RoboCup Logistics League
Graz Robust and Intelligent Production System GRIPS

Leo Fürbaß, Peter Kohout, Marco De Bortoli,
Stefan Moser, Gerald Steinbauer-Wagner, Anna Masiero,
Jan Tscherko, Martin Nagele, David Beikircher

Graz University of Technology

Abstract. The present paper presents team GRIPS (Graz Robust and Intelligent Production System) and its approaches to the challenges in the RoboCup Logistics League 2023. Festo's Robotino, a customized additional construction and some further hardware, such as laser scanners, cameras, PC, and PLC, are used as a hardware platform. The software architecture is built in a multi-layer format. A scheduler/planner functions as a central decision-making instance for all robots at the top player. Beneath this layer, various BehaviourTrees are used for individual decisions during an action. At the lowest level, the robot operating system (ROS) executes the tasks assigned to the robot by the top layer.

1 Introduction

The main aim of the RoboCup Logistics League is to promote research in automated and flexible production and serves as a testbed for methods dealing with smart production. For this purpose, the league's competition simulates a smart factory in which different products must be manufactured in several steps at different production machines. During production, the various machines must be supplied with blanks and resources required for product manufacturing. A so-called RefBox orders products in a random order and thus simulates demands for products. All transportation tasks for products and intermediate products are performed by a fleet of autonomous robots. As this setting involves continuous changing of the product demands, the autonomous robots simulate the logistics challenges in future production systems. Therefore, the RoboCup Logistics League is an ideal testbed for innovative planning/scheduling algorithms and control methods for fleets of robots.

Team GRIPS was founded in 2015 as part of the practical course "Construction of Mobile Robots" held at Graz University of Technology. Students in this class have to find solutions for different challenges relevant to the Robocup Logistics League. When GRIPS was first founded, it already participated in the RoboCup World Cup which in 2016 was held in Leipzig, Germany (team description paper [1]). GRIPS finished in third place in its first year of participation and was thus voted rookie of the year. At the RoboCup 2017 held in Nagoya, GRIPS achieved second place. In 2018, GRIPS has already participated at the RoboCup German Open in Magdeburg, Germany and finished in second place. After winning the thrilling finals at the RoboCup 2018 in Montreal, Canada, GRIPS earned its first world champion title. At the RoboCup 2019 in Sydney GRIPS achieved the second place. 2022 GRIPS won the RoboCup German Open in Aachen and came in second at the RoboCup in Bangkok.

The following section of this paper describe the hardware modifications performed on the Robotino and any additional hardware used. Section 3 explores the algorithms and software architecture implemented on the configured hardware platform. The next section briefly discusses the overall mission strategy. In Section 5 the implemented development process are presented. The next section bridges this content to current research at the institute. Finally, this paper concludes with Section 7.

2 Hardware

GRIPS uses a multi-layer system architecture (see Fig. 1). An external PC acts as a so-called teamserver which is used to coordinate the autonomous robots by assigning tasks to them. The robotic base we are using is Festo's Robotino [2] (see Fig. 2b). The robot team consists of the three robots allowed per team according to the rulebook. All three robots are identical in terms of hardware and are equipped with a customized construction to achieve the correct height for the mounted gripper in order to be able to grab the products from the conveyors and shelves. Furthermore, we equipped the Robotinos with an additional external computer to improve computational power. For localization and navigation a Sick TIM551 laser scanner is used. Furthermore, we mounted a front facing camera on all robots to detect the machines' AR-tags.

A 3-axis gripper constructed with Festo parts is used for grabbing and delivering the products. There are 3 degrees of freedom:

- z-Axis: Vertical movement of the gripper
- r-Axis: Linear axis for moving the gripper forward and backward
- φ-Axis: Rotational axis around the robot center

We use two Pepperl+Fuchs short-range laser distance sensors for detecting the conveyor belt and the products. One is mounted front-facing for detecting the products, and one is mounted bottom-facing for detecting the conveyor belts. A Beckhoff PLC with various I/O extension cards reads the laser distance sensor values and controls the gripper axis.

A network router and a switch connect all these network devices and provide reliable WiFi capabilities for the robots. The used equipment is listed in Table 1.

3 Software

Following the idea of a three-layer architecture [3], the software is structured (as described in [4]) as follows: scheduler/planner, mid-level control and low-level. We introduced a strict communication scheme where only adjacent layers communicate with each other to achieve an increasing abstraction of the real world from layer to layer. Lower layers provide functionality to the higher layers (see Fig. 2a).

