Planning and Execution Competition for Logistics Robots in Simulation

Rules and Regulations – ICAPS 2017

The Technical Committee 2017

Tim Niemueller **Erez Karpas**
Tiago Vaquero **Eric Timmons**

Revision: v2 (Friday, June 9th 2017)

Contents

1 Introduction

With robots gaining ever more capabilities, both in terms of perception and manipulation, and with the desire to solve tasks of increasing complexity and higher relevance, the design and composition of robot behavior becomes more complex and tedious. The goal is to automate this process as much as possible, which would improve longevity, extensibility, and robustness of integrated robot systems.

Planning systems would be a natural component for developing complex robot behavior. However, such systems are still the exception rather than the norm in robotics applications. We see several reasons, both, from the robotics and the planning side. On the one hand, robotic systems are often used to develop, demonstrate, and evaluate specific capabilities like perception or manipulation. On the other hand, the planning community often uses robotics as a motivation, rather than as a full evaluation testbed. System integration and actual execution of plans and typical time constraints encountered in robotic domains are not considered. Robotic systems often involve a considerable number of components and planning systems further add to the complexity. An effort is required to ease the integration effort and make the communities more accessible towards each other. This lead to the creation of this competition [10]. It builds on the industry-inspired scenario of the RoboCup Logistics League (RCLL). We will adopt the standard platform of the Carologistics team [8, 7], winner of the last three RCLL competitions. Furthermore, to make participation as easy as possible, the competition will be run in simulation (at least, in its first iteration). Thus, the challenge for competitors is one of planning and execution.

In the first part of this document we describe the rules of the Planning and Execution Competition for Logistics Robots in Simulation. They are based on the 2016 scenario of the RCLL [11]. In the second part, we detail the technical interfaces to integrate own systems with the integration and the provided infrastructure. Finally, in the third part, we describe the deployment of the simulation in a compute cluster and how to produce the artifacts necessary to participate.

Part I
Challenge

2 The Task

The RoboCup Logistics League (RCLL) deals with a group of three robots performing in-house logistics of a smart factory scenario [5] maintaining and optimizing the material flow among processing machines. The full description of the challenge can be found in the RCLL rulebook [11]. This competition is only about the production phase of the RCLL competition. We provide a brief review of the most relevant rules here, especially defining the scenario.

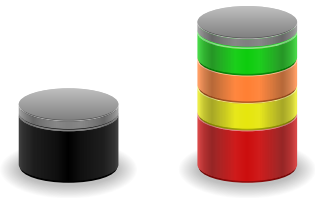


Figure 1: Two example configurations for products, one for a low complexity product without rings (left), and one of the highest complexity with three colored rings (colors and order of colors matters).

RCLL is played between two teams (cyan and magenta). Each team receives orders, and must complete those orders and deliver them on time to score points. Each order is composed of two to five pieces, which must be dispensed using different machines. More specifically, each order consists of one base, zero to two rings, and a cap. Each order specifies the color of each of these elements, as well as the number and order of the rings. Additionally, each order has a

time window for delivery, as well as a specific delivery gate to be delivered to. A (transient) stack of a base with or without rings and cap is called a workpiece, it is the entity to operate the machines with. A workpiece ready for delivery is also called a product.

There are two principal active components on the fields: machines and robots. Machines have a fixed position on the field and provide processing steps for workpieces. They are controlled by the referee box, a reasoning system providing environment agency processing input information from machines and triggering the proper responses. Robots are mobile entities under the control of the teams.

2.1 Playing Field

The playing field is 12 m × 6 m,[1] see Figure 2. The machines of both teams are distributed semi-randomly around the field. There are two primary sides on the field split by the middle Y-axis (center axis perpendicular to the long side of the field). The positively valued side is assigned to cyan, the negatively valued side to magenta. Some machines of each team will be on the opposite team's primary side. The machines are distributed symmetrically with respect to the Y-axis. The robots start in the insertion areas along the X-axis (going through one side of the field) on the respective far side.

[1]The field is divided into 24 virtual rectangular zones of 2m × 1.5m. However, these zones are only relevant during exploration and thus should be ignored. Note that we might switch to the more fine grained 106 square zones of 1 m $times$ 1 m, as it allows for better automated field construction in simulation. But again, since the zones itself to not carry any meaning for the production, this change does not matter for the ICAPS competition.

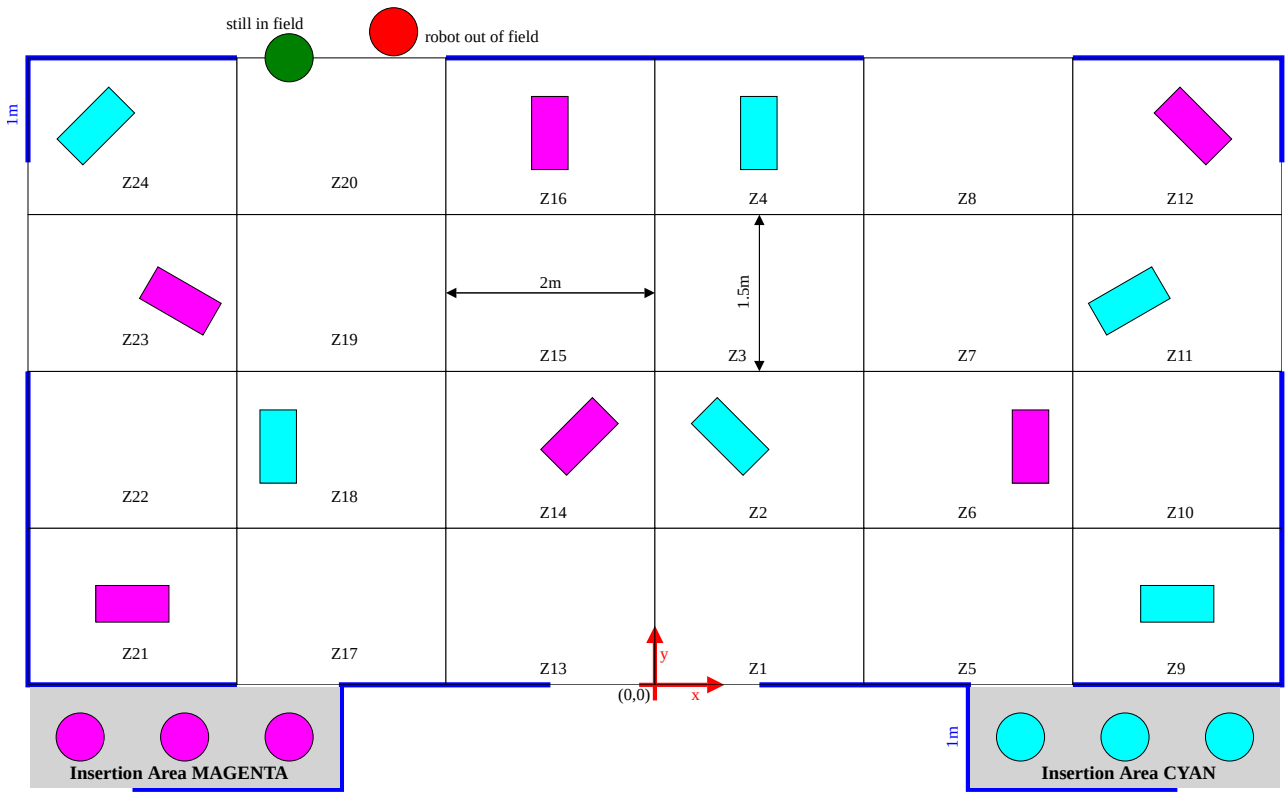


Figure 2: Competition area: squares indicate possible machine placements, circles denote robots. Cyan and magenta are team assignment colors. Thick blue lines are wall elements. The thin black inner lines indicate zones. The green robot is at an acceptable position, while the red one would have to be penalized for leaving the field. Grey areas are insertion areas where robots start [11].

2.2 Machines – MPS Stations

Machines are represented by MPS stations.[2] Each team owns 6 machines which are of 4 different types. Each machine has an input side and an output side. Typically, a machine is fed on the input side, and the processed workpieces are picked up on the output side. A machine requires a message with instructions to prepare it before each operation. The machine can be sent at any time before the machine is used. However, there is no way to reset preparation. Hence a short time before using the machine is typically advisable. Using a machine without preparation is an error and will lead to a temporary broken state of the machine (the machine is inoperable for 30 seconds and will loose any previous state). The machine types and their instructions are:

Base Station (BS) (1 per team)
: The BS dispenses bases of a specified color on the specified side (input or output). The prepare message specifies color and side.

Delivery Station (DS) (1 per team)
: The DS is used to deliver products. The completed order should be loaded into the input side by a robot, and is moved to the delivery gate specified during preparation.

Ring Station (RS) (2 per team)
: The RS is used to mount a rings on an order in progress. Each ring station can dispense rings in two colors (out of the four possible ring colors), so that both machines cover all possible colors. Some colors require one or two additional bases that must be loaded onto a separate slide before mounting a ring. An RS must be prepared with the ring color to mount. Feeding material onto the slide does not require preparation and can be done any time, even after preparing a specific ring color.

