Multi-Agent Goal Reasoning with the CLIPS Executive in the RoboCup Logistics League

Till Hofmann [a], Tarik Viehmann [b], Mostafa Gomaa [c], Daniel Habering [d], Tim Niemueller* [e]
and Gerhard Lakemeyer [f]

Knowledge-based Systems Group, RWTH Aachen University, Germany

Keywords: Goal Reasoning, Multi-Agent Coordination, Execution Monitoring, Smart Factory, RoboCup Logistics League.

Abstract: Production processes in smart factories moved away from a process-centered paradigm into a modular production paradigm, facing the variations in demanded product configurations and deadlines with a flexible production. The RoboCup Logistics League (RCLL) is a robotics competition in the context of in-factory logistics, in which a team of three autonomous mobile robots manufacture dynamically ordered products. The main challenges include task reasoning, multi-agent coordination, and robust execution in a dynamic environment. We present a multi-agent goal reasoning approach where agents continuously reason about which objectives to pursue rather than only planning for a fixed objective. We describe an incremental, distributed formulation of the RCLL problem implemented in the goal reasoning system CLIPS Executive. We elaborate what kind of goals we use in the RCLL, how we use goal trees to define an effective production strategy and how agents coordinate effectively by means of primitive lock actions as well as goal-level resource allocation. The system utilizes a PDDL model to describe domain predicates and actions, as well as to determine the executability and effects of actions during execution. Our agent is able to react to unexpected events, such as a broken machine or a failed action, by monitoring the execution of the plan, re-evaluating goals, and taking over goals which were previously pursued by another robot. We present a detailed evaluation of the system used on real robots.

1 INTRODUCTION

Planning and reasoning in robotics remains a challenging task. A robot acts in a dynamic environment, where the world continuously changes while the agent decides what to do. While long planning times are acceptable in many domains, robots are required to make decisions quickly. Furthermore, robots acts in a physical world, where actions often have unintended effects or fail completely.

The RoboCup Logistics League (RCLL) (Niemueller et al., 2015) is a robotics competition that show-cases those challenges. In the RCLL, a team of three mobile robots has to run a smart factory to manufacture dynamically ordered products. They do so by operating a set of Modular Production System (MPS) at fixed locations. Each MPS can perform a particular step of the production. Challenges include determining the production steps necessary to fulfill an order, coordinating the three robots for effective cooperation, and reacting to unexpected events and failed actions, such as a broken machine or a failed grasping action.

In this paper, we present a multi-agent goal reasoning approach to tackle the RCLL[1]. *Goal reasoning* is "the study of agents that can deliberate on and self-select their objectives" (Aha, 2018), instead of just pursuing a fixed objective. We describe an agent implementation that uses the CLIPS Executive (CX) (Niemueller et al., 2019), an integrated goal reasoning system with an explicit goal representation. The CX implements a goal lifecycle (Roberts et al., 2014) which specifies how the mode of a goal

---

[a] https://orcid.org/0000-0002-8621-5939
[b] https://orcid.org/0000-0003-0264-0055
[c] https://orcid.org/0000-0002-2185-6065
[d] https://orcid.org/0000-0002-4917-3055
[e] https://orcid.org/0000-0001-6827-8648
[f] https://orcid.org/0000-0002-7363-7593
*T. Niemueller is now with X – The Moonshot Factory.

[1] The code is open source and is available at https://www.fawkesrobotics.org/projects/rcll2019-release/.

progresses over time. Instead of pursuing long-term goals, our distributed agent incrementally decides what to do, using fine-grained goals, structured in a goal tree which defines an overall strategy. To make sure that no two conflicting goals are pursued simultaneously, the agents coordinate by means of a shared world model, locking actions, and goal-level resource allocation. PDDL is used for plan representation and as a model for execution monitoring.

Using goal trees of fine-grained goals, a distributed coordination strategy, and a PDDL model for action execution results in a robust system that is able to cope with frequent action failures, unexpected events, and even temporary outage or permanent drop-out of a robot, as demonstrated by an extensive evaluation.

In the following, we start by describing the RCLL and its challenges for a reasoning system in Section 2. In Section 3 we summarize related work in goal reasoning then describe other approaches to the RCLL. In Section 4, we explain how we applied goal reasoning in the RCLL. We demonstrate how the production process in the RCLL can be described in terms of goals, and how those goals can be organized in a goal tree to define an overall strategy. In Section 5, we detail the coordination primitives of the CX and how we use them in the RCLL domain to allow effective multi-agent cooperation. In Section 6, we describe how we monitor plan execution and react to failed actions or unexpected events. Finally, we provide a detailed evaluation of our system on real robots in Section 7, before concluding in Section 8.

2 THE RoboCup LOGISTICS LEAGUE

RoboCup (Kitano et al., 1997) is an international robotics competition with the goal to foster AI and intelligent robotics research and is mostly known for its soccer competitions, although it also provides competitions for domestic service robots and rescue robots. The RoboCup Logistics League (RCLL) (Niemueller et al., 2015) is an in-factory production logistics competition under the RoboCup umbrella. Two teams consisting of three robots each compete against each other on a playing field of size 14 m × 8 m for 17 minutes. The main goal is to manufacture products to fulfill orders, where an order describes the configuration of the requested product, the delivery time window and the requested quantity. Orders are generated by a semi-autonomous game controller, the *referee box (refbox)* (Niemueller et al., 2016b), which also takes care of generating the field layout, workpiece

tracking, and score keeping. Products consist of (a) a base piece with color black, silver, or red; (b) zero to three rings with colors orange, yellow, blue, or green; (c) and a cap, either gray or black; resulting in 243 possible product configurations. The number of rings determines the product complexity, where a `C0` is a product with no ring and minimal complexity, and `C3` is a product with three rings and maximal complexity.

Each team has its exclusive set of Modular Production System (MPS) machines that perform the manufacturing steps, teams do not have to compete for machine usage. However, the machines are pseudo-randomly spread across the field such that the robots cross paths frequently. There are several different types of machines: A *base station* (BS) dispenses bases, a *ring station* (RS) mounts rings, a *cap station* (CS) can buffer and mount caps, and a *delivery station* (DS) is used to deliver the manufactured products.

Figure 1: The production chain for a `C2` (Coelen et al., 2019).

