaching a run time of more than 8 hours using 4x speed up, which we deemed necessary given the possibility for high variance in the results of random games. We did not actively reuse a game configuration for both agents, due to the large number of tests, instead choosing to cover a wider selection of

Table 5.6: Key indicators for game performance (points and deliveries) for both agents in the full game setting with mean, median and standard deviation over 50 games.

Category	Measure	Baseline	w/ Promises
Points	mean	251.59	197.74
	median	278	192
	stdev	76.36	87.69
Deliveries	mean	4.02	3.91
	median	4	4
	stdev	1.07	1.25
Promise Usage	mean goals dispatched	0	12.13

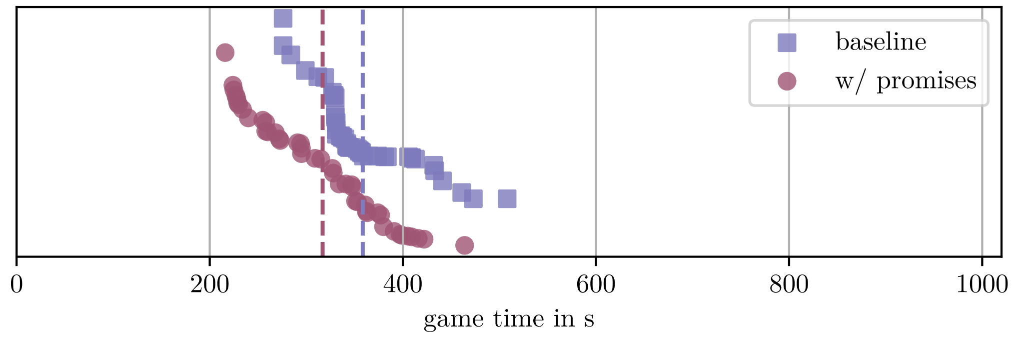
possible configurations. This also streamlined our testing procedure, as this was easily automatable given the existing setup, which does not support automatically reusing a complete game configuration.

Results

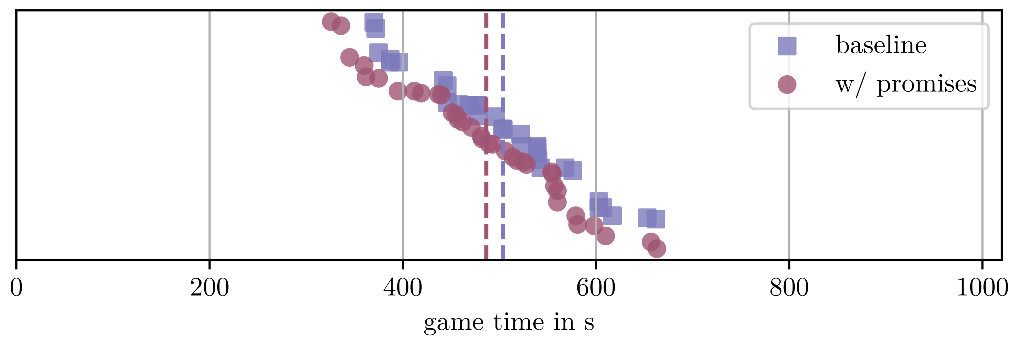
The results of the evaluation of our large sample are presented in Table 5.6. We can see that both the baseline and the agent with promises and requests have roughly the same amount of deliveries per game, with a median of 4 and a mean of 4.02 and 3.91 respectively. Therefore, we conclude, that both agents are able to produce roughly the same amount of products per game. However, when we look at the achieved points, we can see that the agent with promises performs noticeably worse than the base agent, with a mean of 197 points compared to the base agent with a mean of 251, a 22% decrease. This is a statisitically significant result with $p = 0.002$ under the assumption that both agents should perform equally well. When we look at the median values, we see an even bigger difference of 192 to 278, or a 31% decrease compared to the base agent.

However, they still deliver almost the same amount of products. Therefore, we suspected that the modified agent produces more products of a lesser complexity than the base agent. In Table 5.7, the number of deliveries per product are listed together with descriptions of the products. The agents produce roughly the same amount of O1, which is usually the first order to be fulfilled in a game. The modified agent produced less products for order O3 (C2) and especially O4 (C3), while it fulfilled orders O5, O6 and O7 more often, which are of complexity C0, which confirms our hypothesis.