: At most three additional bases may be fed into a machine at a time, i.e., the difference of the total number of bases filled into a machine and the number of bases used by ring productions must be smaller than or equal to three.

Cap Station (CS) (2 per team)
: The CS is used to buffer (store) a cap, and then mount the cap on a workpiece. First, a base work piece with a cap (retrieved from a shelf on the CS) must be loaded in the input side by a robot. The cap is buffered and the empty base must be retrieved and be discarded or fed into an RS as additional material. Then, a workpiece belonging to an order can be fed to the machine to mount the buffered cap. Before retrieval or mounting, the station must be prepared stating the next operation to perform.

: In the simulation, the shelf on the team's CS1 is pre-filled with gray caps, on the CS2 with black caps. They are available from three positions on the shelf. Once the shelf is empty, it is refilled automatically.

2.2.1 Machine State

The machines have a state machine. The state is communicated periodically by the referee box and thus observable at any time (but not necessarily instantly on state change) from any place. In the following, we denote the possible states (in the message, they appear as an upper-case string exactly as in the list below.

IDLE The machine is ready for use by receiving a prepare message.

[2] The Modular Production System (MPS) is a mechatronics education platform by Festo Didactic SE. Actual machines are used in the RCLL. For more information see `http://www.robocup-logistics.org/links/festo-mps`.

BROKEN The machine has received an invalid prepare message or was used erroneously, e.g., feeding a workpiece without preparation.

PREPARED The machine has been successfully prepared. Note that you won't see this on a BS as that station immediately starts processing.

PROCESSING The machine has received an input workpiece and starts processing or preparation triggered processing.

PROCESSED Processing has finished and the workpiece is currently being moved to the output point.

READY-AT-OUTPUT A workpiece is ready to be picked up at the output.

WAIT-IDLE A workpiece has been retrieved from the output. The machine will shortly go into the IDLE state.

DOWN The machine is down for maintenance for a limited time and may not be used.

During a game, CS and RS may undergo a short period of planned maintenance in which it cannot be used (denoted by the DOWN state). Per team, there will be 2 such triggered events during a match. The affected machine will remain out of order for 30 to 60 seconds.

2.3 Robots

Each team controls three robots. The robots are based on the Festo Robotino 3[3] platform. The simulation models are based on information provided by the vendor. The specific model is based on the modified robot by the Carologistics team shown on the right. The robots feature a holonomic three-wheel drive. In the simulation, they are equipped with a laser range finder at about 30 cm height. They feature a simple two-finger gripper to handle workpieces. While full camera images could be simulated, here we generate a higher level of abstraction generating some information directly from simulation models, for example markers on the machines and their position relative to the robot. The laser data is used for self-localization based on adaptive Monte Carlo localization, collision avoidance during navigation, and machine detection (detecting a machine's straight line surface).

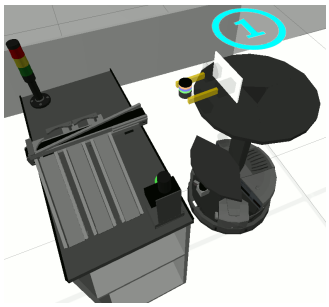


Figure 3: Robot approach DS in simulation.

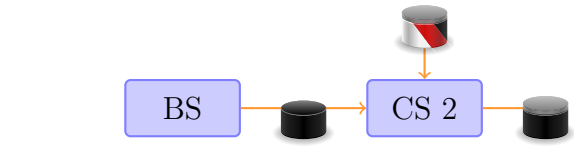
2.4 Orders

Orders denote the products which may be delivered and thus score points. Orders have a specific complexity denoting the number of rings to mount from C_0 (no ring) to C_3 (three rings). A product therefore consists of a colored base (out of three colors), zero to three rings (out of four colors), and a cap (out of two colors). The order of the rings matters. Each order will list each ring color at most once. And order is due for delivery in a specified time window given in seconds of game time. For instance, an order open from 300 to 360 is due to be delivered after five minutes of production, but no later than 6 minutes after production start. Only some score may be achieved if delivered outside the time window. Each order has a lead time, which denotes when the order is announced to the teams. Generally, the more complex a product the longer the lead time.

The order schedule is generated randomly by the referee box according to some distribution and announced at run-time according to delivery time windows and lead times. There is always a standing C_0 order, meaning that it can be delivered (once) anytime during the production phase.

Production chains for each of the respective complexity levels are shown in Figure 4.

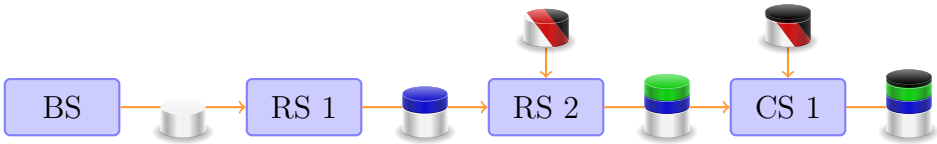
[3] `http://www.robocup-logistics.org/links/festo-robotino-3`



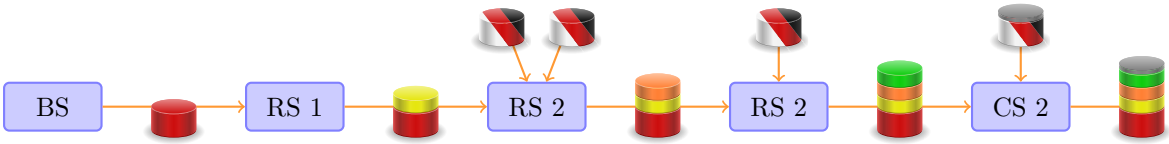
(a) Production of complexity C_0 only mounting a cap on a BE.

(b) Production of complexity C_1 with a single intermediate ring.

(c) Legend



(d) Complexity C_2 with two rings. RS 2 requires an additional base (of any color) for the green ring.



(e) Production of the highest complexity C_3 with a three intermediate rings. Note that RS 2 requires two additional BE (of any color) for the orange ring and another one for the green ring.

Figure 4: Production Chains of four example products [11]. Blue boxes represent machines. RS 1 mounts yellow and blue, and RS 2 orange and green rings. Mounting an orange ring requires one additional BE (of any color). Similarly two additional BE are required for a yellow ring. Cap stations require a BE (of any color) with a cap of the matching required color. *Note that this is a particular example. The actual production chains and requirements for additional BEs are determined randomly by the referee box for each game.*

2.5 Scoring

The main metric to evaluate a game and to determine the winner is game score. Points are awarded for the successful delivery of ordered products according to the previously mentioned order schedule. Points are awarded on delivery. Therefore, score for intermediate steps is only awarded when the related product is eventually delivered. If the product is not delivered during the game no score is awarded. The risk assessment and order decision must be done by the robots or planning system autonomously. The scoring is shown in Table 1.

3 Challenges and Tracks

3.1 Planning and Execution Challenges

The planning task itself is a complex one, involving:

- Temporal planning with soft goals and deadlines
- Uncontrollable action durations
- Environment uncertainty (machine failures, robots from other team)

Sub-task	Production Phase	Points
Additional base	Feed an additional base into a ring station	$+2$
Finish CC_0 step	Finish the work order for a color requiring no additional base	$+5$
Finish CC_1 step	Finish the work order for a color requiring one additional base	$+10$
Finish CC_2 step	Finish the work order for a color requiring two additional bases	$+20$
Finish C_1 pre-cap	Mount the last ring of a C_1 product	$+10$
Finish C_2 pre-cap	Mount the last ring of a C_2 product	$+30$
Finish C_3 pre-cap	Mount the last ring of a C_3 product	$+80$
Mount cap	Mount the cap on a product	$+10$
Delivery	Deliver one of the final product variants to the designated loading zone at the time specified in the order	$+20$
Delayed Delivery	An order delivered within 10 seconds after an order is awarded a reduced score. For delivery time slot end T_e and actual delivery time T_d in seconds the reduced score is given by $\lfloor 15 - \lfloor T_d - T_e \rfloor * 1.5 + 5 \rfloor$	up to $+20$
Late Delivery	An order delivered after 10 seconds	+5
Wrong delivery	Deliver one of the final product variants to the designated loading zone out of the requested time range or after all products requested in the period have already been delivered	$+1$
False delivery	Deliver an intermediate product	0
Obstruction Penalty	Deliver a workpiece to a machine, which belongs to the opposing team	-20

Table 1: Scoring Scheme for the Production

The task is made even more challenging by the fact that planning and execution must be integrated, in order to handle:

- Real-time planning — planning is done online, while robots are supposed to be operating
- Dynamic goals — goals arrive online

3.2 Competition Tracks

The competition will consist of three tracks, described below. Participants are free to use any planner, executive, planning formalism/language, or domain model they see fit in all tracks. However, we will provide a limited reference implementation based on PDDL, with ROSPlan [1] or MPEX.

Track 1 – Task Planning with Execution
In this track, participants will use the already implemented behaviors and path planner. The participants can exchange any or all of the planner, the executive, the domain model, or the state estimator (essentially anything related to the gray box in Figure 6). There will be two reference implementations the participants can use and/or extend.

Track 2 – Task and Path Planning with Execution
In this track, in addition to the components mention in Track 1, participants may also modify or exchange the path planning and navigation components.