Figure 1 shows an exemplary production of a `C2`. First, robot 1 fetches a silver base piece from the BS, carries it to RS 1 and puts it on the machine's conveyor belt. Then, it instructs the RS to mount a blue ring on the workpiece. At the same time, robot 2 fetches an additional workpiece, either from a CS or the BS, and feeds it into the RS 2 as raw material for the next ring. Robot 1 then carries the workpiece with the silver base and the blue ring to RS 2 and feeds it into the machine, before it instructs the machine to mount the second, green ring. In the meantime, robot 3 prepares the CS by feeding a cap carrier from the shelf into the machine, which then removes the cap from the cap carrier for later usage. Robot 3 then fetches the cap-less base from the output of the CS and discards it. After this is done, one of the robots fetches the workpiece with the two rings, feeds it into the CS and instructs the machine to mount the cap. Finally, a robot carries the finished product to the DS and instructs it to complete the delivery. This example shows that producing a single `C2` requires about 11 steps like mounting a ring. Each step involves several actions, e.g, moving to a machine, putting down the workpiece, and instructing the machine.

As there are nine orders in a regular game, it is not possible to fulfill all orders. This *over-subscription* is one of the major challenges of the competition as the robots have to carefully decide which orders to pursue. Also, the domain contains a number of sources

of uncertainty: For one, the orders are unknown beforehand and machines may turn off during the game for unscheduled maintenance and behave unexpectedly due to handling errors, e.g., a workpiece getting stuck in the machine. Also, action failures are common, as machine handling in particular is a challenging task. As the production of a single product already takes significant time, some of the required steps can be parallelized and multiple orders can be pursued concurrently, multi-agent coordination is crucial to fulfill orders effectively.

3 RELATED WORK

We summarize related work on goal reasoning and describe a number of different reasoning approaches for the RCLL.

3.1 Goal Reasoning

Goal reasoning is "the study of agents that can deliberate on and self-select their objectives" (Aha, 2018). A goal-reasoning agent not only reasons about its actions to accomplish a fixed goal, but also reasons about which goals to pursue. Goal refinement represents the context in which a goal is pursued by a goal-reasoning agent. The goal lifecycle (Roberts et al., 2014) is such a goal refinement and specifies how the mode of a goal progresses over time. Its semantics are defined in terms of goal-task network (GTN) planning (Alford et al., 2016), which combines hierarchical task networks (HTNs) with hierarchical goal networks (HGNs) (Shivashankar et al., 2012). The goal lifecycle has been implemented in *ActorSim* (Roberts et al., 2016a,b), a general platform for research on autonomy in simulated environments. It provides a goal refinement library with goal management and implementations for goals, goal types, and goal refinement strategies. Goal reasoning has been applied in various domains, e.g., Minecraft (Abel et al., 2015), underwater vehicles (Wilson et al., 2018), and most closely related to our application, to coordinate a team of robots for disaster relief (Roberts et al., 2015).

3.2 Reasoning in the RCLL

We present other approaches that were successfully used in the Planning and Execution Competition for Logistics Robots In Simulation (PExC) (Niemueller et al., 2016a) or in the RCLL. The PExC has the same rules as the RCLL but only runs in simulation. Thus,

in contrast to our approach, those approaches were not used on real robots.

3.2.1 ASP

Schäpers et al. (2018) encoded the RCLL in answer set programming (ASP) to implement a centralized, global planner. Their system plans continuously while the game is running. They use a coarse representation in terms of compound tasks, rather than actions. They compute a temporal plan with discretized time intervals of 10 s and a planning horizon of 180 s. By using the multi-shot solver *clingo* (Gebser et al., 2019), their system can re-plan without regrounding the whole program. They have successfully used their system in the PExC 2018.

3.2.2 OpenPRS

The Procedural Reasoning System (PRS) is a high-level control and supervision framework to represent and execute plans and procedures in a dynamic environment (Ingrand et al., 1996). It is based on the belief-desire-intention (BDI) model (Bratman, 1987).

The RoboCup team Carologistics developed a centralized reasoning approach extending on (Niemueller et al., 2017b). All tasks required to produce a chosen order are intended in parallel, as roots of the intention graph, and sleep until their preconditions are satisfied. Consequently, each intention monitors the execution of a sequence of location transition goals performed by an allocated actor to a workpiece. Their approach scored the second position in PExC 2017.

The RoboCup team GRIPS uses a combination of HTNs and OpenPRS for a centralized reasoning agent in the RCLL (Ulz et al., 2019). A central global HTN planner decomposes each order into a set of tasks, which are then dispatched incrementally to the robots using a request-response mechanism. The central planner uses a coarse task representation, which is then refined locally on the robot with OpenPRS. GRIPS was able to win the RCLL World Cup 2018 with their approach.

3.2.3 OMT-based planning

Optimization modulo theories (OMT) extends satisfiability modulo theories (SMT) solving with optimization functionalities. Leofante et al. (2019) model the RCLL as state-based planning problem by representing the domain with mixed-integer arithmetic formulas with an initial state and a transition relation. The transition relation is determined by the agent's actions and defines possible execution paths,

where each transition has an associated reward. They then optimize the execution path to maximize the reward. Their approach won the PExC 2018.

3.2.4 ROSPlan

ROSPlan (Cashmore et al., 2015) is a framework for task planning in the Robot Operating System (ROS). It provides ROS nodes for the knowledge base, which is responsible for knowledge gathering and generating the initial situation for the planner, and the planning system, which takes care of invoking the planner, as well validating and dispatching the resulting plan. ROSPlan has been used in the PExC in conjunction with the partial-order planner POPF (Coles et al., 2010).

3.2.5 Auction-based planning

Hertle and Nebel (2018) describe a distributed approach using an auction mechanism with temporal planning based on ROSPlan. Based on a task-level domain description, a central auctioneer determines the tasks that needs to be done and offers them to the three robot agents. Each agent computes a temporal plan for the open tasks and bids on a task with the start and end time of the plan. Based on the bids, the auctioneer determines which robot should perform which task and distributes them to the robots. This auction-based approach won the PExC 2017.

3.2.6 CLIPS Agent

Niemueller et al. (2013) implemented an agent for the RCLL in the rule-based production system CLIPS. In contrast to the CX, it does not provide an explicit task specification language or an explicit goal representation, but instead directly uses the rule-based system to decide which action to execute next, whenever the robot is idle. The system has been used successfully in the RCLL (Niemueller et al., 2017a; Hofmann et al., 2018).