The *high-level* is responsible for coordinating the robots to achieve a common goal. It is also in charge of distributing tasks to the robots and to ensure that two robots do not use the same resource at the same time. Thus, the high-level needs to have global

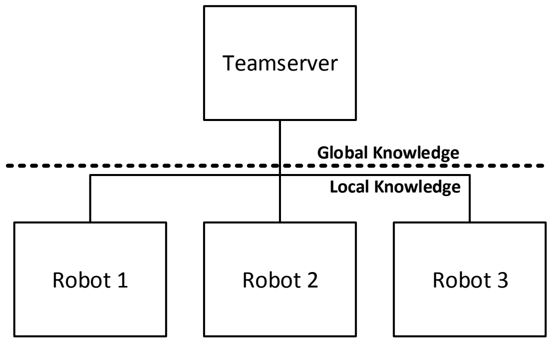


Fig. 1: System architecture used in our approach. The teamserver holds global knowledge of the game state while each robot only possesses local knowledge.

Table 1: Hardware components used in our approach.

Component	Quantity	Description
Robotino 3	3	Basis of all three robots.
Intel NUC	3	Additional computation device on all three robots.
Sick TIM551	3	Laserscanner for robot navigation.
Linksys Router	4	Network connections for all three robots and teamserver.
Axis Camera	3	Camera used for AR tag detection.
3-axis Festo Gripper	3	Gripper for retrieving and delivering products.
Beckhoff PLC CX9020	3	PLC for controlling the gripper and connecting the laser distance sensors.
Pepperl+Fuchs Distance Sensor	6	Laser distance sensor used for conveyor and product detection

knowledge of the game state. As can be seen in Fig. 1, the high-level is running on a dedicated PC.

The *mid-level* controls the individual robots, i.e. the mid-level plans the actions required to finish the tasks assigned to the robot by the high-level. Additionally, the mid-level is responsible for rectifying faults in the task execution as well as possible for the robot to resolve the problem locally by the respective robot.

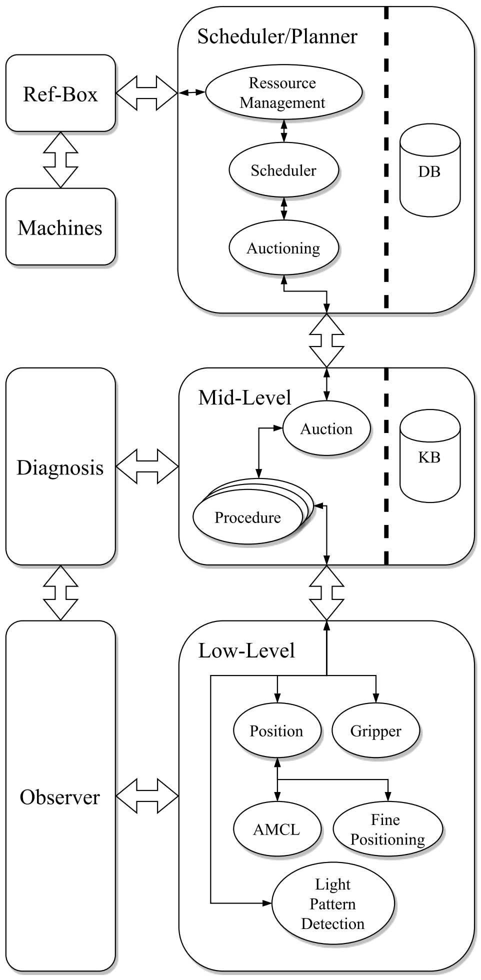
The *low-level* is responsible for executing primitive actions. These actions range from detecting the orientation of the machine to moving between way-points or opening the gripper. This level is the most reactive one and is the only level which performs near real-time activities.

Each level will be described in more detail in the remaining subsections of this section.