Track 3 – Task and Path Planning with Execution and Basic Behaviors

In this track, in addition to the components mentioned in Track 2, participants may also modify or exchange the basic behaviors using or replacing the Lua-based Behavior Engine.

> **!** In 2017, Tracks 1 and 2 will compete in the same pool (i.e., against each other), and Track 3 will not be run. If there are two many teams in Tracks 1 and 2 to perform round-robin games in a feasible time, we may still split into two pools, again.

Part II
Simulation Environment and Interfaces

The competition will be run using the Gazebo simulation and the environment description, models, and plugins developed by the Carologistics team [12, 8]. In the following, we will outline the technical parameters regarding the run-time environment of the simulation and the available technical interaction interfaces. We close with information about running the provided reference testbed from a USB stick and how to reproduce the setup.

4 System Architecture and Overview

First, we describe the overall system architecture including the simulation. In Figure 5, the center box shows a detailed model of the simulation. The red parts are 3D models provided through the `gazebo-rcll` package, as are the Gazebo plugins (blue boxes) that generate the sensor data of the robot and execute low-level actuation commands. There is also a plugin providing environment agency through communicating with the referee box (refbox, provided in the `llsf-refbox` package). The refbox reads all information from the MPS stations and issues commands for the appropriate responses. It is also responsible for generating the order schedule. All simulated data and command interfaces are exposed through a Gazebo API using the Gazebo middleware (green box). This is not directly accessible to competitors. It is also used to access information by the visualization. But more importantly, it is used by one Fawkes instance per robot for data access (red boxes, provided by the `fawkes-robotino` package). Within Fawkes, sensor processing, actuation, and behavior plugins are located. For each robot, there is a separate ROS master and robot-specific ROS API (violet box embedded in Fawkes). It is mainly used for per-robot state visualization, e.g., for navigation. There is also a unified ROS API (large violet box at top, described in greater detail below) that allows to access the relevant information of all robots through a single ROS master. All access is performed through Fawkes (as it provides a better native multi-robot middleware) and the refbox communication layer. There are two principal ways to integrate with Fawkes, internally as a plugin or as an external application. Examples for internal integration are the CLIPS-based agent system or OpenPRS (denoted as gray PS boxes inside Fawkes). They operate in a distributed fashion with local decision making for each individual robot. In case of a centralized planning system, internal integration is possible as

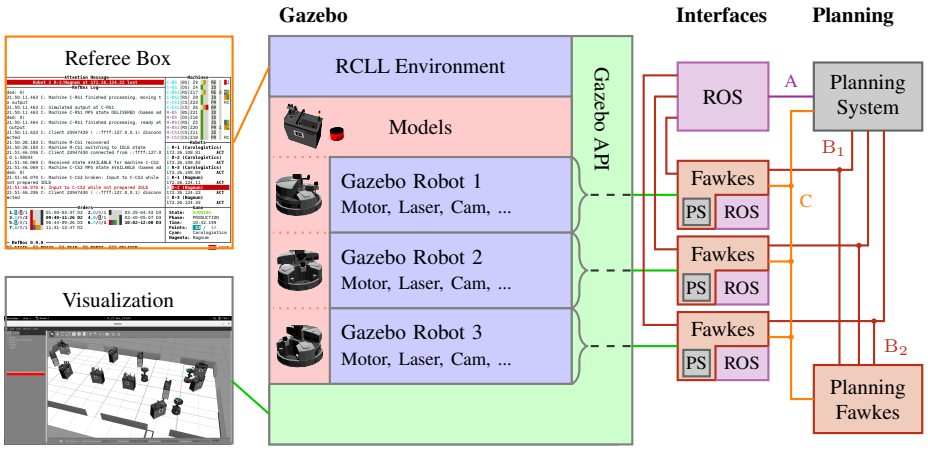


Figure 5: Simulation Architecture

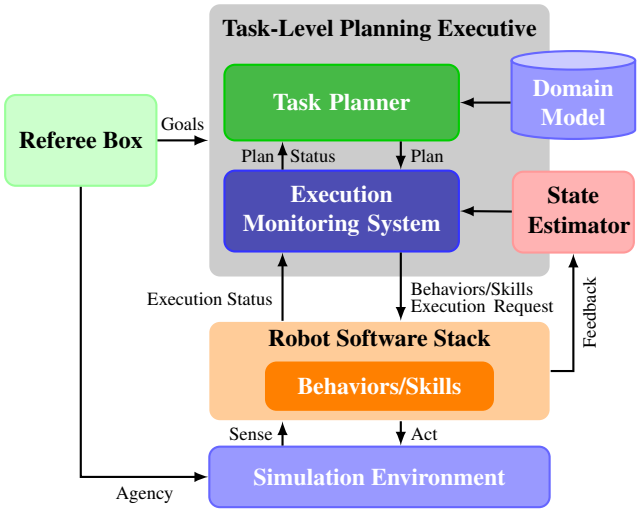


Figure 6: Planning System Architecture

another Fawkes instance (red box bottom right). There are three principal ways of communication. First, for external ROS-based planning systems, such as ROSPlan or MPEX, a full centralized ROS API is provided (A). However, such systems may also choose to use a similar Fawkes API that interacts with the robots directly (B_1, this is very much the same API the ROS wrapper uses). That API may also be used by Fawkes-based planning systems (B_2). Then there is a peer-to-peer communication channel based on `protobuf_comm` (C, which is the same protocol used for communication with the refbox). It allows to send and receive messages in a broadcast fashion to all robots.

The planning architecture is shown in Figure 6. It is essentially an abstracted or zoom view of the upper right corner of Figure 5. As we explained before, multiple ways of integration are possible. For the sake of simplicity, and since we think it will be the most requested mode, we focus now on integrating an external planning system through a ROS-only interface over a single ROS master.

Gazebo is encapsulated in the Simulation Environment. The Robot Software Stack are the Fawkes and ROS modules running in the Interfaces slice of Figure 5. The Task-Planning Executive (Planning System in Figure 5) is assumed to be a centralized, global planner. The Planning System uses communication mode A (full ROS wrapping) for Execution Status and Behaviors/Skills Execution Requests, and Feedback to the state estimator. The communication with the refbox is opaque, as it is encapsulated through ROS nodes as well. The teams can now freely modify, extend, or replace anything (partially) covered by the gray box in Figure 6. The reference implementation sections will explain the actual implementation of the individual components.

5 Components

There are several software components required which implement specific parts of our architecture. While we describe all basic components, in the remainder of the document we focus on the ROS integration.

5.1 Simulation

Gazebo[4] provides the basic simulation environment and several base plugins. The simulation supports Gazebo 5, 6, or 7. No earlier version is supported. Version 8 has not been thoroughly tested and there may be issues. Models and plugins are provided in the `gazebo-rcll`[5] package. These are domain- and platform-specific. It is being developed as a joint effort.[6]

5.2 Fawkes

Fawkes[7] is a Robot Software Framework similar to ROS. It differs in its communication middleware using a hybrid blackboard/messaging approach. Data is provided in the blackboard as data structures by a single writer which any number of threads can read. Messages can be sent to the writer for command instructions. Furthermore, the middleware can more easily incorporate multiple hosts in a multi-robot system than ROS (which requires techniques called "multi-master". Within Fawkes, plugins provide functionality. In our system, about 50 plugins provide integration with the simulation, basic functionality such as self-localization and navigation, and basic behaviors. One instance is running per robot. To look into the internal data, point your web browser to `http://localhost:8081` for the web interface of the first robot (8082 and 8083 for the second and third robot, respectively). We use the *Fawkes RCLL 2016* release[8] which is based on Fawkes 1.0.1. Earlier versions are not compatible.

5.3 ROS

The Robot Operating System (ROS)[9] is a popular integration environment in robotics. It uses a publisher/subscriber middleware with a central broker (the master) providing service discovery for endpoints (nodes) to connect to each other. In our system, there will be one master per robot (that can be used for visualization, set the master URI to port 11321, 11322, or 11323 for the three robots respectively). There is a central ROS master at the default port. It exposes all relevant interface for the overall fleet of robots in appropriate namespaces per robot. We will detail these in Section 7. We use *ROS Kinetic* for integration.

5.4 Lua-based Behavior Engine (BE)

The BE[10] [3] is used to model, implement, and run the basic behavior used by the executive. Skills are modeled as hybrid state machines. They are encapsulated as a single function called from the outside, for example like `ppgoto{place="C-BS-I"}`. More details and a list of available skills is provided in Section 8.

[4] `http://gazebosim.org/`
[5] `https://github.com/robocup-logistics/gazebo-rcll`
[6] `https://www.fawkesrobotics.org/projects/rcll-sim/`
[7] `https://www.fawkesrobotics.org`
[8] `https://www.fawkesrobotics.org/projects/rcll2016-release/`
[9] `http://www.ros.org/`
[10] `https://www.fawkesrobotics.org/projects/behavior-engine/`

5.5 RCLL Referee Box

The referee box (refbox)[11] provides environment agency and controls the overall game. We use version 1.1.0. It is currently being refactored in a generalized refbox for various competitive industrial robotic scenarios [9].