4 GOAL REASONING IN THE RCLL

In goal reasoning, an agent reasons about which objectives to pursue rather than only planning for a fixed objective. The CLIPS Executive (CX) (Niemueller et al., 2019) is an integrated goal reasoning system that provides an explicit goal representation, implements a goal lifecycle, and structures goals in trees. In the following, we describe the core functionalities
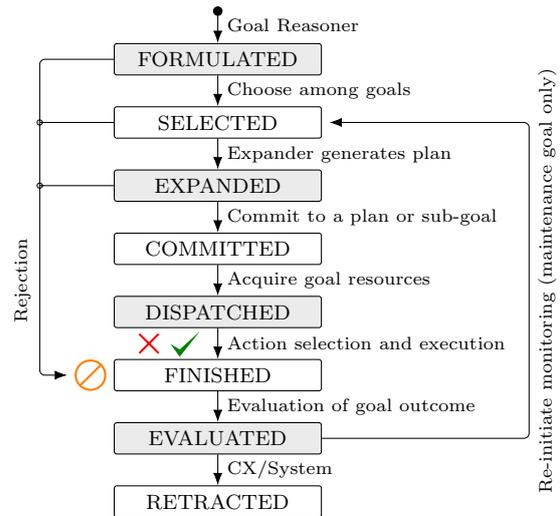

Figure 2: The goal lifecycle of the CLIPS Executive (Niemueller et al., 2019).

of the CX and how we applied goal reasoning to the RCLL. We start by presenting the goal lifecycle and the simple goals that we use in the RCLL. We continue with goal trees and how we use them to define a production strategy in the RCLL. Finally, we explain how we use a plan library to expand a goal into a plan.

4.1 Goals and the Goal Lifecycle

A *goal* is an explicit representation of an objective and relevant aspects necessary to reach it. A goal can either *achieve* or *maintain* a condition or state, e.g., mount a cap or keep the shelf filled. A goal is a grounded instance of a certain *class* describing a category of goals, e.g., DELIVER. The goal lifecycle, shown in Figure 2, describes how a goal progresses over time, where the *goal mode* describes the current state of the goal. Initially, the goal reasoner *formulates* a set of goals, meaning that it may be relevant for consideration. The goal reasoner then *selects* one or more goals, e.g., by picking the most promising goal. A selected goal is then *expanded* by generating (possibly multiple) plans. Next, the reasoner *commits* to one plan and acquires necessary goal resources, before it *dispatches* the goal by executing the plan.

4.1.1 Goals in the RCLL

We pursue a distributed incremental strategy, where each agent decides locally which goal to pursue next. We use fine-grained goals, splitting long-term goals such as the production of a single product in multiple smaller goals. This avoids planning for events over a long duration, often leading to sub-optimal or infeasible plans, as events such as new orders, failed actions,
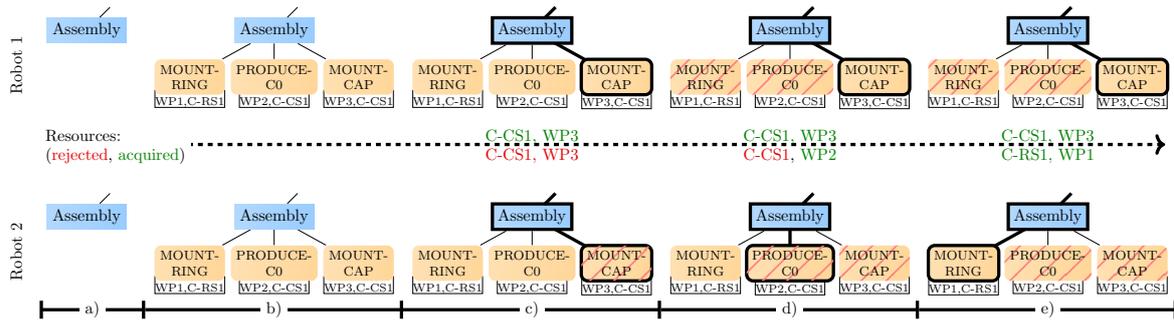
Figure 3: Dispatching a goal from a sub-tree with a *run-one* compound *achieve* goal as root and simple goals as leaves: (a) a compound goal *Assembly* is formulated; (b) *Assembly* is expanded by formulating all currently feasible sub-goals of *Assembly*; (c) *Assembly* is dispatched and committed to MOUNT-CAP, Robot 1 dispatches MOUNT-CAP by acquiring the resources, Robot 2 rejects it as its resource request gets denied; (d) Robot 1 is committed to MOUNT-CAP and rejects all other formulated goals; *Assembly* of Robot 2 commits to PRODUCE-C0, which is rejected due to resource C-CS1 already being locked; (e) Robot 2 commits to MOUNT-RING and acquires resources C-RS1 and WP1.

and the loss of a robot impair the result.

A goal achieves a production step or maintains a condition necessary for sustaining the production, by acting in the physical world or by communicating to instruct a machine. Simple goals were crafted with the following considerations:

1. Robots transport workpieces between machines. When a robot picks up a workpiece, the agent already knows what the workpiece will be used for and where it needs to be taken. There are limited reasonable transportation operations, modeled as the following simple goals:

Purpose	Goal Class	Source	Dest.
Assembly	MOUNT-RING	$BS_{I/O}$, RS_O	RS_I
Assembly	MOUNT-CAP	RS_O	CS_I
Assembly	PRODUCE-C0	$BS_{I/O}$	CS_I
Preparation	FILL-CAP	CS_{Shelf}	CS_I
Preparation	FILL-RS	$BS_{I/O}$, CS_O	RS_{Slide}
Preparation	CLEAR-MPS	Any MPS_O	-
Completion	DELIVER	CS_O	DS_I

2. A robot can carry a single workpiece at a time. Thus, it can only pursue a single transportation goal at a time.

3. A workpiece may only be fed into a machine if the machine is ready to perform the right manufacturing step, e.g., a CS must have a buffered cap to perform a MOUNT-CAP. A workpiece may not be picked up from the input of a machine.

4. A machine can perform a manufacturing step, e.g., mounting a ring, if a workpiece is at its input and no workpiece is at its output. However, a workpiece may already be placed at the input and the instruction may be sent later.

In case a workpiece cannot be used for anything anymore, e.g., a capless carrier when all ring stations are

already filled, it may be necessary to drop it on the ground to effectively remove it from the game, which is possible using a DISCARD goal.