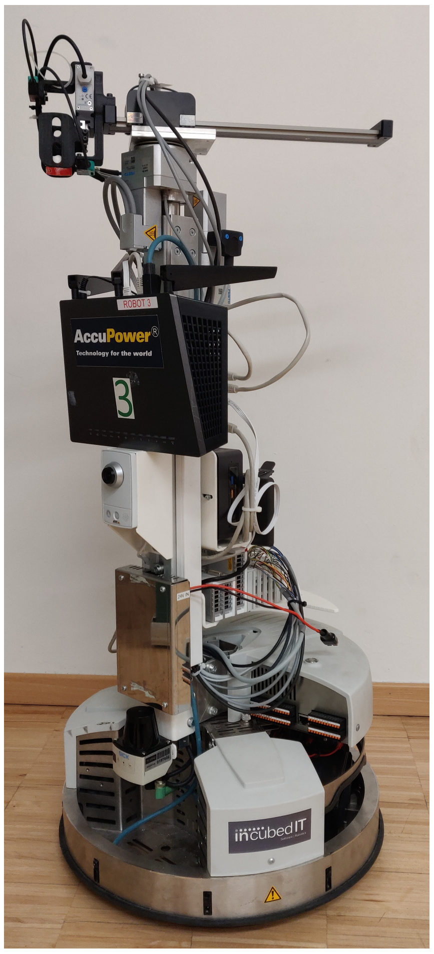
3.1 High-Level Planning and Execution

For the *high-level* task management, we developed a Planning and Dispatching Framework implementing a centralized control strategy for multi-agent system in a dynamic domain. It is based on three main components: (1) a Goal Reasoner, (2) a Planner, and (3) a Dispatching and Monitoring system. In Figure 3, the interaction between the components is shown . The Dispatching and Monitoring component plays the role of the main controller that invokes the Goal Reasoner, and consequently the planner, and executes the obtained plan. The plan execution is constantly monitored for issues that may require a regeneration of the goals or the plan, e.g. failed actions or deadline violations.

Modeling and Planning The planner is responsible for finding a plan to achieve the goals selected by the Goal Reasoner while minimizing its makespawn. The Goal Reasoner commits the selected goals to the planner. The planning process is performed using the temporal planner Optic [5] that provides a considerable set of features from PDDL 2.1 [6], like handling of time, action concurrency, and temporal constraints. For an introduction to temporal planning, we refer to [7]. We favor temporal planning over other approaches, like HTNs, as it is able to find better plans in terms of makespawn. The latter is needed to be able to serve many orders. Moreover, temporal planning makes handling temporal deadlines and coordinating multiple agents easier. The downside is the high computational cost of the planning process. In order to address this issue for the RCLL domain, we simplified the domain model and introduced a corresponding goal selection. The domain is modeled in PDDL by representing the two possible interactions with a station as abstract action: *get* and *delivery*. The former is used to retrieve a workpiece from a station after it has been processed, while the latter is needed to deliver the workpiece to a station for processing. An example is the action *deliverProductToCS(r cs p color)*. This action models the delivery of a product to a cap station, in order to mount a cap. The parameters include the robot *r* performing the action, the cap station *cs*, the partial product *p*, and the color of the requested cap *color*. The result of the planning process is a temporal plan, which is represented by the schedule σ, formed by a set of triples $\langle a, t_a, d_a \rangle$, where a is an action, t_a is the time when the action a needs to be started, and d_a is the duration of the action. Listing 1.1 shows the plan to deliver a simple product in the RCLL domain.



(a) Overview of the software architecture.



(b) Adapted Robotino 3.

Fig. 2: Used software architecture and hardware setup.

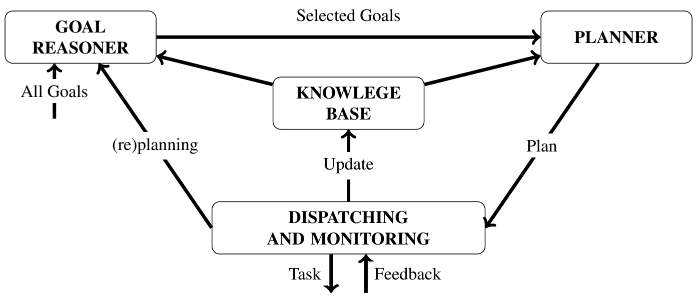


Fig. 3: Planning and Dispatching Architecture.

Listing 1.1: Temporal plan a C_0 Product. Parameters (except for the agent) have been omitted for better readability.

```
0.000:(getBaseFromBS r1)[51.000]
0.000:(bufferCapBaseFromCS r2)[89.000]
89.001:(getBaseFromCS r2)[52.000]
141.002:(deliverProducttoCS r1)[85.000]
226.003:(getProductFromCS r1)[52.000]
278.004:(deliverProductToDS r1)[73.000]
```