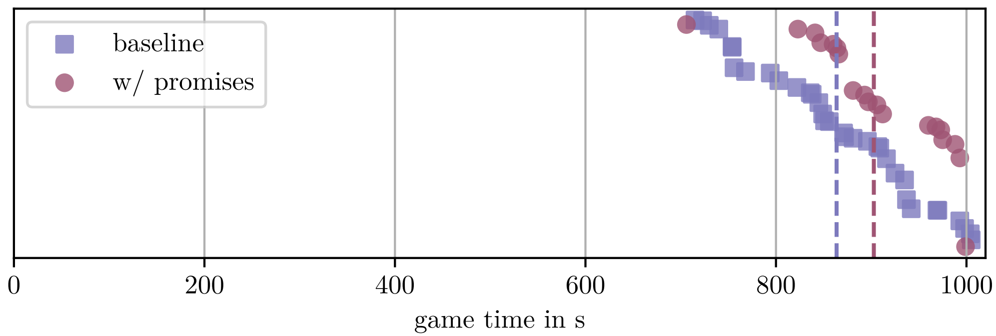
The results for the delivery times, show a similar picture to what we would expect from our first experiment. Figure 5.7 shows that for products of complexity C0, the modified agent tends to have equal or better performance, delivering the product faster in the median case, even when they occur later in the game like O5, while for the more complex cases, it performs worse. When we analyze the games of the modified agent where no C2 or C3 product was delivered in more detail, we can see the same effects



(a) Delivery times for order O1.



(b) Delivery times for order O5.



(c) Delivery times for order O4.

Figure 5.7: Delivery times for order O1 (C0), O5 (C0) and O4 (C3) products plotted over the game time on the x-axis, sorted by their time on the y-axis. Red dots represent our extension, purple squares represent the baseline. The vertical lines mark the mean for the sets.

Table 5.7: The delivery numbers for each order over 50 runs per agent with additional order descriptions. An early-game product has a delivery window starting in the first third of the game, a mid-game product starts in the second third, while a late-game product is expected to be delivered starting in the third third. There are no results for O2 and O8, because C1 products were disabled. The delivery numbers are filtered for multiple deliveries (were more than the ordered number of products were delivered), which slightly reduces the numbers of delivered items.

Order	Complexity	Points		Description
		Baseline	w/ Promises	
O1	C0	47	44	early-game product; one order
O2	C1	-	-	early-game product; one order
O3	C2	44	34	late-game product; one order
O4	C3	34	19	late-game product; one order
O5	C0	32	36	mid-game product; one order
O6	C0	24	28	mid-game product; two orders
O7	C0	15	19	late-game product; one order
O8	C1	-	-	mid-game product; one order

that also caused the delays in the first experiment occured in these cases as well, leading to products being finished too late to be delivered.

Figure 5.8 highlights the cooperative and coordination aspects of promises and requests during a full game. It shows several cases of promised produce-C0 goals, which are caused by the promises of a simultaneous fill-cap goal, one example being fill-Cap-gen442 on robot 3 and produce-C0-gen495 on robot 1, in which the agent starts working on the produce goal shortly after the fill-Cap itself was dispatched. Because robot 1 first needs to get the base from the base-station, robot 3 can work on filling the cap-station without interfering with robot 1, allowing it to start working on the goal earlier than it could without promises.

Coordination between agents is also highlighted in Figure 5.8 as we can see with fill-rs-gen583 on robot 1 and fill-rs-gen518 on robot 3. Here, robot 1 could start working early on filling the machine with a base that it was already holding, because of robot 3's promise, resulting in a short succesion of two goals on the same machine, without one needing to wait for the other to finish before it starting.

Discussion

Our results clearly show that our extension does not positively contribute to the overall agent performance in full-game scenarios, because of the worse performance for higher value products, which are more important to the game outcome. We manage to increase performance of the production of C0 products even in a normal game scenario. However, these positive aspects do not manage to offset the negative effects caused by our extension.