Machine instructions are universally handled by the PROCESS-MPS goal, where a robot instructs an MPS to start performing a machine operation (e.g., assembly or preparation) and then anticipates the outcome of the operation to update its world model. The outcome is an exogenous event which can be sensed by observing the MPS state changes. Decoupling the machine instruction from transportation goals allows robots to pursue other transportation goals after placing the workpiece at the input of a machine. A drawback is that all robots could end up getting busy executing low priority goals (e.g., FILL-RS), while a machine had finished an assembly, resulting in a high priority goal (e.g., DELIVER). This might delay starting the high priority goal (until a robot becomes available). To remedy this, without forcing robots to physically wait for the completion of machine operation, one robot is assigned a special role so it can only perform goals crucial for production (i.e., starting an order, continuing an order, preparing an empty CS).

Apart from transportation and machine instruction goals, we also use goals to model communication tasks that are not immediately changing the physical world. The SEND-BEACON goal provides the current robot status to the refbox, as required by the rules of the game. EXPIRE-LOCKS provides robustness to multi-agent coordination by cleaning up all the locks held by robots that are not part of the game anymore due to a critical failure. Further, goals can also encapsulate necessary changes to the current world model that are not caused by robot actions. In the RCLL humans refill the supply of raw material at the base stations and the cap station shelves. From a robots point of view a base station essentially has an endless sup-

ply of bases, hence it can simply spawn a corresponding object in its world model whenever needed using a `SPAWN-WP` goal. This effectively allows to model the RCLL as a finite domain. Similarly shelves from cap stations are restocked with cap carriers, modeled via a `REFILL-SHELF` goal.

4.2 Goal Trees

In the CX, goals may be organized in *goal trees* to express some relation among goals. A goal tree recursively consists of *compound goals* and *simple goals*, where a compound goal has one or multiple subgoals, and a simple goal is a leaf of a tree. A compound goal is expanded by creating its sub-goals, a simple goal is expanded by creating one or more plans that accomplish the goal. For a compound goal, some transitions are performed automatically by the CX, e.g., committing to the sub-goal with the highest priority.

There are five types of compound goals: A *run-all* goal runs all its sub-goals; a *try-all* goal runs all sub-goals until at least one sub-goal succeeded, failing sub-goals are ignored unless all sub-goals fail; a *run-one* goal runs the first non-rejected sub-goal and fails if that goal fails; a *retry* goal retries a sub-goal for a specified number of times if it fails; a *timeout* goal runs a sub-goal and fails if the sub-goal does not complete in time.

A compound goal has a similar lifecycle as a simple goal, but with a different semantics. A compound goal is *expanded* by formulating sub-goals. If at least one sub-goal exists, the goal reasoner *commits* to the compound goal, otherwise it is rejected. By selecting a sub-goal, the goal is *dispatched*.

Figure 3 illustrates the interaction of lifecycles from different goals in a tree from a perspective of two different robots with the same world model that both determine a goal to dispatch. A simple leaf goal is chosen to be executed by recursively dispatching the highest priority goal . If a goal cannot be dispatched because required resources cannot be acquired, as detailed later in Section 5, the next best alternative is chosen. If all sub-goals are rejected, a compound goal is rejected as well, enabling backtracking until a leaf goal in a different branch is found.

4.2.1 Goal Trees in the RCLL

In the RCLL, an agent maintains multiple objectives simultaneously. All root goals *maintain* some condition and create *achieve* sub-goals if the condition is not satisfied. Figure 4 shows the six goal trees used by the RCLL agent. The *Maintain Beacon* root goal creates a `SEND-BEACON` sub-goal once per second to

send a status message to the refbox, realizing a heartbeat signal. In a similar way the *Maintain Lock Expiration* root goal periodically triggers the creation of an `EXPIRE-LOCKS` goal to continuously react to possible outages in the multi-agent communication backend. The *Maintain WP Spawning* and *Maintain Shelf Refill* root goals are populated whenever a workpiece is dispensed at the base station or a shelf is emptied, respectively. Hence they realize the changes in the world introduced by humans that replenish the respective stations with materials according to the rules. The *Maintain MPS Handling* root goal manages machine instructions and creates a `PROCESS-MPS` sub-goal whenever some instruction needs to be sent.

The largest goal tree is the production tree with the *Maintain Production* goal as its root. The tree contains all transportation goals and is responsible for driving the production forward. As those goals require the robot to act in the physical world, only one of the tree's leaf nodes can be dispatched simultaneously. We also added goals for when no progress can be made. In that case a robot should drive to a waiting position to clear the paths for other robots. The structure of the tree effectively encodes our strategy. Goals performing assembly steps, for example, are of higher priority compared to goals that only prepare material on the machines. Having a tree structure instead of a flat-ordered sequence of goals enables adapting parts of the tree without impacting unrelated branches. This is particularly helpful to adapt to game rule changes, as the RCLL is continuously improved and extended with new challenges.

The inner goals of the *Maintain Production* tree are *run-one* compound goals, i.e., it runs the first non-rejected sub-goal and the outcome is bound to whether that sub-goal succeeded or failed. If a selected sub-goal is rejected (e.g., because another robot had already claimed a required resource), the next best goal in the same branch is selected. If no leaf goal in that branch can be executed, the selection goes recursively up the tree until an executable leaf goal in a next best branch is reached. The lowest priority branch *No Progress* contains goals that are executed if nothing else can be selected. This guarantees that a simple goal will be executed in each lifecycle of the root goal. As an example, the `GO-WAIT` goal will let the robot drive to some waiting position to keep the idle robot from blocking crucial paths of other robots.