5.6 ROS Packages

A number of ROS nodes are required that provide the interface explained below. We briefly explain the packages involved.

- *fawkes_msgs*: general interface messages to provide Fawkes-based data and services to ROS.
  `http://wiki.ros.org/fawkes_msgs`
- *rcll_fawkes_sim*: Fawkes-based RCLL simulation integration nodes, e.g., to access basic behaviors or the navigational information.
  `http://wiki.ros.org/rcll_fawkes_sim`
- *rcll_fawkes_sim_msgs*: Interface messages for rcll_fawkes_sim.
  `http://wiki.ros.org/rcll_fawkes_sim_msgs`
- *rcll_refbox_peer*: RCLL referee box integration node.
  `http://wiki.ros.org/rcll_refbox_peer`
- *rcll_ros_msgs*: RCLL-specific ROS messages, used by rcll_refbox_peer.
  `http://wiki.ros.org/rcll_ros_msgs`
- *rcll_ros*: central RCLL for ROS package providing PDDL domains, configuration, and launch files to start the ROS integration.
  `http://wiki.ros.org/rcll_ros`

In case your are using the ROSPlan reference system, the following additional packages are necessary.

- *rosplan*: The ROSPlan base system.
  `https://kcl-planning.github.io/ROSPlan/`
- *occupancy_grid_utils*: ROSPlan requirement.
  `https://github.com/clearpathrobotics/occupancy_grid_utils`
- *mongodb_store*: ROSPlan requirement.
  `http://wiki.ros.org/mongodb_store`
- *rosplan_initial_situation*: Node to create object instances and assert initial predicates upon startup.
  `https://github.com/timn/rosplan_initial_situation`
- *rosplan_interface_behaviorengine*: Action dispatcher for ROSPlan via the Lua-based Behavior Engine and knowledge base updater for navgraph information.
  `https://github.com/timn/rosplan_interface_behaviorengine`
- *rosplan_interface_rcllrefbox*: Action dispatcher for instructions to the refbox (machine preparation) and knowledge base updater for machine and order info.
  `https://github.com/timn/rosplan_interface_rcllrefbox`

[11] `http://www.robocup-logistics.org/refbox`

6 System Environment

The simulation generally expects a recent Linux distribution such as Fedora 25 or Ubuntu 16.04. Fedora is strongly recommended as it is our main development environment. We provide a USB stick image you can use for development or at least testing and inspecting the integrated system. We briefly describe possible types of infrastructure we might use to run the competition. The final decision about the exact infrastructure to be used will be made based on competitor requests and available resources.

6.1 USB Stick Image

We provide an image that can be written to a USB flash drive to fully boot the machine off of the stick. To do so, you need a minimum of 32 GB on the flash drive. We strongly recommend a USB 3.0 (super speed) compatible drive and a computer with a compatible port that can use the full speed during boot. We use Sandisk Ultra Flair USB 3.0 32 GB. USB 2.0 is possible but vastly slower.

Download the latest image for the USB flash drive.[12] Then write the image to a USB flash drive. On Windows, we recommend using Etcher,[13] on Linux either the Gnome Disks tool, or on the command line using[14]:

```
pv rcll-simstick-2017-03-27.img.xz | xz -dc | sudo dd of=/dev/sdc
```

Afterwards boot a computer from the USB stick. You may use either UEFI or class BIOS boot procedures, as the image is prepared for both. The user name is robosim, the password is simcomp2017. Network access via SSH is disabled by default and may be enabled using

```
systemctl enable sshd && systemctl start sshd
```

6.2 Docker Container and Virtual Machine

We yet have to determine the mode of operation for performing the competition. The most probable solution is an automated deployment based on Kubernetes and Deployment. We will provide Docker base images for own development. To be announced by May 2017.

Note that it is also possible to run the USB stick as a virtual machine. To do so using Virtual Box,[15] download the image and unzip it. Then convert the image to a VDI file by running:

```
vboxmanage convertfromraw rcll-simstick-2017-03-27.img \
rcll-simstick-2017-03-27.vdi --format VDI
```

You can then create a virtual machine through the GUI, using the file you created (rcll-simstick-2017-03-27.vdi) as the hard drive. Make sure to give it plenty of RAM and video memory.

7 ROS Interface

As outlined in Section 4, there are several principle ways of interacting with the simulation. For the moment, we will describe the ROS-based interface for an external planning system (as in communication mode A, Figure 5) in more detail and defer the Fawkes interface to a later beta release.

Running the simulation provides a central ROS master (roscore) that runs on the default port. It is used to run the main launch file starting all services. For each robot, there is a separate namespace covering the relevant topics and services. They are named `robot1`, `robot2`, and `robot3`

[12] `https://files.fawkesrobotics.org/projects/rcll-simstick`
[13] `https://etcher.io/`
[14] Details at `https://trac.fawkesrobotics.org/wiki/RestoreUSBStickImage`
[15] `https://www.virtualbox.org/`

respectively. Note that when running with fewer robots there are also fewer namespaces. We will now describe the different aspects of the ROS-based API. Note that for the sake of brevity, we will only describe the API of the first robot in its namespace.

7.1 Referee Box Communication

Communication with the refbox is provided through the rcll_refbox_peer node. It communicates with the referee box and exposes the information via ROS topics and provides services to send messages. Further documentation is available at `http://wiki.ros.org/rcll_refbox_peer`. We omit topics and services relevant to the exploration phase only.

It is important to understand the basic form of communication with the refbox. It is generally done broadcasting datagrams (via UDP) to all systems, i.e., robots and the refbox. There are three channels: the public channel used by the refbox to send general game announcements, and per-team private encrypted channels. These are used by teams to send instructions and receive information from the refbox. The private channel may also be used for inter-robot communication. Note that the datagram nature of the transport does entail packet loss in typical real-robot deployments. This is modeled in the simulation and will cause a conservative loss of 10 % of the broadcast messages. This makes it a requirement to verify, e.g., if preparation messages have been processed by the refbox before continuing (and re-sending them until it did). Additionally, information is sent out periodically, typically in intervals of 1 or 2 seconds to ensure that eventually the information is received. The network medium should be used sparingly. The packet size for a single message is limited to 1 KB.

7.1.1 Topics

The following topics are relevant for receiving information from the refbox.

robot1/rcll/beacon (rcll_ros_msgs/BeaconSignal)
: Incoming beacon signals from other robots and refbox (see below).

robot1/rcll/game_state (rcll_ros_msgs/GameState)
: The game state contains the current phase of the game (e.g., SETUP or PRODUCTION) as well as the current game state (e.g., RUNNING or PAUSED). It also contains the current score of both teams as well as the game time. The game time starts anew in each phase and is a monotone real counter in seconds. For example, after the SETUP phase, the game time will reset to zero and be increased with each further game state message. Note that in the competition, only the PRODUCTION phase will occur.

robot1/rcll/machine_info (rcll_ros_msgs/MachineInfo)
: Messages on this topic contain information about the machines relevant to the team, most notably the machine state which can be used to determine, for example, whether preparation of a machine has succeeded. For RS, it also indicates which ring colors are available on the machine.

robot1/rcll/order_info (rcll_ros_msgs/OrderInfo)
: During production, the refbox will periodically send a list of all announced orders. An order is announced ahead of the delivery time window. Thus the listing of an order on this topic does not indicate that it is already or still due for immediate delivery. It might only open up in the future or have already passed. An order specifies the product complexity, and the color specification for bases, caps, and optionally rings (if required by the complexity level). It also states how many products of this kind are requested and when and where delivery is due.

robot1/rcll/ring_info (rcll_ros_msgs/RingInfo)
: The ring info specifies how many additional bases are required for the specific ring colors.

Constants for the different coloring of items is available in rcll_ros_msgs/ProductColor.

7.1.2 Services

Services are used to send messages and instructions to the refbox.

/robot1/rcll/send_beacon (rcll_ros_msgs/SendBeaconSignal)
Inquire sending a beacon signal to the refbox (see below).

/robot1/rcll/send_prepare_machine (rcll_ros_msgs/SendPrepareMachine)
Order the preparation of a machine. It requires a parametrization specific to the actual machine type (consult the RCLL rulebook). The result is an ok flag (true on successful preparation or message sending, false otherwise) and an optional message in case of an error.
The service provides two operating modes indicated through the boolean wait field in the request. If it is set to true, the service may block longer to increase the chance of successful communication. It will wait for up to 10 seconds for the machine to be instructed to go into the IDLE state (this avoid problems when immediately trying to prepare a machine after retrieving a workpiece). If this does not happen, the service fails. Otherwise, it will send the prepare instruction and then wait indefinitely for incoming machine info updates to verify that the machine has left the IDLE state. If the machine entered the BROKEN state, that is it received an invalid preparation message, the service reports failure. Otherwise, it returns success. Note that all machines but the BS will enter the PREPARED state. However, the BS immediately starts dispensing the requested product and the state quickly toggles through PROCESSING to READY-AT-OUTPUT.

7.1.3 Beacon Signal

The beacon signal is a periodic message sent by all robots and the refbox. It is used to indicate presence and to detect fatal network failures. It is expected that the highest-level controlling (software) component (for example, the executive) sends this signal periodically to signal liveness. In order to do so, the component has to periodically call the rcll/send_beacon service.