This representation poorly supports action dispatching, since it is not easy to determine how delays, occurred in the execution of an action, affect the other triples of the plan. To address this, we represent the plan as a temporal graph (Simple Temporal Network), which encodes the temporal dependencies and the partial order between the actions, without constraining the start of actions to specific time points. It can be computed from the temporal plan and the planning domain. A temporal graph G is a directed graph $G = (V, E)$. The set of vertices V represents time points, like starting or ending of an action, while each edge (or arc) $e \in E$ is a pair of vertices $(v_1, v_2) \in V \times V$. The function $\Lambda : E \to \mathbb{N} \times \{\mathbb{N}, \infty\}$ labels each edge with a pair of natural numbers $[lb, ub]$ or with $[lb, +\infty)$. In a label $[lb, ub]$ for an arc (v_1, v_2), lb and up express the lower and upper bound of time difference between the time points associated to v_1 and v_2. This allows to express temporal constraints between time points. The fact that (v_1, v_2) is a directed edge encodes a partial order relation from v_1 to v_2, stating that v_2 can not happen before v_1. In its less restrictive form, a label $[0, +\infty)$ encodes just a partial order relation from v_1 and v_2, without imposing any temporal constraint. Given an action a with estimated duration d_a, we denote with a_s and a_e the two vertices representing start and end of a. An edge labeled $[d_a, d_a]$ is created from a_s to a_e. Figure 4 shows the temporal graph equivalent to the plan in Listing 1.1.

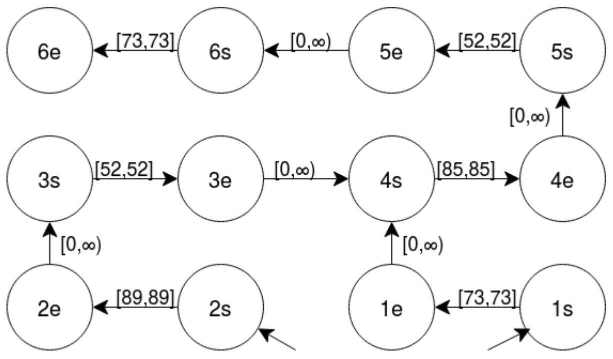


Fig. 4: Temporal graph for the plan shown in Listing 1.1.

An accurate estimation of actions duration is crucial, both for appropriate planning and monitoring plan execution. Since the temporal planner tries to optimize the makespawn of the plan, the plan is considered optimal in reality only if the estimated duration of an action is close to the real execution time. In our domain every action models a specific interaction between a robot and a station. For performance reasons, we do not model the actions to move a robot between two locations. As a consequence, in every action we need to consider the time needed for the robot to move from its position to the involved station. Each action is thus characterized by two cost (or time) components: (1) the movement costs and the (2) interaction costs. The interaction costs have been estimated for each action empirically. We determined the average duration during several invocations of the action. In order to estimate the duration of the movement included in the actions, we precompute a matrix for each setup (variable location of stations), containing all the estimated costs for traveling between each pair of locations. Since the same path finding algorithm is used as in the robot's navigation skill to find the best path between locations, the estimated times are close to the real ones. To make the estimation even more precise, we are also considering orientation costs at the start and at the end of each movement action. By adopting this procedure we achieve an accurate estimation of actions duration that is incorporated into the PDDL domain.

Goal Reasoning and Multi-Thread Planning Since we are interested in dynamic domains, the planning process needs to be fast, in order to be able to react to changes and opportunities in the environment in time. For this reason, the Goal Reasoner selects the most rewarding set of goals the planner may be able to obtain a plan for with a given time budget (e.g., 1 min.). In the RCLL domain a single goal is an individual order (product configuration and delivery time). In general, planning for all received orders is not possible within the given time. Thus, we follow the idea of partial satisfactory planning [8]. The Goal Reasoner solves a relaxed version of the RCLL domain, where it is assumed that each goal can be achieved by a single agent individually, without considering cooperation or management of resources. In this simple setting the set of processing steps needed for goal g are known in advance. Using duration estimation of actions described in the previous section, we can formulate the selection as a simple task allocation problem with an overall deadline. We denote with $M(g)$ the makespan of g (sum of process steps) and with $R(g)$ the reward for achieving it. G denotes the set of all possible goals, $SG \subseteq G$ the set of the selected goals. DL is the total remaining time for achieving said goals. Solving the relaxed problem means to find a task allocation τ, formed by a set of allocations $\langle r_i, g_j \rangle$ (robot r_i works on goal $g_j \in SG$) such that $\forall r_i \sum_{\langle r_i, g_j \rangle \in \tau} M(g_j) < DL$ and $max \sum_{g_j \in SG} R(g_j)$. We employ the ASP solver CLINGO [9] to compute this optimization task.

A drawback of this goal selection heuristic is that the solution of the relaxed problem can be too optimistic. In this case, the planner is not able to find a plan achieving all the selected goals while respecting the given deadlines. To mitigate this we exploit parallel planning over multiple sets of goals. The sets are the selected goal set as well as sets obtained by transforming it by either dropping goals (reducing the number of goals) or replacing goals with less complex ones. For each goal set, a planning thread is ran for 1 minute. The rational behind this approach is that in general it is more likely to find a feasible plan for simplified goal sets with a bounded time budget. Among the returned feasible plans, we select and dispatch the one with the highest reward. Thus, we trade a higher reward for a better response time.