Figure 4 shows all goal trees with all goal classes that can be formulated. While a robot is pursuing no goal, it continuously updates its shared world model and reformulates its goals, while only formulating goals that can actually be pursued. As an example, a `MOUNT-RING` goal is only formulated if there is a
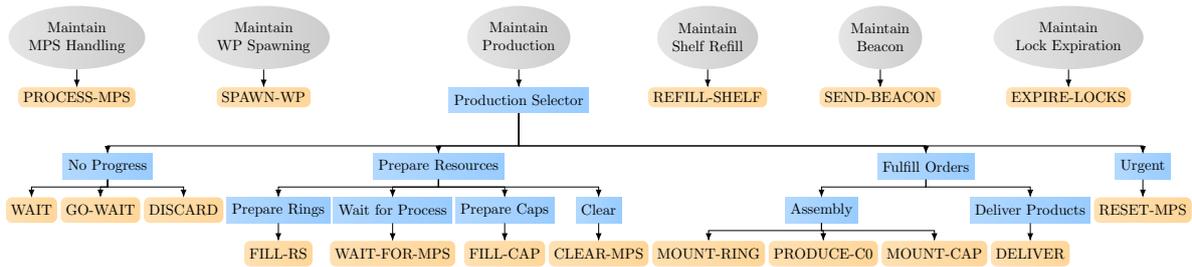
Figure 4: RCLL Goal Trees. The root goals (ellipsis) are *maintenance* goals, the inner sub-goals (blue rectangular) are *run-one* compound *achieve* goals and the leaves (yellow rounded corners) are simple *achieve* goals. The arrangement of the sub-goals conveys the static goal priorities used for goal selection. Priorities increase from left to right.

```
1  (go-wait ?start ?side (wait-pos ?ds INPUT))
2  (location-lock ?ds INPUT)
3  (move (wait-pos ?ds INPUT) ?ds INPUT)
4  (lock ?ds)
5  (wp-put ?wp ?ds)
6  (prepare-ds ?ds ?order)
7  (fulfill-order ?complexity)
8  (unlock ?ds)
9  (location-unlock ?ds INPUT)
10 (go-wait ?ds INPUT (wait-pos ?ds INPUT))
```

Listing 1: The parameterized plan for DELIVER goals.

```
1  (:action wp-put
2   :parameters (?wp - workpiece ?m - mps)
3   :precondition (and (robot-at ?m INPUT)
4   (wp-usable ?wp) (holding ?wp)
5   (mps-side-free ?m INPUT))
6   :effect (and (wp-at ?wp ?m INPUT)
7   (not (holding ?wp)) (can-hold)
8   (not (mps-side-free ?m INPUT))))
```

Listing 2: The action wp-put, which puts a workpiece on a machine, and which is used in the plan in Listing 1.

workpiece that requires an additional ring and the RS has all required materials.

4.3 Goal Expansion

After presenting the structure of goal trees that select appropriate simple goals for execution, we now focus on the goal lifecycle (Figure 2) of each individual simple goal. After a simple goal is selected, it is expanded by determining a suitable plan. The CX supports a PDDL-based *planner model* (Niemueller et al., 2018), where domain predicates, object types and action descriptions are generated from a PDDL description file. As an example, Listing 2 shows the definition of the action wp-put, which puts a workpiece into a machine.

In the RCLL, we do not use a planner, but instead rely on a plan library based on a PDDL model of the RCLL. By doing so, we avoid run-time overhead,

as we do not need to wait for the planner to determine a plan. Also, using a hand-crafted plan library provides more fine-grained control of the dispatched plans. While this has the disadvantages that we need to maintain the plan library, we circumvent those issues by using goals which require only short plans with a few actions. Also, as goal expansion is done by the goal reasoner, we can make use of conditional plans. As an example, the plan shown in Listing 1 is one possible branch of the plan for DELIVER; if the robot is not yet holding the workpiece to be delivered, the plan is prepended with a sub-plan that fetches the workpiece first. With these methods, we obtain plans without unexpected actions, while keeping the maintenance of the plan library feasible, even for such a complex domain as the RCLL. Nevertheless, we are considering to (partially) replace the plan library with a PDDL planner in the future to be able to react to unforeseen situations.

The CX supports both sequential and temporal plans. In the RCLL, we only utilize the former. However, we may dispatch multiple goals and thus execute multiple plans concurrently, as long as they do not interfere.

5 MULTI-AGENT COORDINATION

When a robot determines a plan to execute, it has to coordinate with the other agents to avoid conflicts, e.g., they must not use the same machine at the same time. The CX allows multi-agent coordination by providing functionality for a *shared world model* and *mutual exclusion*, the latter is used to implement *locking actions* and *goal resource allocation*. We introduce those concepts and explain how we apply them in the RCLL.

5.1 Shared World Model

To cooperate effectively in a multi-agent setting, agents need to share (parts of) their world model. To do this, the CX shares the world model by using a shared database. The database is set up as a replica set, where each robot runs one instance of the replica set. The replica set manages data distribution, shared updates, and conflict resolution. The agent's world model is mirrored asynchronously to the database, which allows the agent to continue operation, even in the event of a temporary network outage, while guaranteeing eventual consistency. Updates by the other agents are first replicated on the local database instance and then propagated to the agent's world model.

In the RCLL, we share world model facts that describe (a) the configuration and position of a workpiece; (b) the state of a machine, e.g., for the CS, the free spots on the shelf and the color of the buffered cap (if any); (c) the state of an order, e.g., if the order has been started already, and the workpiece that has been assigned to the order. By sharing those world model facts, each agent has all necessary information to decide which goal to pursue next. However, an agent does not know what the other agents do. Instead, we use locking actions and goal resource allocation based on a mutual exclusion mechanism to implement an effective multi-agent strategy.

5.2 Mutual Exclusion

In addition to a shared world model, the CX also provides functionality for coordination. As the primitive principle, it provides mechanisms for mutual exclusion by using the replicated database with a distinguished collection for mutexes. Whenever an agent tries to acquire a mutex, it writes to the database with *majority acknowledgement*, i.e., a majority of the agents have to agree to the update. This way, only one agent can hold the mutex at any point in time. In order to cope with robots that unexpectedly fail, e.g., because of empty batteries, we implemented a *mutex expiration mechanism*. Each agent maintains a distinguished `EXPIRE-LOCKS` goal that expires all mutexes that have been acquired more than 30 s ago. To avoid early mutex expiration, each agent refreshes its mutexes every 5 s by updating the timestamp in the database.

5.3 Locking Actions

Using the mutual exclusion mechanism, primitive `lock` and `unlock` actions were implemented, which

temporarily lock an object that the agent needs exclusive access to. The executor of a locking action requests a mutex for the given object. The action succeeds once the mutex is acquired and fails if the mutex is rejected, e.g., because it is being held by another agent. The execution of *unlocking* works analogously.

The example in Listing 1 demonstrates delivering a product at the DS. Before operating the DS, the robot locks it to prevent simultaneous access of other robots to the MPS.

Another application of mutexes as primitive actions are `location-lock` and `location-unlock` actions, implementing *location-based locking*, similar to the ideas of (Niemueller et al., 2017b). For a location lock, the mutex is released only when the robot is physically far enough from the object, i.e, 0.5 m away. They are used to ensure that no two robots intend to go to the same location simultaneously. The example in Listing 1 demonstrates the usage of location locks. Before moving to the DS (in line 3), the location is locked (in line 2). Later on, the `location-unlock` action (in line 9) succeeds immediately, yet it will asynchronously ensure that the location is only unlocked when the robot has moved away. To resolve potential deadlocks (e.g., if two robots want to swap positions), robots navigate to a waiting position before trying to lock their destination (in line 1).