We did succeed to introduce new cooperative behavior between the agents, as they are working simultaneously together on the same product. We can also see that the agents

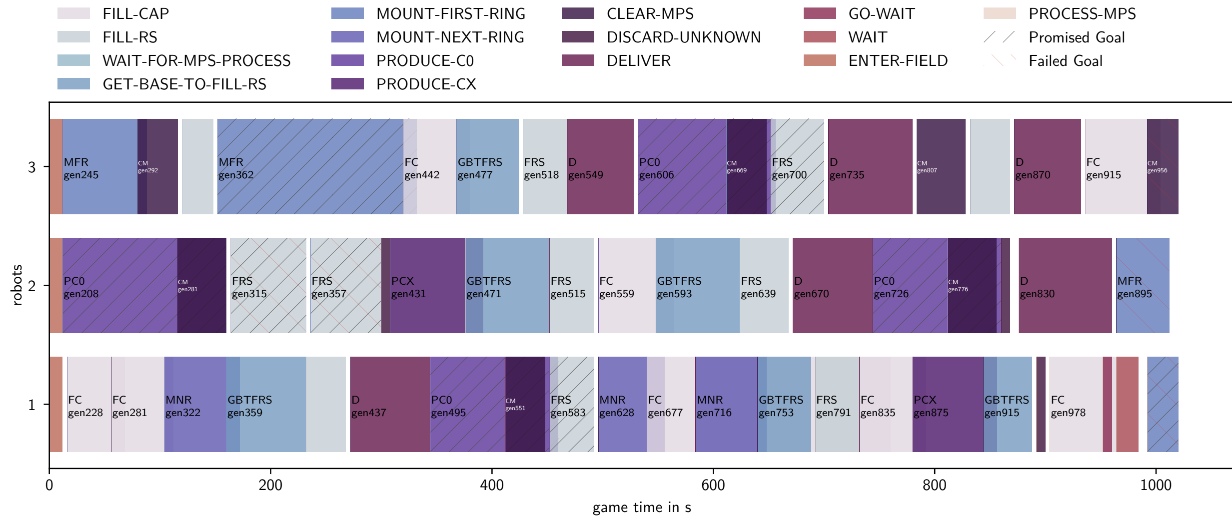


Figure 5.8: Game graph of the modified agent, showing which robot pursued which goal at the given time. We can see multiple cases of promised produce-C0 goals and promised fill-rs goals, which are indicative of the cooperative and coordination benefits of promises and resource requests.

sometimes achieve better utilization of the given machines by starting goals early while a machine is still blocked, leading to lower stand by time of the machine between two goals, in some cases.

6 Conclusion

In this thesis we have introduced two extensions for multi-agent goal reasoning through which agents can reason about near-future world states. We summarize our approach and describe possible improvements in the following section.

6.1 Summary

Promises are the effects that active plans will have on the world state. They represent the intentions of the emitting agent without the need to explicitely model other agents or intentions. Instead, they can be described using semantics similar to normal plan effects. When projected onto the world state, the result represents a near-future state under the assumption that all current operations will be successful. When we reason about the projected world state we can formulate additional goals that are based on future results, giving us access to potentially more valuable goals. Resource requests are a model that allows for co-occupation of symbolic resources to improve parallelization in protected environments. They define a method of acquiring the resources of a promising goal and allow the promised goal to progress in the goal lifecycle.

We used these concepts in an effort to improve cooperation and coordination between agents that perform decentralized, incremental goal reasoning. A formal model building on goal reasoning semantics and STRIPS operators was introduced as the basis of our proposed changes. We extended the model with semantics for promises and resource requests and integrated both into a new formulation procedure for goals on the projected world state.

Based on the formal model, we implemented promises and resource requests as extensions of the CLIPS Executive agent for the RCLL domain. Here, each goal has a fixed plan, consisting of several actions described using PDDL. We manually determined the plan effects for each goal, which will serve as promises, and translated them into CLIPS facts akin to the ones used in the CX to describe the world model. If a goal is dispatched, we share its effects with the other agents and let each dynamically project them onto the domain model, leading to a constantly updated projected world model, which we use to formulate our goals. Once a goal is finished, we remove the promises from the projection.

We integrated resource requests, by determining if a promise was used as a precondition for goal formulation, followed by the creation of a request fact, which guides a multi stage handover process. The requesting goal blocks any other agent from creating requests or acquiring the requested resource. When the promising goal is finished, it signals to the requesting goal, that the resources can now be acquired, completing the

handover procedure and lifting limitations.

The agent was further changed by introducing two additional modifications that support promises and requests. We changed the structure of the CX's goal tree, to bind semantically related goals from two goal trees to combine their promises and increase their lifetime. We also introduced timing to promises, which allows us to reject goals, that fire on promises that are expected too far in the future in an effort to reduce conflicts from promised goals.

We have provided detailed evaluation of promises and resource requests in the context of the RoboCup Logistics League, analyzing how agent behavior is affected and how the modified agent performs in full game scenarios. We have seen that it performs well on simple products, decreasing delivery time significantly. Furthermore, new cooperative behavior is introduced and coordination improved. However, more complex products, suffer from wait times induced by promises and their production times are increased in comparison to the base agent. This also leads to the agent performing significantly worse in full game scenarios.