Dispatching and Monitoring The task of this module is to dispatch the actions of the obtained plan in time, to monitor its proper execution (in terms of state and time), and to initiate replanning when necessary. The following three events may trigger replanning: (1) failed execution of a task, (2) successful execution of the actual plan, (3) impossibility to achieve a goal in time.

In ROSPlan a temporal graph is dispatched starting from the root node and traversing its outgoing edges, respecting their lower and upper bound $[lb, ub]$. A new node can only be dispatched after all its incoming edges have been traversed successfully. For instance, looking at Figure 4, we can see that action 4 (*deliverProducttoCS*) must be preceded by action 3 and 2. However, action 1 can be executed in parallel to actions 2 and 3. In this work, we use a slightly different approach to check the temporal constraint on the edges, in order to allow an earlier detection of deadline violations. As can be seen in Figure 4, there are two types of edges: (1) action duration edges $[d_a, d_a]$ between start and end of the action a, and (2) partial order edges $[0, +\infty)$ between the end of an action and the start of the following one. Nodes corresponding to start actions are dispatched by sending the corresponding action to the robot for execution, while end nodes are dispatched when a feedback for the successful execution of the corresponding action is received. The partial order encoded through the edges is ensured by the dispatching. Unfortunately, it can not be expected that the execution in the real world perfectly respects the temporal constraint $[d_a, d_a]$. The approach presented in [10] propagates the detected delay by fixing the execution times of already executed actions and updating (by constraint propagation) the remaining deadlines. In our approach, violations of such edges are tolerated, since our early deadline detection strategy immediately recognizes if such delays will cause a deadline violation in the future. Thus, we adopted the opposite approach, propagating back the deadlines from the goal nodes to the rest of the graph, labeling each node with a relative deadline $rdl(x)$. This value represents the latest time point we can dispatch that node without violating the deadline x. Starting from the *goal* vertices, we calculate the relative deadlines rdl of each vertex by traversing the edges in the opposite direction, subtracting the lower bound of the edge at each step. If there are more paths from a normal vertex to a *goal* vertex, we keep the more restrictive relative deadline. Every time a node is dispatched, we check if all its relative deadlines are respected, otherwise we trigger replanning. The advantage is that constraint propagation is performed only once, instead of every time a delay is detected. In Figure 5 a partial result of this approach applied to the graph from Figure 4 is shown.

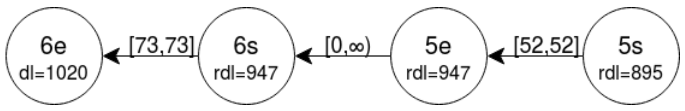


Fig. 5: The deadline propagation process on a part of the graph in Figure 4.

Teamserver Visualization Since every team uses three robots and the seven machines provided in the production hall during the RCLL competitions, it is hard to observe the tasks of robots and the current states of machines simultaneously. However, knowing what a robot is currently doing or planning to do is essential in many situations to foresee errors or possible problems. Issues, such as bad localization or wrong task execution, could disrupt the manufacturing process. If we are able to foresee such issues, maintenance can be executed in an earlier stage and thus avoid the occurence of errors beforehand. As a solution, we developed a Angular website to visualize information on the robots and machines. For debugging it is also possible to manually send all the commands to a Robot. The information about each robot contains its state, current task, next task, current product and many more. The machines information contains the current state, whether it is currently being used by a robot, etc. As our team server uses the Spring framework, which is based on a model-view-control architecture and already supports web developement, the creation of a website skeleton linked to our team server was not a difficult task. The Angular UI uses a post http endpoint to collect the data. We also periodically save the visualization data during a game, so that it is possible to step through a game afterwards.

3.2 Mid-Level Control

The mid-level control is based on BehaviourTrees. We use the BehaviourTreeC++ library that is already available for ros noetic [11]. BehaviourTree are used to model complex behaviours, but with them it is simpler than with a Finite State Machine. The midlevel control consits of two nodes, first the gateway node which is used to communicate with the high level control. Right now there are two implementations, the *prs_gateway_node* and the *fake_gateway_node*. The first is used if the tasks which are received are send from the high level control. The second node is used via the command line, it is used for debugging. Note that via the *fake_gateway_node* is is possible to execute more trivial tasks, like aligning to a machine.