5.4 Goal Resource Allocation

Apart from primitive *locking* actions, we also use the mutual exclusion mechanisms to coordinate goal execution among the agents. Each goal has a set of associated *resources*, which must be held during its whole execution. This is done in the CX by requesting a mutex for each specified resource of a committed goal. The goal is only dispatched if all required resources have been acquired, otherwise it will be *rejected*. After the goal has finished, acquired resources are automatically released. Thus, in contrast to `lock` actions, which only lock a resource for parts of the plan, a resource is exclusively assigned to one agent throughout the execution of the respective goal.

Generally, a goal needs to require an object as resource if it changes the state of that object, as this may conflict with other goals. In the RCLL, the main source of conflicts occur when robots operate the same (a) machines, (b) orders, and (c) workpieces. Thus, every goal that operates on any of those needs to acquire the respective resources. The design outlined in Section 4.1.1 implies that transportation goals will eventually lead to a machine instruction at the destination MPS. Each assembly step changes the properties of the destination MPS and the workpiece; the former

by consuming available material, the latter by changing the location and possibly the assembly progress. Therefore, workpieces and destination machines have to be locked as resources. In a delivery step on the other hand, the destination machine is not affected by the processing of a placed workpiece. For this reason, it is not necessary to lock it as resource. In fact, by not doing so and instead only locking it with a lock action, it is possible for two robots to each dispatch a delivery goal simultaneously, e.g., because products were finished by each of the two CS around the same time.

6 ROBUST EXECUTION

Once a simple goal committed to a plan and acquired all required resources, the goal is dispatched and the plan execution starts, involving the call to the system's action executors, monitoring of the execution and evaluation of the results. The approach to deal with failures and unexpected events is two-fold in the CX. On the one hand an agent may re-evaluate the actions scheduled to reach the current goal. This is called *execution monitoring*. On the other hand it can be necessary to re-evaluate the world model, since the failed action may have unexpected effects on the world.

6.1 Action Execution

The CX utilizes the PDDL domain description to control plan execution. The *action selection* determines the action to execute next, marks it as *pending* and only passes it to the executor once all its preconditions are met. This ensures that during execution the planner model and the actual world model do not diverge. The CX supports multiple executors, e.g., it can execute physical actions by passing the action to the system's Lua-based Behavior Engine (Niemueller et al., 2010), while a different executor handles MPS communication tasks. After an action has been executed, its effects are applied to the agent's world model. The CX also supports *sensed effects*: instead of blindly applying an effect the CX waits until the sensed effect has been observed, before the remaining effects are applied and the action is marked as *final*. In the RCLL, we make use of sensed effects to model machine interactions. After sending an instruction, the agent waits for a status update from the refbox and only then applies the action's effects, e.g., a workpiece going from the input to the output of the machine.

6.2 Execution Monitoring

Occasionally, the execution of an action may fail or the world may change in an unexpected way. In these cases, the agent has to decide how to continue with the execution of the current plan. Execution monitoring has three different ways to react on a failure or unexpected change. It can retry a failed action, adapt the planned action in hindsight of a unexpected change or abort the current action and/or goal.

Retrying. If an action failure occurs, the execution monitoring may decide to retry that action using information about the number of previous fails and the cause of failure, provided by the executor, to aid the decision. For example, a failed alignment to the conveyor belt should be retried, whereas a failed grabbing often knocks down the workpiece and therefore should not be retried.

Plan Adaptation. Unexpected changes to the world model may require an agent to change the planned sequence of actions. For example, the agent may try to retrieve a workpiece from the BS, which is accessible from both sides. If during execution the desired side is already occupied by another robot, the agent can switch to the other side.

Aborting. Since the agent will wait for all preconditions of an action to be fulfilled before starting the execution, the action can get stuck upon waiting for an unsatisfied precondition. Those failures are typically caused by network issues causing a temporary inconsistent world model or because an exogenous event did not convey the anticipated effects, e.g., an MPS instruction may have been marked as successful but the machine failed to complete the requested operation, resulting in an unexpected state. Similarly, sensed effects the agent is waiting for may never be observed. In order to deal with these kinds of situations, the execution monitoring maintains timeouts for those action states. As soon as a timeout is reached, the action is aborted.

Also, a goal that utilizes a certain machine may become infeasible when the state of the machine changes unexpectedly during the pursuit of the goal. This mainly happens due to workpieces being misplaced on the conveyor belt or a MPS failure during assembly. In these cases, the machine will end up being temporarily broken. Goals that rely on such a broken machine are aborted by the execution monitoring.

6.3 World Model Re-evaluation

Some failures may have additional undesired effects, e.g., a failed `wp-put` may leave a workpiece some-

where on the machine. We deal with those effects either during the *evaluation* of a failed goal (e.g., removing the workpiece after a failed `wp-put` action), or by reacting on exogenous events (e.g., when a machine breaks due to mishandling, all workpieces and materials buffered at the MPS are removed).

6.4 Maintenance and Reinsertion

In the unfortunate scenario where a robot fails to stay operational, a human operator may decide to temporarily remove it from the game for maintenance. The *lock expiration*, as described in Section 5.2, ensures that the locks held by a malfunctioning agent are released eventually. This allows other agents to take over tasks and/or resources that were previously allocated to the suspended agent. When the maintenance time is over, the human operator may decide to reinsert the maintained agent into the game. On startup, the agent senses that a game has already been running and retrieves all world model facts from other agents.

7 EVALUATION

We evaluate the performance of our approach based on both competitive and test games, providing data from three robots that accumulated a total running time of about 36 hours.

Game Level. Figure 5 shows the production time of successfully delivered products. Deliveries of `C0` products take mostly between 2.5 min and 4 min. Most of the time, the production of `C1` products was disabled as strategic decision to reserve the RS for more complex orders, so the low delivery number was expected. This payed off by yielding six delivered `C2` products.
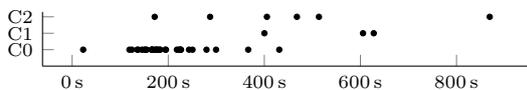
Figure 5: Time of successfully delivered workpieces, accumulated from 33 `C0`, 3 `C1` and 6 `C2` deliveries.