However, as this might be problematic for some planning systems, especially during early integration, we provide the rcll_beacon_sender node, which can perform this task in the meantime. A message will be published to the rcll/beacon topic when a beacon is received. It thus allows to detect other robots and their positions.

7.2 Navigation

Only a small amount of information is provided via ROS. This is provided by the rcll_fawkes_sim and fawkes_navgraph nodes. Note that the navgraph information is present only once in the global namespace, and not per robot.

The robot uses a dynamically generated navigation graph (navgraph) for global path planning and to keep information about specific points of interest, e.g., MPS stations.[16] In the graph, nodes represent points of interest or intermediate travel nodes. Edges mark ways that can be traveled among nodes. Nodes can have arbitrary properties as key/value pairs. For navigation, the robot first searches a global path based on the navgraph. That path is then given node-by-node to the local planner for execution. The local planner may deviate arbitrarily far from the original path and is prepared to

[16] More details about the navgraph and documentation about its features and specification is available at `https://trac.fawkesrobotics.org/wiki/NavGraph`

take shortcuts if getting close to a node further down along the path. All positions are with respect to the global (map) coordinate frame. There are tools to use statically defined graphs from a file, to dynamically generate a graph (used in this competition), and to interactively inspect and modify graphs. You can inspect the navgraph through rviz on the per-robot ROS master. Currently, search is performed using a straight-line distance heuristic and cost is provided in path length (meters).

7.2.1 Topics

robot1/rcll_sim/amcl_pose (geometry_msgs/PoseWithCovarianceStamped)
: This topic provides the position of the robot on the field. It is published at about 10 Hz for timely updates.

navgraph (fawkes_msgs/NavGraph)
: This latched topic provides a full description of the currently used navgraph including all properties of nodes and edges. Messages are only sent on navgraph updates and on subscription (latching publisher).

7.2.2 Services

/navgraph/search_path (fawkes_msgs/NavGraphSearchPath)
: This service can be used to calculate a path between two nodes or between a node and an arbitrary position in global (map) coordinates. If the destination node name is empty, the pose will be used for calculation. The result contains an ok flag to indicate success or failure. An optional error message is set on error. The path consists of a sequence of nodes assumed to be traveled in consecutive order. The cost is also provided.

/navgraph/get_pairwise_costs (fawkes_msgs/NavGraphGetPairwiseCosts)
: This service allows for calculating the pairwise cost among a set of nodes. The argument is a list of names of nodes to consider. The result contains an ok flag to indicate success or failure. An optional error message is set on error. On success, a list of pairwise connections among all nodes is returned with the respective cost for each connection (connections from a node to itself are not considered).

7.3 Behavior Execution

Basic behaviors are encoded as skills and provided through the Lua-based Behavior Engine. Here, we use it integrated into Fawkes and provide ROS access through the fawkes_skiller node of the rcll_fawkes_sim package.

Skills are invoked by passing a Lua-string denoting a function for execution through an ROS actionlib[17] action. Each skill implements a hybrid state machine with a designated start state and two terminal states, FINAL for success and FAILED for failure. The returned state is thus either INACTIVE, RUNNING, FINAL, or FAILED, respectively.

For a list of available skills see Section 8.

7.3.1 Topics

robot1/skiller/skill_status (fawkes_msgs/SkillStatus)
: This topic provides access to the skill status when not initiating the actionlib goal.

[17] `http://wiki.ros.org/actionlib`

7.3.2 Actions

/robot1/skiller (fawkes_msgs/ExecSkillAction)

This action allows to execute skills. The high-level skill state machine is directly mapped onto the actionlib state machine. On error, the goal is aborted with an error message. Successful completion will mark the action goal as succeeded.
The only argument is the skill string to be executed. While a goal is running, no further feedback is sent (you may use the skill_status topic mentioned above if necessary). On failure, an optional message may explain the cause of the error.

8 Basic Behaviors

We provide a set of basic skills to act in the environment. How to call such skills has been described in Section 7.3. In this section, we will describe the available behaviors. For each action, we give a general description, the skill string and its parameters, and an example PDDL operator (for how these could be mapped to a skill string please cf. Section 9). For the sake of simplicity, we only specify non-durative actions.

8.1 Skills

ppgoto – Move to a place

This issues a movement command through the global navgraph-based path planning.

Lua

```
ppgoto{place="<place>"}
```

Arguments:

place	Name of the place to go to.

PDDL

```
(:action move
  :parameters (?r - robot ?from - location ?to - machine)
  :precondition (and (entered-field ?r) (at ?r ?from))
  :effect (and (not (at ?r ?from)) (at ?r ?to))
)
```

drive_into_field – enter the field on start

This issues a special movement command to enter the field initially. You must ensure to call this only on one robot at a time or with a suitable time offset.

Lua

```
drive_into_field{team="<team-color>", wait=<R>}
```

Arguments:

team	Color of the team we are playing as, must be either CYAN or MAGENTA.
wait	Real number in seconds before starting to move (for concurrent execution on multiple robots).

PDDL

```
(:action enter-field
  :parameters (?r - robot ?team-color - team-color)
  :precondition (robot-waiting ?r)
  :effect (and (entered-field ?r) (at ?r START INPUT) (not (robot-waiting ?r)))
)
```

ax12gripper – control the gripper

This allows to control the state of the gripper. While generally not necessary as this is done by other skills, it may be useful to drop a workpiece.

Lua

```
ax12gripper{command="<ACTION>"}
```

Arguments:

command	action to perform, either OPEN or CLOSE.

PDDL

```
(:action wp-discard
  :parameters (?r - robot ?cc - cap-carrier)
  :precondition (and (holding ?r ?cc))
  :effect (and (not (holding ?r ?cc)) (can-hold ?r))
)
```

get_product_from – retrieve a workpiece from an MPS

This skill aligns at a machine and retrieves a workpiece. It can be used for both, retrieval from the conveyor, or a shelf on a CS.

Lua

```
get_product_from{place="<PLACE>", shelf="<SPOT>"}
get_product_from{place="<PLACE>", side="<SIDE>"}
```

Arguments:

place	MPS position to retrieve from without side, e.g., C-BS.
shelf	shelf spot to retrieve from, one of LEFT, MIDDLE, RIGHT
side	MPS side to retrieve from, one of “input” or “output”

PDDL

```
(:action wp-get-shelf
  :parameters (?r - robot ?cc - cap-carrier ?m - mps ?spot - shelf-spot)
  :precondition (and (at ?r ?m INPUT) (wp-on-shelf ?cc ?m ?spot) (can-hold ?r))
  :effect (and (holding ?r ?cc) (not (can-hold ?r)) (not (wp-on-shelf ?cc ?m ?spot)))
)

(:action wp-get
  :parameters (?r - robot ?wp - workpiece ?m - mps ?side - mps-side)
  :precondition (and (at ?r ?m ?side) (can-hold ?r) (wp-at ?wp ?m ?side)
                     (mps-state ?m READY-AT-OUTPUT) (wp-usable ?wp))
  :effect (and (not (wp-at ?wp ?m ?side))  (holding ?r ?wp) (not (can-hold ?r))
               (not (mps-state ?m READY-AT-OUTPUT)) (mps-state ?m IDLE))
)
```

bring_product_to – put a workpiece into an MPS

This skill aligns at a machine and places a workpiece in the machine. It can be used for both, putting onto the conveyor, or into the slide on an RS.

Lua

```
bring_product_to{place="<PLACE>", side="<SIDE>"}
bring_product_to{place="<PLACE>", side="input", slide=true}
```

Arguments:

place	MPS position to retrieve from without side, e.g., C-BS.
side	MPS side to retrieve from, one of “input” or “output”
slide	true when putting onto a slide

PDDL

```
(:action wp-put
  :parameters (?r - robot ?wp - workpiece ?m - mps)
  :precondition (and (at ?r ?m INPUT) (mps-state ?m PREPARED)
                     (wp-usable ?wp) (holding ?r ?wp))
  :effect (and (wp-at ?wp ?m INPUT) (not (holding ?r ?wp))  (can-hold ?r)
               (not (mps-state ?m PREPARED))  (mps-state ?m PROCESSING))
)

(:action wp-put-slide-cc
  :parameters (?r - robot ?wp - cap-carrier ?m - mps ?rs-before ?rs-after - ring-num)
  :precondition (and (mps-type ?m RS) (at ?r ?m INPUT) (holding ?r ?wp)
                     (rs-filled-with ?m ?rs-before) (rs-inc ?rs-before ?rs-after))
  :effect (and (not (holding ?r ?wp))  (can-hold ?r)
               (not (rs-filled-with ?m ?rs-before)) (rs-filled-with ?m ?rs-after))
)
```

9 ROSPlan Reference Implementation

A reference implementation is provided based on ROSPlan[18] [1]. We provide basic PDDL domains, tool support, and launch files to run the system with the simulation. The USB Stick comes readily integrated to immediately run the ROSPlan-based game. The required ROS packages have been described in Section 5.6. We are going to explain the ROSPlan-specific ones in more detail here.