An example BehaviorTree that is used to navigate to a pre-defined waypoint can be seen in figure 6. This example should illustrate some functionalities and the clearly visible control flow. All nodes of the BehaviorTree are executed from left to right. Some nodes, so-called control-flow nodes, have defined behaviors, like the nodes *Sequence*, *Fallback*, *ReactiveSequence* and *RetryUntilSuccessful*. The other nodes are the actual actions which are performed. Thus, the basic procedure of the navigation is clearly visible from naming of the nodes. After the waypoint is read, it is checked if the robots is already at the specified waypoint. If not, a goal with the waypoint is sent to the move base. Afterwards, twice a second it is checked if the robot is still moving. The robot may have stopped moving because of an error within the move-base or because of a defined stop-behavior. Finally, it is checked if the robot successfully reached the waypoint. If not, the whole navigation procedure fails and is retried until it returns success (amount of retries is specified in control-flow node *RetryUntilSuccessful*).

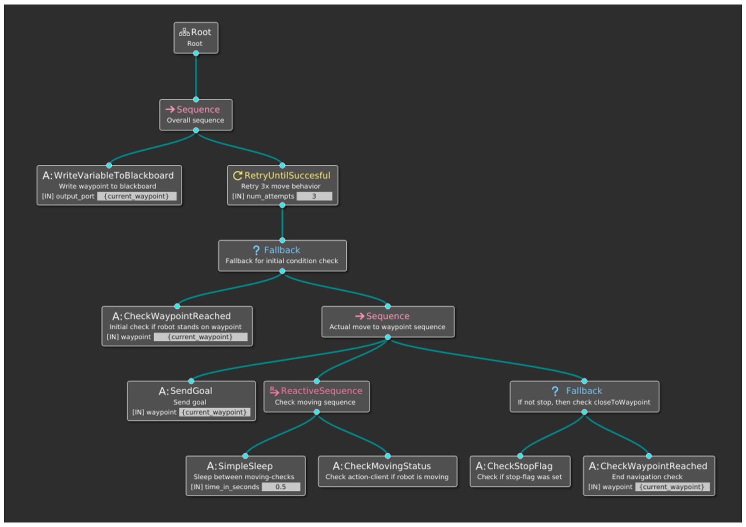

Root
Root
Sequence
Overall sequence
A: WriteVariableToBlackboard
Write waypoint to blackboard
[IN] output_port {current_waypoint}
RetryUntilSuccesful
Retry 3x move behavior
[IN] num_attempts 3
Fallback
Fallback for initial condition check
A: CheckWaypointReached
Initial check if robot stands on waypoint
[IN] waypoint {current_waypoint}
Sequence
Actual move to waypoint sequence
A: SendGoal
Send goal
[IN] waypoint {current_waypoint}
ReactiveSequence
Check moving sequence
Fallback
If not stop, then check closeToWaypoint
A: SimpleSleep
Sleep between moving-checks
[IN] time_in_seconds 0.5
A: CheckMovingStatus
Check action-client if robot is moving
A: CheckStopFlag
Check if stop-flag was set
A: CheckWaypointReached
End navigation check
[IN] waypoint {current_waypoint}

Fig. 6: Sample behaviour tree for moving the robot.

3.3 Low-Level

The lowest layer is mainly based on the open robot operating system (ROS [12]). Here, basic atomic actions are advertised to the layer above as a ROS action server. Different services are advertised here, e.g. opening/closing the gripper, locally navigating to a machine, aligning in front of a machine and so on. Performing these actions, all the error detection and a possible error recovery is made by the ROS service. In the following part of this subsection we will briefly discuss the most important parts of the low-level functions.

Way-Point Navigation In order to retrieve or deliver objects, the robot moves between defined way-points. These way-points are stored in the mid-level as abstract identifiers e.g. a defined zone of the playing field is stored as `M_Z22`. In order to move to a given way-point, the low-level uses a table to look up the real world coordinates for the abstract identifier and afterwards plan to move to this way-point. To move to the given coordinates, we use the move_base package of ROS [13]. This package comprises a global planner that uses the map to find a path between the current robot position and the desired final position. In order to avoid the different production machines, these machines are added to the navigation map. Furthermore, after finding a global plan a local planner is used to move along the path by considering and avoiding detected objects on the planned path.

Conveyor Alignment During the production phase the robot needs to deliver and retrieve products from the conveyor multiple times. This is done through a way-point that is in close proximity to the desired conveyor. Thus, a robot is able to move near the intended machine. After that, the QR-Code and the laser scanner is used to approach the machine and to align the robot close to the conveyor belt.