Goal Level. Figure 6 shows statistics on all simple goals from the production tree. It can be seen that roughly 28 % of the total time was spent on goals that eventually failed. This emphasizes the importance of robust execution that recovers from those failures.

Figure 7 depicts a schedule of all dispatched goals throughout our most successful competitive RCLL game, where two `C0` and one `C2` were delivered and

a `C3` was almost finished. One robot had to be reinserted after a major failure and the other two robots became inoperable shortly after that. We chose to not reinsert them into the game as there was not enough time to finish another order. Not having them obstruct paths of the last robot made it more likely to finish the production in progress. In this particular game, the third robot delivered the `C2` product right before the end of the game.

goal class	FIN	FAIL	TIME(FIN)	TIME(FAIL)	failure rate	avg(TIME)
FILL-CAP	108	113	7152.17	3474.16	0.51	48.48
WAIT-FOR-MPS	1700	309	4720.39	317.3	0.15	1.9
CLEAR-MPS	77	69	3119.99	1779.71	0.47	33.16
RESET-MPS	63	3	155.81	60.05	0.05	11.24
MOUNT-FIRST-RING	61	133	5741.08	3152.51	0.69	58.91
MOUNT-NEXT-RING	24	13	1944.87	606.05	0.35	63.83
MOUNT-CAP	17	12	1364.07	533.24	0.41	62.34
PRODUCE-C0	43	32	3633.44	1832.41	0.43	70.88
DELIVER	38	14	3009.83	960.09	0.27	73.89
WAIT	337	0	817.69	0	0	2.43
GO-WAIT	47	371	454.33	200.48	0.89	5.1
DISCARD	39	6	324.61	92.03	0.13	11.83
FILL-RS	179	88	8576.59	3326.4	0.33	43.02
TOTAL	2733	1163	41014.87	16334.43		

Figure 6: Statistics on executed simple goals from the production tree. MOUNT-RING is split into distinct goals depending on the type of source MPS. Times in seconds.
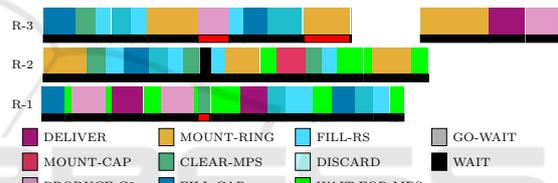
Figure 7: Goal schedule of a full 17 minute RCLL game. The bottom bar of each schedule indicates the goal outcomes, black for completed and red for failed goals.

Action Level. Looking into more detail on the failures on action level, we can distinguish three task types that significantly differ from each other in the failure rate: (a) Moving operations, where the failure rate is well below 10 %. Failures are mainly caused by temporarily unreachable positions. (b) Conveyor belt operations and discards, with a failure rate around 15 %. Failures on any picking or putting action may be caused by alignment issues, decalibrated gripper axes, or failures to detect the object in the gripper. (c) CS shelf and RS slide operations with more than 35 % failure rate. The CS and RS shelves are harder to accurately detect due to their shape and position on the respective machines.

Comparing the duration of action execution with the dispatched simple goal durations may give an idea of the occurring overhead such as synchronizing the world model, handling of locks, and execution monitoring that happen in between actions. Evaluating the total times on failed and successfully executed goals from Figure 6 against the subsumed action execution times it can be seen that around 77 % of the goal execution time was spent on running actions.
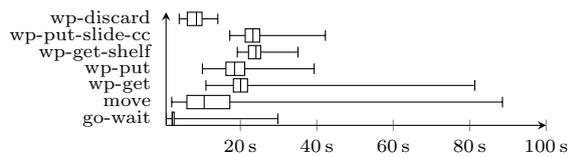
Figure 8: Boxplots of successfully executed action timings.

Execution Monitoring. We first consider retrying failed actions. There were 134 plan executions that contained at least one action retry; 22 % of those resolved the problem such that the respective goal was successful. Considering that action failures can have a severe impact on a game, e.g., it may lead to workpieces getting lost, every successful recovery may significantly increase the overall performance.

Execution monitoring also aborted 422 actions that were stuck, because preconditions were not met until a timeout threshold was reached. 83 % of those stuck actions concern machine assembly steps, indicating a malfunctioning MPS or a misplaced workpiece.

Game Score. On average, we scored 123 points per game, compared to an average of 110 points of the top three teams at RoboCup 2019. Comparing the medians results in 135 versus 104 points. The relative deviation of the score per game of our approach was 43 % compared to 51 % of the top three teams. Thus, our approach was both more successful and more robust than the average of the top three teams at RoboCup 2019.

8 CONCLUSION

In this paper, we have described a multi-agent goal reasoning approach to the RoboCup Logistics League (RCLL), a smart factory scenario in which a team of robots has to manufacture dynamically ordered products. The system is based on the CLIPS Executive (CX) with an explicit goal representation, a goal lifecycle, and the ability to coordinate a team of robots. We have detailed how we designed goals that accomplish steps of the RCLL production process and how we structured those goals in goal trees to define an overall production strategy. We described a distributed multi-agent strategy that uses a shared world model, locking actions, and goal resource allocation to coordinate a team of robots. Goals are expanded into a PDDL plan by using a plan library. To accomplish robust execution, we monitor the execution and react to failed actions either by re-trying the action, adapting the plan, or aborting the goal. We have

evaluated our system by analyzing RCLL games in a competitive scenario. The evaluation shows that even though the system frequently has to deal with failed actions and goals, it is able to effectively control a team of three robots and contributed to the success of the team *Carologistics* at RoboCup 2019.

ACKNOWLEDGEMENTS

REFERENCES

Abel, D., Hershkowitz, D., Barth-Maron, G., Brawner, S., O'Farrell, K., MacGlashan, J., and Tellex, S. (2015). Goal-based action priors. In *Proceedings of the 25th International Conference on Automated Planning and Scheduling (ICAPS)*.

Aha, D. W. (2018). Goal Reasoning: Foundations, Emerging Applications, and Prospects. *AI Magazine*, 39(2):3–24.

Alford, R., Shivashankar, V., Roberts, M., Frank, J., and Aha, D. W. (2016). Hierarchical Planning: Relating Task and Goal Decomposition with Task Sharing. In *Proceedings of the 25th International Joint Conference on Artificial Intelligence (IJCAI)*.

Bratman, M. (1987). *Intention, Plans, and Practical Reason*. Center for the Study of Language and Information.