Summarizing, promises and requests succeed in introducing new cooperative patterns and coordinative behavior in the CX for the RCLL domain, but fail at providing performance improvement for full games compared to the base agent, except for reduced production time for simple goals. This discourages their deployment in the RCLL.

6.2 Future Work

Currently promises and resource requests are implemented specifically for the RCLL domain. As a next step of improving the concept, they might be integrated into the base CX, making integration into different agents for other domains easier, which would allow evaluation of the concept in different settings. Additionally, different restrictions for promises might be tested. We faced the problem where goals that were considered to be less important to the overall production process take more time because of promises. If we restrict promises and use them only as preconditions for goals that we deem more valuable, it might reduce the impact of less important goals and could therefore increase the performance compared to the baseline, as important production steps, which often need to interact with two or more machines would reach higher parallelization. Finally, goal requests, the second form of requests in our domain, might be implemented in combination with a centralized reasoning agent. In our semantics, a goal request could be modelled as the addition of a goal $g \notin G_{\text{form}}(s)$ to a new formulation space $G'_{\text{form}} = G_{\text{form}}(s) \cup g$. The agent would then perform its goal reasoning over the extended space G', selecting the newly added goal g, if it were to be of the highest priority. Using goal requests for task-assignment would require few changes to the actual agents and could be well integrated into our existing formal model.

Bibliography

[1] David W. Aha. “Goal Reasoning: Foundations, Emerging Applications, and Prospects”. en. In: *AI Magazine* 39.2 (July 2018), pp. 3–24. ISSN: 2371-9621, 0738-4602. DOI: 10.1609/aimag.v39i2.2800. URL: https://aaai.org/ojs/index.php/aimagazine/article/view/2800 (visited on 04/15/2020).

[2] L. Barreto, A. Amaral, and T. Pereira. “Industry 4.0 implications in logistics: an overview”. en. In: *Procedia Manufacturing* 13 (2017), pp. 1245–1252. ISSN: 23519789. DOI: 10.1016/j.promfg.2017.09.045. URL: https://linkinghub.elsevier.com/retrieve/pii/S2351978917306807 (visited on 04/14/2020).

[3] Samuel Barrett et al. “Learning Teammate Models for Ad Hoc Teamwork”. en. In: *Proceedings of the 11th International Conference on Autonomous Agents and Multiagent Systems*. 2012.

[4] Samuel Barrett et al. “Teamwork with Limited Knowledge of Teammates”. en. In: *Proceedings of the Twenty-Seventh AAAI Conference on Artificial Intelligence*. 2013, pp. 102–108.

[5] Yongcan Cao et al. “An Overview of Recent Progress in the Study of Distributed Multi-Agent Coordination”. en. In: *IEEE Transactions on Industrial Informatics* 9.1 (Feb. 2013), pp. 427–438. ISSN: 1551-3203, 1941-0050. DOI: 10.1109/TII.2012.2219061. URL: http://ieeexplore.ieee.org/document/6303906/ (visited on 04/14/2020).

[6] Michael T Cox. “A model of planning, action, and interpretation with goal reasoning”. en. In: *Advances in Cognitive Systems 4*. Evanston,Illinois, June 2016.

[7] M.B. Dias et al. “Market-Based Multirobot Coordination: A Survey and Analysis”. en. In: *Proceedings of the IEEE* 94.7 (July 2006), pp. 1257–1270. ISSN: 0018-9219, 1558-2256. DOI: 10.1109/JPROC.2006.876939. URL: http://ieeexplore.ieee.org/document/1677943/ (visited on 04/14/2020).

[8] D. A. Dolgov and E. H. Durfee. “Resource Allocation Among Agents with MDP-Induced Preferences”. en. In: *Journal of Artificial Intelligence Research* 27 (Dec. 2006), pp. 505–549. ISSN: 1076-9757. DOI: 10.1613/jair.2102. URL: https://jair.org/index.php/jair/article/view/10478 (visited on 04/14/2020).

[9] A. Farinelli, L. Iocchi, and D. Nardi. “Multirobot Systems: A Classification Focused on Coordination”. en. In: *IEEE Transactions on Systems, Man and Cybernetics, Part B (Cybernetics)* 34.5 (Oct. 2004), pp. 2015–2028. ISSN: 1083-4419. DOI: 10.1109/TSMCB.2004.832155. URL: http://ieeexplore.ieee.org/document/1335496/ (visited on 04/15/2020).