Gripper control The gripper control software is implemented directly on the PLC using Structure Text (ST) as programming language. It exposes an interface to the gripper ROS-node for accepting commands and sending sensor measurements. A command for the gripper can be fine-grained like opening or closing the gripper for grabbing a product, but the interface also allows for more complex commands. A more complex command sequence is for example grabbing a product from the shelf and directly delivering it to the input of the Cap Station.

The recognition of the product and conveyor belt positions is implemented on the PLC. By moving the r-axis and φ-axis along a linear path and simultaneously storing the values of the short-range laser sensors we get a point array of gripper positions and distance values. For scanning the product position on the conveyor belt at the machine output a front facing sensor is used. The gripper moves along a parallel path in front of the machine. For scanning the conveyor belt position at the machine input we use a bottom-facing sensor. The gripper moves along a linear path above the conveyor belt, to scan the characteristic shape of the conveyor. Using the sensor value point array and taking into account the expected dimensions of a product or conveyor belt the exact location of the product or conveyor can be calculated.

3.4 Communication

The communication between the scheduler/planner and the mid-level control is done in a similar way as the communication with the referee box where serialized messages generated with the open protocol buffer data format (protobuf) developed by Google are used. Also, the internal communication in the mid-layer is implemented using such serialized streams since the standard communication interface of Open PRS has some limitations that are mitigated by our implementation. The mid-level control communicates with the low-level using so-called ROS action servers, i.e. the low-level offers actions that can be triggered by the mid-layer. Implementing ROS action servers allows fetching the current status of the action during execution as well as a return status after the execution is finished or failed. The communication between ROS and the gripper PLC is implemented with the OPC UA protocol. The PLC acts as OPC UA server, and on the ROS side we use the Open62541 library for implementing an OPC UA client. Via the OPC UA protocol the sensor values measured by the PLC are exposed to low-level, and the low-level can send commands to the gripper.

4 Mission Strategy

The game can be split up into two main phases, exploration and production.
In the production phase, the Planning and Execution framework discussed in Section 3.1 is used to maximize the number of achieved product in the given time window.
In the exploration phase each robot will be in charge of exploring one column of our half of the field. To do so, the robot moves on the field and searches for unseen QR-tags.

Whenever one is found, the robot moves to the corresponding machine and measures the machine's orientation using the laser scanner. The collected information is then sent to the high-level Control. The scheduler/planner gathers all reports from the robots with additional information about the certainty of these observations. Having multiple observations with given probabilities, a safe report can be sent to the referee box to avoid negative points resulting from wrong reports. Exploring only one half of the field is sufficient due to the symmetry of the game field. Currently we develop a constraint based consistency check for the reported machines and zones.

5 Development

In order to build up a robust system during our development, we use a continuous integration [14] server that executes builds as well as unit and integration tests. These integration tests are executed with the help of the Gazebo simulation that is provided by BBUnits and the Carologistics team [15,16]. Through this continuous testing software faults and integration problems can be found more easily. To encourage other teams to follow the idea of continuous integration, we plan to release the software to perform these tests under an open source license after it has reached a certain level of maturity.

6 Influence Through Current Research

Within the focus of the current research at the Institute for Software Technology (IST) hosting the team is the usage of a model-driven approach to develop dependable autonomous robots. The idea is to use models as a central part of the developing process to automatically test during software development and to diagnose the system at runtime. These processes should be automated as much as possible transforming information such as requirements to models automatically or by reusing existing models [17,18]. This approach has already been shown to be applicable for an industrial use case. Thus, we hope that this approach also results in a robust system for the logistic league.

Furthermore, current research is done in order to design an agent architecture which allows a model driven approach to be easily integrated into a robotic system. This architecture should allow a robotic system to be more dependable and thus run for long periods of time. We expect that the result of this research is of special interest for the logistics league as this properties are of high interest for smart production use cases.

In the past we have investigated different high-level approaches to control the robot fleet together with the Carologistics team [19]. The high-level approach evaluated was based on YAGI (Yet Another Golog Interpreter), which is an implementation based on the ideas of the logic-based agent control language GOLOG. YAGI allows imperative as well as declarative parts to be used; thus, programming and planning can be balanced between ease of use and performance.

Then we shifted the focus on temporal planning systems, which are able to further optimize the returned plan in a multi-agent systems where deadlines are involved.