Cashmore, M., Fox, M., Long, D., Magazzeni, D., Ridder, B., Carrera, A., Palomeras, N., Hurtos, N., and Carreras, M. (2015). ROSPlan: Planning in the Robot Operating System. In *Proceedings of the 25th International Conference on Automated Planning and Scheduling (ICAPS)*, pages 333–341.

Coelen, V., Deppe, C., Gomaa, M., Hofmann, T., Karras, U., Niemueller, T., Rohr, A., and Ulz, T. (2019). The RoboCup Logistics League Rulebook for 2019.

Coles, A., Coles, A., Fox, M., and Long, D. (2010). Forward-chaining Partial-order Planning. In *Proceedings of the 20th International Conference on International Conference on Automated Planning and Scheduling (ICAPS)*, pages 42–49. AAAI Press.

Gebser, M., Kaminski, R., Kaufmann, B., and Schaub, T. (2019). Multi-shot ASP solving with clingo. *Theory and Practice of Logic Programming*, 19(1):27–82.

Hertle, A. and Nebel, B. (2018). Efficient auction based coordination for distributed multi-agent planning in temporal domains using resource abstraction. In *Proceedings of the 41st German Conference on Artificial Intelligence (KI)*.

Hofmann, T., Mataré, V., Neumann, T., Schönitz, S., Henke, C., Limpert, N., Niemueller, T., Ferrein, A., Jeschke, S., and Lakemeyer, G. (2018). Enhancing Software and Hardware Reliability for a Successful Participation in the RoboCup Logistics League 2017. In *RoboCup 2017: Robot World Cup XXI*, pages 486–497. Springer International Publishing.

Ingrand, F., Chatila, R., Alami, R., and Robert, F. (1996). PRS: A High Level Supervision and Control Language for Autonomous Mobile Robots. In *Proceedings of the IEEE International Conference on Robotics and Automation*.

Kitano, H., Asada, M., Kuniyoshi, Y., Noda, I., and Osawa, E. (1997). RoboCup: The Robot World Cup Initiative. In *Proceedings of the 1st International Conference on Autonomous Agents*.

Leofante, F., Ábrahám, E., Niemueller, T., Lakemeyer, G., and Tacchella, A. (2019). Integrated Synthesis and Execution of Optimal Plans for Multi-Robot Systems in Logistics. *Information Systems Frontiers*, 21(1):87–107.

Niemueller, T., Ferrein, A., and Lakemeyer, G. (2010). A Lua-based behavior engine for controlling the humanoid robot Nao. In *RoboCup 2009: Robot Soccer World Cup XIII*.

Niemueller, T., Hofmann, T., and Lakemeyer, G. (2018). CLIPS-based Execution for PDDL Planners. In *2nd ICAPS Workshop on Integrated Planning, Acting, and Execution (IntEx)*.

Niemueller, T., Hofmann, T., and Lakemeyer, G. (2019). Goal reasoning in the CLIPS Executive for integrated planning and execution. In *Proceedings of the 29th International Conference on Automated Planning and Scheduling (ICAPS)*, pages 754–763.

Niemueller, T., Karpas, E., Vaquero, T., and Timmons, E. (2016a). Planning competition for logistics robots in simulation. In *4th ICAPS Workshop on Planning and Robotics (PlanRob)*.

Niemueller, T., Lakemeyer, G., and Ferrein, A. (2013). Incremental task-level reasoning in a competitive factory automation scenario. In *AAAI Spring Symposium on Designing Intelligent Robots*.

Niemueller, T., Lakemeyer, G., and Ferrein, A. (2015). The RoboCup Logistics League as a Benchmark for Planning in Robotics. In *2nd ICAPS Workshop on Planning in Robotics (PlanRob)*.

Niemueller, T., Neumann, T., Henke, C., Schönitz, S., Reuter, S., Ferrein, A., Jeschke, S., and Lakemeyer, G. (2017a). Improvements for a Robust Production in the RoboCup Logistics League 2016. In *RoboCup 2016: Robot World Cup XX*, pages 589–600. Springer International Publishing.

Niemueller, T., Zug, S., Schneider, S., and Karras, U. (2016b). Knowledge-Based Instrumentation and Control for Competitive Industry-Inspired Robotic Domains. *KI - Künstliche Intelligenz*, 30(3):289–299.

Niemueller, T., Zwilling, F., Lakemeyer, G., Löbach, M., Reuter, S., Jeschke, S., and Ferrein, A. (2017b). Cyber-Physical System Intelligence. In *Industrial Internet of Things: Cybermanufacturing Systems*, pages 447–472. Springer International Publishing.

Roberts, M., Apker, T., Johnson, B., Auslander, B., Wellman, B., and Aha, D. W. (2015). Coordinating robot teams for disaster relief. In *Proceedings of the 28th Florida Artificial Intelligence Research Society Conference*. AAAI Press.

Roberts, M., Hiatt, L., Coman, A., Choi, D., Johnson, B., and Aha, D. (2016a). ActorSim, A Toolkit for Studying Cross-Disciplinary Challenges in Autonomy. In *AAAI Fall Symposium on Cross-Disciplinary Challenges for Autonomous Systems*.

Roberts, M., Shivashankar, V., Alford, R., Leece, M., Gupta, S., and Aha, D. W. (2016b). Goal Reasoning, Planning, and Acting with ActorSim, The Actor Simulator. In *Proceedings of the 4th Annual Conference on Advances in Cognitive Systems*, volume 4.

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

Schäpers, B., Niemueller, T., and Lakemeyer, G. (2018). ASP-based time-bounded planning for logistics robots. In *Proceedings of the 28th International Conference on Automated Planning and Scheduling (ICAPS)*.

Shivashankar, V., Kuter, U., Nau, D., and Alford, R. (2012). A Hierarchical Goal-based Formalism and Algorithm for Single-agent Planning. In *Proceedings of the 11th International Conference on Autonomous Agents and Multiagent Systems*, pages 981–988. International Foundation for Autonomous Agents and Multiagent Systems.

Ulz, T., Ludwiger, J., and Steinbauer, G. (2019). A Robust and Flexible System Architecture for Facing the RoboCup Logistics League Challenge. In *RoboCup 2018: Robot World Cup XXII*, pages 488–499. Springer International Publishing.

Wilson, M. A., McMahon, J., Wolek, A., Aha, D. W., and Houston, B. H. (2018). Goal reasoning for autonomous underwater vehicles: Responding to unexpected agents. *AI Communications*, 31(2):151–166.