[10] Richard E. Fikes and Nils J. Nilsson. “Strips: A new approach to the application of theorem proving to problem solving”. en. In: *Artificial Intelligence* 2.3-4 (Dec. 1971), pp. 189–208. issn: 00043702. doi: 10.1016/0004-3702(71)90010-5. url: https://linkinghub.elsevier.com/retrieve/pii/0004370271900105 (visited on 09/24/2020).

[11] John Grant, Sarit Kraus, and Donald Perlis. “A Logic-Based Model of Intentions for Multi-Agent Subcontracting”. en. In: *Proceedings of the Eighteenth National Conference on Artificial Intelligence*. 2002, pp. 320–325.

[12] Martin Hagele. “Robots Conquer the World [Turning Point]”. en. In: *IEEE Robotics & Automation Magazine* 23.1 (Mar. 2016), pp. 120–118. issn: 1070-9932. doi: 10.1109/MRA.2015.2512741. url: http://ieeexplore.ieee.org/document/7426868/ (visited on 04/14/2020).

[13] Andreas Hertle and Bernhard Nebel. “Efficient Auction Based Coordination for Distributed Multi-agent Planning in Temporal Domains Using Resource Abstraction”. en. In: *KI 2018: Advances in Artificial Intelligence*. Ed. by Frank Trollmann and Anni-Yasmin Turhan. Vol. 11117. Series Title: Lecture Notes in Computer Science. Cham: Springer International Publishing, 2018, pp. 86–98. isbn: 978-3-030-00110-0 978-3-030-00111-7. doi: 10.1007/978-3-030-00111-7_8. url: http://link.springer.com/10.1007/978-3-030-00111-7_8 (visited on 04/30/2020).

[14] Till Hofmann, Tim Niemueller, and Gerhard Lakemeyer. “Macro Operator Synthesis for ADL Domains”. en. In: *Proceedings of the 24th European Conference on Artificial Intelligence*. Santiago de Compostela, 2020, p. 8.

[15] Till Hofmann et al. “Multi-Agent Goal Reasoning with the CLIPS Executive in the RoboCup Logistics League”. en. 2020.

[16] Till Hofmann et al. “Winning the RoboCup Logistics League with Fast Navigation, Precise Manipulation, and Robust Goal Reasoning”. en. In: *RoboCup 2019: Robot World Cup XXIII*. Ed. by Stephan Chalup et al. Vol. 11531. Series Title: Lecture Notes in Computer Science. Cham: Springer International Publishing, 2019, pp. 504–516. isbn: 978-3-030-35698-9 978-3-030-35699-6. doi: 10.1007/978-3-030-35699-6_41. url: http://link.springer.com/10.1007/978-3-030-35699-6_41 (visited on 04/15/2020).

[17] Tom Holvoet and Paul Valckenaers. “Beliefs, desires and intentions through the environment”. en. In: *Proceedings of the fifth international joint conference on Autonomous agents and multiagent systems - AAMAS ’06*. Hakodate, Japan: ACM Press, 2006, p. 1052. isbn: 978-1-59593-303-4. doi: 10.1145/1160633.1160820. url: http://portal.acm.org/citation.cfm?doid=1160633.1160820 (visited on 04/14/2020).

[18] Luca Iocchi et al. “Distributed Coordination in Heterogeneous Multi-Robot Systems”. en. In: *Autonomous Robots* 15.2 (Sept. 2003), pp. 155–168.

[19] Long Jin et al. "Dynamic task allocation in multi-robot coordination for moving target tracking: A distributed approach". en. In: *Automatica* 100 (Feb. 2019), pp. 75–81. ISSN: 00051098. DOI: 10.1016/j.automatica.2018.11.001. URL: https://linkinghub.elsevier.com/retrieve/pii/S0005109818305338 (visited on 04/14/2020).

[20] Jelle R. Kok, Matthijs T.J. Spaan, and Nikos Vlassis. "Non-communicative multi-robot coordination in dynamic environments". en. In: *Robotics and Autonomous Systems* 50.2-3 (Feb. 2005), pp. 99–114. ISSN: 09218890. DOI: 10.1016/j.robot.2004.08.003. URL: https://linkinghub.elsevier.com/retrieve/pii/S0921889004001290 (visited on 04/14/2020).