7 Conclusion

In order to get a global view of the current game state we use a central planning and scheduling instance that distributes the robot's tasks through an auctioning system. The robots use a BDI system in order to execute the tasks in a reliable and reactive manner. Additionally, the robots will be observed during run-time in order to detect faults and allow a fast recovery from failures. The robotic base we use for our robots is Festo's Robotino 3 which was modified such that the robot is capable of fulfilling all necessary tasks. Based on the research performed at the Institute for Software Technology at Graz University of Technology, we expect to provide the league with new ideas, to design more dependable systems.

Conformity with the Rules *We hereby declare that our presented hardware and software architecture satisfies all the requirements stated in the RoboCup Logistics Rulebook 2019 (published April 12, 2019).*

References

1. Haas, S., Keskic, D., Mühlbacher, C., Steinbauer, G., Ulz, T., Wallner, M.: Robocup logistics league tdp graz robust and intelligent production system grips. (2016)
2. Karras, U., Pensky, D., Rojas, O.: Mobile robotics in education and research of logistics. In: IROS 2011–Workshop on Metrics and Methodologies for Autonomous Robot Teams in Logistics. (2011)
3. Gat, E., et al.: On three-layer architectures. Artificial intelligence and mobile robots **195** (1998) 210
4. Wallner, M., Muehlbacher, C., Steinbauer, G., Haas, S., Ulz, T., Ludwiger, J.: A robust and flexible software architecture for autonomous robots in the context of industrie 4.0. In: Austrian Robotics Workshop. (2017) 67–73
5. Benton, J., Coles, A., Coles, A.: Temporal planning with preferences and time-dependent continuous costs. Proceedings of the International Conference on Automated Planning and Scheduling **22**(1) (May 2012) 2–10
6. Fox, M., Long, D.: Pddl2.1: An extension to pddl for expressing temporal planning domains. J. Artif. Intell. Res. (JAIR) **20** (12 2003) 61–124
7. Nau, D., Ghallab, M., Traverso, P.: Automated Planning: Theory and Practice. Morgan Kaufmann Publishers Inc., San Francisco, CA, USA (2004)
8. Van Den Briel, M., Sanchez, R., Do, M., Kambhampati, S.: Effective approaches for partial satisfaction (over-subscription) planning. In: Proceedings of the National Conference on Artificial Intelligence (AAAI). (2004) 562–569
9. Gebser, M., Kaminski, R., Kaufmann, B., Schaub, T.: Clingo= asp+ control. arXiv preprint arXiv:1405.3694 (2014)
10. Castillo, L., Fdez-Olivares, J., González-Muñoz, A.: A temporal constraint network based temporal planner. (01 2002)
11. Faconti, D.: Models and tools to design robotic behaviors. (2019)
12. Quigley, M., Conley, K., Gerkey, B., Faust, J., Foote, T., Leibs, J., Wheeler, R., Ng, A.Y.: Ros: an open-source robot operating system. In: ICRA workshop on open source software. Volume 3. (2009) 5
13. Marder-Eppstein, E., Berger, E., Foote, T., Gerkey, B., Konolige, K.: The office marathon: Robust navigation in an indoor office environment. In: International Conference on Robotics and Automation. (2010)

14. Duvall, P.M., Matyas, S., Glover, A.: Continuous integration: improving software quality and reducing risk. Pearson Education (2007)
15. Zwilling, F., Niemueller, T., Lakemeyer, G.: Simulation for the robocup logistics league with real-world environment agency and multi-level abstraction. In: RoboCup 2014: Robot World Cup XVIII. Springer (2014) 220–232
16. Niemueller, T., Reuter, S., Ewert, D., Ferrein, A., Jeschke, S., Lakemeyer, G.: Decisive factors for the success of the carologistics robocup team in the robocup logistics league 2014. In: RoboCup 2014: Robot World Cup XVIII. Springer (2014) 155–167
17. Simón, J.S., Mühlbacher, C., Steinbauer, G.: Automatic model generation to diagnose autonomous systems. In: Proceedings of the 26th International Workshop on Principles of Diagnosis (DX-2015) co-located with 9th IFAC Symposium on Fault Detection, Supervision and Safety for Technical Processes (Safeprocess 2015), Paris, France, August 31 - September 3, 2015. (2015) 153–158
18. Mühlbacher, C., Gspandl, S., Reip, M., Steinbauer, G.: Improving dependability of industrial transport robots using model-based techniques. In: Robotics and Automation (ICRA), 2016 IEEE International Conference on, IEEE (2016) 3133–3140
19. Ferrein, A., Maier, C., Mühlbacher, C., Niemueller, T., Steinbauer, G., Vassos, S.: Controlling logistics robots with the action-based language yagi. In: IROS Workshop on Taks Planning for Intelligent Robots in Service and Manufacturing. (2015)