9.1 ROSPlan Base System

The ROSPlan base system consists of the knowledge base which uses MongoDB as a general data store (inspired by the generic robot database [6]). An action dispatcher is responsible for triggering action execution. The planning system combines all elements, calls the actual planner, and interprets the plan to trigger execution. In its default configuration it uses the POPF planner.[19] Our PDDL domain has been designed with its expressive limitations in mind.

9.2 Knowledge-Base Updating/State Estimation

Several nodes called knowledge base updater (KBU) are responsible to feeding information into ROSPlan. They use the ROSPlan services to assert and retract information to and from the knowledge base. The nodes are written in such a way that they try to make the minimum number of changes required whenever possible. Thus they avoid frequent inconsistent partial states on the knowledge base (there are no transactions to prevent this). First, the kb_updater_rcll_machine_info reads the machine_info topic of the first robot. It creates object instances for machines, type and state predicates, ring color configuration per machine, and additional base requirements for ring colors. The node kb_updater_rcll_order_info publishes information about announced orders and functions denoting the delivery time window. The kb_updater_navgraph node allows to assert function values for the pairwise traveling cost among nodes. It supports setting a cost factor. This allows to estimate costs for example as time. So for an assumed average speed of 0.2 m/s a factor of 5 would be configured.

9.3 Action Dispatching

There are two general action dispatchers, one to issue preparation commands to the refbox, and the other to trigger the execution of skills.

Skill execution is performed by the rosplan_interface_behaviorengine node. It runs once per robot with appropriate namespace binding. It generally assumes that PDDL operators have a specific argument to identify which robot is supposed to execute an action (the ROSPlan action dispatch interface cannot make that distinction itself, like the `?r` parameter of the operators in Section 8. ROSPlan passes dispatched actions as an operator name and key-value pairs denoting the assignment of parameters to values. The interface nodes provides a configurable translation mechanism to convert actions to skill strings (cf. rcll_ros/config/rcll_action_mapping.yaml). This is done through a set of configuration parameters, which are defined with the configuration key indicating the PDDL operator name, and a conversion string. Note that the mapping depends on the PDDL domain description and its respective actions (durative and non-durative), and the available skills.

A mapping is a tuple of two elements:

[18] `https://kcl-planning.github.io/ROSPlan/`

[19] `https://nms.kcl.ac.uk/planning/software/popf.html`

- parameter key or path element (left of the colon), this is the PDDL operator name
- parameter value, this is a skill string template

The mapping value can use the following elements as a pattern modification for the skill string. Note that parameters are always converted to lower-case by ROSPlan (at least in the default combination with POPF).

Variable substitution has the general pattern `?(varname)M`, where varname is the name of the operator parameter and M is a modifier. The modifier must be one of:

- s or S: convert value to string, that is add quotation marks around the value. s passes the string as is, S converts to uppercase.
- y or Y: pass value as symbol, i.e., the string value as is without quotation marks. Y additionally converts to upper case.
- i: converts the value to an integer.
- f: converts the value to a float value

Additionally, the arguments may be modified with a chain of regular expressions. Then, the expression looks like this: `?(var|/pattern/replace/|...)` There can be an arbitrary number of regular expressions chained by the pipe (|) symbol. The "/pattern/replace/" can be a regular expression according to the C++ ECMAScript syntax. Note that the expressions may contain neither a slash (/) nor a pipe (|), not even if quoted with a backslash. This is because a rather simple pattern is used to extract the regex from the argument string. The replacement string may contain reference to matched groups.[20] In particular, the following might be useful:

- $$: an actual dollar sign
- $&: the full matched substring
- $n: the n'th capture (may also be $nn for $10 \leq \text{nn} \leq 99$) Note that regular expression matching is performed case-insensitive, that is because PDDL itself is also case-insensitive.

Here are some examples for the translation.

PDDL Operator
```
(move ?r - robot ?from ?to - location)
```
Config Parameter
```
move: ppgoto{place=?(to)S}
```
Translation
```
(move R-1 START C-BS-I) → ppgoto{place="C-BS-I"}
```

PDDL Operator
```
(enter-field ?r - robot ?team-color - team-color)
```
Config Param
```
enter-field: |
   drive_into_field{team=?(team-color)S,
         wait=?(r|/R-1/0.0/|/R-2/3.0/|/R-3/6.0/)f}
```
Translations
```
(enter-field R-1 CYAN)
→ drive_into_field{team="CYAN", wait=0.0}
(enter-field R-2 CYAN)
→ drive_into_field{team="CYAN", wait=3.0}
```

9.4 PDDL Domain

A POPF-compatible PDDL domain is provided in the rcll_ros package in `pddl/rcll_domain_production.pddl` along with an example problem. It models almost all relevant aspects of the game (delivery time windows are currently not supported), but has not been optimized in depth for efficient planning.

[20] For details see `http://ecma-international.org/ecma-262/5.1/#sec-15.5.4.11`

9.5 Running ROSPlan in the Simulation

On the USB stick, a basic scenario can be run by activating the icon labeled "ROSPlan" in the left application selector. To run from the command line, the use the following command:

```
cd $FAWKES_DIR
bin/gazsim.bash -n 1 -t -r \
  --ros-launch-main rcll_ros:rosplan_production_1robot.launch \
  --team-cyan Carologistics --start-game
```

This runs a game with a single robot (`-n` denotes the number of robot instances to start, from 1 to 3), using ground truth for navgraph generation (`-t`) and enabling ROS support (`-r`). The passed ROS launch file (format package:launchfile) will be run on the primary ROS master. It contains the appropriate settings for all nodes mentioned above. Setting the team is required to start the game in order for the system to receive information over the private team channel.

Detailed information about general setup and running the game is available at `https://trac.fawkesrobotics.org/wiki/RCLLSimulationSetup`.

10 CLIPS-based Reference Implementation

The most successful approach used in the RoboCup Logistics League is based on the CLIPS rule-based reasoning framework [4]. It is included in the Fawkes RCLL release the competition simulation software stack is based on.[21] While it does not contain the full domain encoding for 2016, it contains a working subset that was used in 2015. It is enabled by default on the USB stick.

The agent does not use long horizon planning, but it rather has a heuristic evaluation of current situation that chooses an action for a robot as soon as it becomes idle. In a further coordination step, robots ensure mutual exclusion on job assignments and MPS usage.

10.1 Running CLIPS Agent

On the USB stick, a basic scenario can be run by activating the icon labeled "CLIPS" in the left application selector. To run from the command line, the use the following command:

```
cd $FAWKES_DIR
bin/gazsim.bash -n 1 -t -r -a --team-cyan Carologistics --start-game
```

The `-a` flag triggers running the CLIPS agent.

11 MPEX Reference Implementation

A reference implementation based on MPEX will be provided in a future release.

[21] `https://www.fawkesrobotics.org/projects/rcll2016-release/`

Part III
Simulation Cluster Deployment

While in Part II described the specific simulation environment and available interfaces, in this part we describe how the overall system is integrated and deployed during the competition. That is, how the several system parts are combined into a working deployment constituting a game or tournament.

12 Simulation Cluster

The simulation is run using a Kubernetes cluster. Given declarative manifests of what to run, Kubernetes will distribute the given workload onto a set of compute nodes (bare metal or virtual machines running a kubelet, the Kubernetes client). This distribution respects resource limits as imposed by the nodes and as described in the manifests.

12.1 Kubernetes Terminology

We will now describe the most relevant terminology required to understand the simulation deployment. For more detailed information, we refer to [2].

Node
Worker machine that may be a virtual or a physical machine that runs the actual workloads.

Image
This can be understood as the disk image used to run a workload. It consists of a number of layers modeling dependencies, for example, a basic image would contain an operating system, on top an application layer could add relevant artifacts and access the lower layers' content.

> **!** In the simulation competition, teams will need to provide one or more images that contain their integrated part or parts of the system.
> For example, a team using the ROS API would provide a compatible image that contains the team's own ROS nodes along with a launch file or script to run them.

Container
A container is a specific instance at run-time executing based on a given image.

Pod
The pod describes a set of containers to run along with configuration information. It is the smallest schedulable entity in the Kubernetes cluster.

Manifest
Resource descriptions for items like pods are called manifests. They are written as YAML[22] or JSON[23] documents. In our setup, we use the YAML format.

[22] `http://www.yaml.org/`
[23] `http://www.json.org/`

12.2 RCLL Simulation Cluster at KBSG

This year's competition will be hosted by the Knowledge-Based Systems Group on a commodity bare-metal cluster. It is available during the qualification and competition period.[24]

The cluster consists of seven Intel i7 quad-core machines with 16 GB of RAM each. One machine is exclusively used for cluster management and critical add-ons, such as DNS and monitoring. The machines feature 250 GB of SSD and 500 GB spinning disk drive storage, each. The operating system is Fedora 25 running Kubernetes 1.5.3. For the competition, a total of 48 cores/threads and up to 96 GB of RAM are available. The cluster will be used to run a number of parallel concurrent games.

The cluster runs the following principal components. These cannot be modified or controlled by the teams but provide the basic infrastructure to run the competition. The list is informative only.