[21] G. Ayorkor Korsah, Anthony Stentz, and M. Bernardine Dias. "A comprehensive taxonomy for multi-robot task allocation". en. In: *The International Journal of Robotics Research* 32.12 (Oct. 2013), pp. 1495–1512. ISSN: 0278-3649, 1741-3176. DOI: 10.1177/0278364913496484. URL: http://journals.sagepub.com/doi/10.1177/0278364913496484 (visited on 04/14/2020).

[22] Drew McDermott et al. *PDDL - The Planning Domain Definition Language*. Tech. rep. 1.2. Yale Center for Computational Vision and Control, 1998.

[23] Minh Hoai Nguyen and Wayne Wobcke. "A Flexible Framework for SharedPlans". en. In: *AI 2006: Advances in Artificial Intelligence*. Ed. by David Hutchison et al. Vol. 4304. Series Title: Lecture Notes in Computer Science. Berlin, Heidelberg: Springer Berlin Heidelberg, 2006, pp. 393–402. ISBN: 978-3-540-49787-5 978-3-540-49788-2. DOI: 10.1007/11941439_43. URL: http://link.springer.com/10.1007/11941439_43 (visited on 04/14/2020).

[24] Tim Niemueller, Till Hofmann, and Gerhard Lakemeyer. "CLIPS-based Execution for PDDL Planners". en. In: *Workshop on Integrated Planning, Acting, and Execution*. 2018.

[25] Tim Niemueller, Till Hofmann, and Gerhard Lakemeyer. "Goal Reasoning in the CLIPS Executive for Integrated Planning and Execution". en. In: *Proceedings of the Twenty-Ninth International Conference on Automated Planning and Scheduling*. 2019.

[26] Stuart J. Russell, Peter Norvig, and Ernest Davis. *Artificial Intelligence: A Modern Approach*. en. 3rd ed. Prentice Hall series in artificial intelligence. Upper Saddle River: Prentice Hall, 2010. ISBN: 978-0-13-604259-4.

[27] Trevor Santarra and Arnav Jhala. "Adapting Plans through Communication with Unknown Teammates (Doctoral Consortium)". en. In: *Proceedings of the 2016 International Conference on Autonomous Agents & Multiagent Systems*. 2016, pp. 1526–1527.

[28] Trevor Santarra and Arnav Jhala. "Communicating Intentions for Coordination with Unknown Teammates (Extended Abstract)". en. In: *Proceedings of the 2016 International Conference on Autonomous Agents & Multiagent Systems*. 2016, pp. 1423–1424.

[29] Trevor Sarratt and Arnav Jhala. "Policy Communication for Coordination with Unknown Teammates". en. In: *The Workshops of the Thirtieth AAAI Conference on Artificial Intelligence*. 2016, pp. 567–573.

[30] Weiming Shen, Douglas H. Norrie, and J-P. Barthes. *Multi-Agent Systems for Concurrent Intelligent Design and Manufacturing*. Englisch. CRC Press, Sept. 2003. ISBN: 0-203-30560-4 978-0-203-30560-7.

[31] Douglas Vail and Manuela Veloso. "Dynamic Multi-Robot Coordination". en. In: *Multi-Robot Systems: From Swarms to Intelligent Automata*. Vol. 2. 2003, pp. 87–100.

[32] Daniel Wahrmann et al. "An Autonomous and Flexible Robotic Framework for Logistics Applications". en. In: *Journal of Intelligent & Robotic Systems* 93.3-4 (Mar. 2019), pp. 419–431. ISSN: 0921-0296, 1573-0409. DOI: 10.1007/s10846-017-0746-8. URL: http://link.springer.com/10.1007/s10846-017-0746-8 (visited on 04/19/2020).

[33] Robert M. Wygant. "CLIPS — A powerful development and delivery expert system tool". en. In: *Computers & Industrial Engineering* 17.1-4 (Jan. 1989), pp. 546–549. ISSN: 03608352. DOI: 10.1016/0360-8352(89)90121-6. URL: https://linkinghub.elsevier.com/retrieve/pii/0360835289901216 (visited on 05/07/2020).

[34] Zhi Yan, Nicolas Jouandeau, and Arab Ali Cherif. "A Survey and Analysis of Multi-Robot Coordination". en. In: *International Journal of Advanced Robotic Systems* 10.12 (Dec. 2013), p. 399. ISSN: 1729-8814, 1729-8814. DOI: 10.5772/57313. URL: http://journals.sagepub.com/doi/10.5772/57313 (visited on 04/14/2020).