- Basic add-ons: KubeDNS, Dashboard (with Heapster and InfluxDB)
- Private Docker image registry
- Logging based on fluentd, Elasticsearch, and Kibana
- Ceph storage cluster (covering spinning drive disks to provide 2+ TB storage capacity)
- MongoDB stateful set with replication for management and game data
- Prometheus performance monitoring and Grafana visualization
- Ingress controller based on nginx

13 Architecture and Building Blocks

The system is composed of a number of components, which are integrated into separate pods. In the following, we describe the example of a game of the reference ROSPlan- and CLIPS-based reasoning systems against each other.

13.1 Pods when Playing CLIPS vs. ROSPlan

The ROSPlan system runs only with a single robot, but it runs the ROS integration pod (details below) and pod for ROSPlan and related nodes. The CLIPS-based agent runs with the full set of three agents and does require neither the ROS integration pod nor another custom pod. Rather, the agent runs as a plugin inside the robot pods. For each robot of a team exists a pod running the robot-specific components, most importantly its Fawkes instance performing simulation integration, navigation, self-localization etc. In our example, CLIPS plays as team cyan, ROSPlan as magenta.

Simulation Base Pod (template sim-refbox.yaml.j2)

This pod runs the basic simulation, the referee box integration, and controls the overall game flow. This pod is off limits for teams and may not be modified. Its containers are:

- *gazebo*: Gazebo simulation environment.
- *mongodb-refbox*: MongoDB instance used for refbox logging.
- *refbox*: referee box to oversee and instruct the game.
- *comm-proxy*: communication relay between robots/agents and the referee box.
- *run-game*: script running the game flow, i.e., it waits for robot Pods to be loaded and then starts the game. Collects game reports when the referee box reports game finished.

[24] The cluster is not generally publicly available. Registered participants need to contact the organizers to initiate test games. For 2018, we will provide instructions to setup the cluster in a public or private cloud.

gzweb(template gzweb.yaml.j2)
Simulation visualization over the web.

Robot Pod (template robot-pod.yaml.j2)
One of these pods runs for each robot in the game. In our example, we will have three pods robot-c1, robot-c2, robot-c3 (since CLIPS as cyan uses three robots) and robot-m1 (ROSPlan as magenta only plays with a single robot). Each pod consists of the following containers:

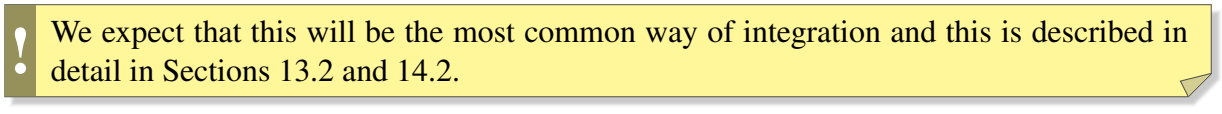
- *roscore*: A ROS master used for single-robot ROS integration (this contains only topics of this single robot and is not to be confused with the ROS integration container!).
- *fawkes*: This is the core of this pod and runs the simulation integration (accessing sensors and issuing commands), self-localization, the behavior engine etc. It can be customized to run additional plugins, which is used in our example to run the CLIPS-based agent.
- *localize-robot*: This is a pod that performs initial self-localization (telling the robot where it is on the field) once the fawkes container is up and ready.

ROS Integration Pod (template rcll-sim-ros.yaml.j2)
This pod runs the central ROS master and basic ROS integration nodes. It acts as the bridge to expose the robots' skill execution interfaces, navigation graph etc. The pod contains the following containers.

- *roscore*: The central ROS master which hosts namespaces for the individual robots (cf. Section 7).
- *roslaunch-integration*: This container runs the basic integration nodes through the `rcll_ros/rcll_sim_integration.launch` launch file. This contains referee box integration (optionally including periodic beacon sending), Behavior Engine access, and some basic information.
- *roslaunch-navgraph*: This has to be split off the main integration launch file as it needs to connect to the first robot's ROS master to access the navgraph. This is then written to a file shared with the roslaunch-integration container, which publishes the graph again to the central ROS master.[25]

Planning and Execution (example template rcll-sim-rosplan.yaml.j2)
Teams may specify an additional pod that has higher resource limits. This is intended as the "planner" pod. An example is the ROSPlan reference implementation (cf. Section 9). It contains the set of ROS nodes required to make the ROS interface accessible to and feed information into ROSPlan (included via `rosplan_production.launch`:

- `rosplan_planning_system.launch`: ROSPlan base system, including planner and knowledge base. If using ROSPlan using the available configuration arguments might be a good start.
- `rosplan_rcll_interface.launch`: Interface and knowledge base updater nodes that provide a basic interface (cf. Section 9.2 and Section 9.3).
- *rosplan_starter_rcll* node: This is a simple example how to trigger ROSPlan execution. The reference implementation must be modified to make useful decision when and what to (re-)plan.



[25] This is unfortunately necessary due to ROS' inability to cope with multiple masters from a single node.

13.2 Creating Custom Images

To participate, at a minimum teams need to create their own image and upload it to the KBSG cluster registry. These may then be run with a default or custom configuration (cf. Section 13.3.1).

To get acquainted with Docker, we recommend the Getting Started Guide.[26] In the following, we give instructions on the most crucial steps. However, creating your own image may need some further research beyond these instructions. In the following, we will assume to replace or extend the rcll-sim-rosplan image with some custom ROS packages, or possibly modified ROSPlan versions.

13.2.1 Docker Setup

To create an image, you must have installed and started docker. We recommend using a recent version of Docker (and thus underlying operating system). We have tested Fedora 25 and Ubuntu 16.04. The newest RCLL SimStick image comes with the necessary packages.

```
# On Fedora:
sudo dnf install docker

# On Ubuntu:
sudo apt-get install docker.io
```

You must either start docker before using it, or enable it permanently to be started on boot (note that this may be different on older Ubuntu versions).

```
sudo systemctl start docker
sudo systemctl enable docker
```

To verify that Docker is working use the following command:

```
sudo docker run --rm hello-world
```

This should print an informative message. Note that all "docker" commands must be executed as root, and thus using `sudo`.

13.2.2 Image Build

We start out from the rcll-sim-rosplan image,[27] which builds on the rcll-sim-ros image. Copy copy `Dockerfile.2016-f25-kinetic` and `rcll-sim-rosplan.rosinstall` from the rcll-sim-rosplan image to a new directory. Then start modifying the two files.

`rcll-sim-rosplan.rosinstall` Change, add, or remove packages in the rosinstall file as necessary. Make sure that your packages can *install cleanly*. That means that the `CMakeLists.txt` in your packages must have proper installation instructions.[28]

`Dockerfile.2016-f25-kinetic` This contains the build instructions. It is in a way similar to a Makefile. The most relevant statements are COPY, to copy files from the local directory into the image, and RUN, to execute commands during building. In the rcll-sim-rosplan file, the following steps are performed:

- `FROM timn/rcll-sim-ros:2016-f25-kinetic`

  Create the image on top of the rcll-sim-ros image.

[26] `https://docs.docker.com/get-started/`

[27] `https://github.com/timn/docker-robotics/tree/master/rcll-sim-rosplan`

[28] `http://wiki.ros.org/catkin/CMakeLists.txt#Optional_Step:_Specifying_Installable_Targets`

- `RUN dnf install -y flex python2-pymongo \&\& dnf clean all`

  Install additional required packages (and cleanup to reduce image size).

- `COPY rcll-sim-rosplan.rosinstall /opt/ros/`

  Copy rosinstall file so that it is accessible in the next step.

- ```
  RUN /bin/bash -c "source /etc/profile; \
    mkdir -p /opt/ros/catkin_ws_${ROS_DISTRO}_tf2_bullet/src; \
    cd /opt/ros/catkin_ws_${ROS_DISTRO}_tf2_bullet; \
    rosinstall_generator tf2_bullet --rosdistro $ROS_DISTRO \
      --deps --wet-only --tar \
      --exclude RPP > $ROS_DISTRO-tf2-bullet.rosinstall; \
    wstool init -j $(nproc) src ${ROS_DISTRO}-tf2-bullet.rosinstall; \
    rosdep install --from-paths src --ignore-src \
      --rosdistro $ROS_DISTRO -y; \
    catkin_make_isolated --install \
      --install-space /opt/ros/$ROS_DISTRO \
      -DCMAKE_BUILD_TYPE=$ROS_BUILD_TYPE; \
    rm -rf *_isolated; \
    "
  ```

  This creates a new rosinstall file (rosinstall_generator) for tf2_bullet, builds and installs (wstool, rosdep, catkin_make_isolated) it, and removes temporary files.

- ```
  RUN /bin/bash -c "source /etc/profile && \
    mkdir -p /opt/ros/catkin_ws_${ROS_DISTRO}_rcll_sim_rosplan/src; \
    cd /opt/ros/catkin_ws_${ROS_DISTRO}_rcll_sim_rosplan; \
    wstool init -j $(nproc) src ../rcll-sim-rosplan.rosinstall; \
    rosdep install --from-paths src --ignore-src \
      --rosdistro $ROS_DISTRO -y; \
    catkin_make_isolated --install \
      --install-space /opt/ros/$ROS_DISTRO \
      -DCMAKE_BUILD_TYPE=$ROS_BUILD_TYPE || exit $?; \
    rm -rf *_isolated; \
  ```

  Retrieve, build, and install the packages based on the previously installed rosinstall file.

- `RUN mkdir -p /opt/rosplan_kb`

  Create a directory required at run-time by the pod using this image.

In the simplest case, only the rosinstall needs to be updated with additional packages or URLs to forks of existing repos, but arbitrary build instructions may be added as necessary. Once the files have been created, the image can be built using

```
sudo docker build -f Dockerfile.2016-f25-kinetic -t image:latest .
```

Replace “image” with your username and image name, similar to timn/rcll-sim-rosplan. Images have a tag that often indicates the version. Using “latest” is similar to using a master branch in git. You may specify more specific tags, similar to “2016-f25-kinetic” in the original image.

13.2.3 Testing your Image

The first test to an image is to create a container (a running instance based on the image) and check if all files are in the expected place:

```
sudo docker run --rm -ti image:latest bash
```

This will create a container and give you an interactive shell in that container. You can now move within that container and execute commands. The ROS installation, where the files necessary for your container should have been installed, is in `/opt/ros/kinetic`.

13.2.4 Uploading your Image

To upload, you must first login to the private cluster registry using the credentials you have received:

```
sudo docker login registry.kbsg.rwth-aachen.de
```

Then tag the image for the registry and upload it. This will upload layers which have been modified since the last update. So after an initial large upload, the amount of data that needs to be uploaded should be reduced.

```
sudo docker tag image:latest registry.kbsg.rwth-aachen.de/user/image:latest
sudo docker push registry.kbsg.rwth-aachen.de/user/image:latest
```

13.3 Parameter and Manifest Templates

The relevant resources for a simulation of the deployment are described in a set of manifests (cf. Section 12.1 and Section 13.1). These pods need certain parameters, for example the names of the playing teams and robots. For flexibility, manifests are generated from a set of templates.[29] These contain variables and expressions which are expanded before instantiating a game. The template language used in Jinja[30] version 2.8 in the default configuration additionally including the "with" extension.

The parameters use during expansion of the manifest templates is taken from another template document, the parameter template `game.yaml.j2`.[31] This includes a number of other templates for the basic resources (simulation and robot pods, ROS integration), and team-specific extensions.

13.3.1 Team-Specific Parameter Template

The team-specific parameter template is a fragment of the game configuration template. You can find examples prefixed "`team-`" in the templates directory. In the following, we will specifically consider the ROSPlan reference implementation configuration. The fragment is imported, and not included. That means, that only variables and macros can be exported, but no actual output is produced.

Input Variables The following input variables are accessible to the template:

- `docker_registry`: registry from which images must be pulled. Prefix image names with this.

Output Variables The following output variables are considered from the fragment template:

- `num_robots`: the number of robots (and thus robot pods) to instantiate, may be 1, 2, or 3.
- `robot_pod_image`: the Docker image to use for instantiation of the robot pods. It must derive from the rcll-sim image (to provide `/opt/rcll-sim/run-component` and must include a compatible version of Fawkes Robotino.

[29] `https://github.com/timn/rcll-sim-cluster/tree/master/sim-ctrl/templates`
[30] `http://jinja.pocoo.org/`
[31] `https://github.com/timn/rcll-sim-cluster/blob/master/sim-ctrl/templates/game.yaml.j2`

- `crypto_key`: the encryption key for refbox communication. If not set, defaults to "randomkey".
- `run_ros_pod`: Set to True to instantiate the default ROS integration pod, False (or leave unset) otherwise.

Pod Macro The fragment may specify a pod macro of the form `pods(team_name, team_color)`. If present, it will be used to one or more instantiate additional pods.[32] The ROSPlan example creates one additional pod. This is based on the `rcll-sim-rosplan.yaml.j2` template. The `vars` map defines the values to substitutes for the variables with the respective names in the pod template. Here, it sets the image to use, the robots to expect, and URI of the ROS master to contact, team information, and a pod-specific variable to automatically start planning.

14 Customizing Base Pods

We envision two major ways modify the deployment for participation:

Modifying the robot base pod
: This will allow for extending the per-robot instances and run components as Fawkes plugins. The CLIPS-based agent is an example for this.

Adding a ROS Pod
: This allows or adding one or more additional pods utilizing the ROS interface to interact with the simulation. The ROSPlan reference implementation is an example for this.

In the following, we are going to describe these options in more detail.

14.1 Modifying the Robot Pods

Modifying the robot pods means to change the image and or configuration of the robot pods (see above). There are strict constraints when modifying the robot pods and images, in particular because they provide the basic interface to the simulation. Note that there are certain resource limits applied to robot pods which may not be increased, e.g., at most a single core may be used and only up to 1.5 GB of memory (which is shared with other minimum required plugins.

Modifying a robot pod essentially means creating an image derived from fawkes-robotino (or fedora-ros) that brings a specific (compatible) version of fawkes-robotino, most likely including custom plugins or configuration. The basic simulation integration plugins may not be modified (other than bug fixes from the simulation provider). Plugins added by competitors may *not* access the Gazebo integration interfaces or Gazebo's middleware. No direct interaction with Gazebo is allowed.

In the team meta template you can specify which plugins are to be loaded in addition to the mandatory base plugins.

14.2 Adding a ROS Pod

Using the ROS interface requires to run the appropriate pods. In the simplest case, this includes the standard ROS integration pod, and a custom pod that runs your specific ROS nodes. For planning systems, we expect the typical scenario to include one central pod for all robots. However, it is possible

[32] If you intend to use more than one contact the organizers to discuss this.

to run one such additional pod per robot. Then the given resource limit must be distributed among the per-robot pods accordingly. The basic ROS integration pod may be replaced (or reconfigured), for example when directly communicating with the referee box from your pod.

Participating teams using the ROS integration need to perform the following steps:

1. Create a Docker container image (use `timn/rcll-sim-ros` as the base image) and upload it to the KBSG cluster (cf. Section 13.2).
2. Create and configure a team-specific fragment template (start from `team-ROSPlan.j2`, cf. Section 13.3.1).
3. (Optional) Create a team-specific pod manifest template[33] (cf. Section 13.3).

15 Running a Game or Tournament

This section is intended for running a game on a cluster. In 2017, this will be handled through the organizers. For 2018, we will provide instructions on how to setup a custom simulation cluster on own machines or in a public cloud.

The general process to run a tournament (or, a number of test games, which we handle just the same) is based on a work queue, in which individual game pairings and parameters are stored. Then, a batch job is run processing all the games in the work queue, ticking off successful executions, and requeing failed games for up to three execution tries in total. The batch job supports configurable parallelism defining the number of games which may be running concurrently (which depends on the available and required computing and memory resources).

At a glance, running a tournament works like this:

1. Generate a game schedule
2. Run batch job to execute games
3. Collect the results

We describe these steps in more detail below.

15.1 Create a Single Game or Tournament Schedule

Creating a game means generating a job entry in the (generic) work queue. The work queue is a MongoDB database, accessed with atomic read and modify on job execution. Each job consists of a unique job name, parameters, and status information. The parameters hold the specifics for the game, such as tournament, team names, and game configuration.

All commands in Section 15.1 work through an instance of the rcll-sim-ctrl image. We provide the `devpod.yaml`[34] manifest to run an appropriate pod. The pod must be configured with the information to access the database. To log into the pod for command execution, use

```
kubectl exec -ti rcll-sim-ctrl-dev bash
```

You may also prefix the commands in this section for direct execution, e.g.,

[33] If creating a custom manifest template is necessary, contact the organizers to verify applicability.

[34] `https://github.com/timn/rcll-sim-cluster/blob/master/kubernetes/rcll-sim/devpod.yaml`

```
kubectl exec rcll-sim-ctrl-dev create-tournament ...
```

To create a job, the central game parametrization template is read (e.g, `game.yaml.j2`). The only parameters passed to the template are `tournament_name`, `team_name_cyan`, and `team_name_magenta`. All other parameters for the manifest instantiation (during execution) are generated by template processing.

15.1.1 Generating and Running a Single Game

Generating a single game is useful during testing. It allows to run a single job conveniently. A job is created like this:

```
create-sim-job --team-cyan Carologistics --team-magenta ROSPlan \
  --template /opt/rcll-sim-ctrl/templates/game.yaml.j2 TestGame
```

This creates a game for Carologistics playing against ROSPlan, using the default game template. The game will be associated to a tournament called TestGame (keep this distinct from potential tournaments you create to keep the results separate). For a single game, you may omit the magenta team to play with only a single team. This is not supported for tournament generation. The script will output the unique name of the job created. To see the generated parameter document, add the `--debug` flag. The `--dry-run` will cause the generation to run (and thus errors can be spotted or debug output analyzed) without actually storing the job in the work queue. The following will print a parameter document without storing the job:

```
create-sim-job --team-cyan Carologistics --team-magenta ROSPlan \
  --template /opt/rcll-sim-ctrl/templates/game.yaml.j2 \
  --debug --dry-run TestGame
```

To run the game (or any open job in the queue, really), execute:

```
run-sim-jobs --namespace testgame
```

This will create a new Kubernetes namespace and instantiate the pods as required per the game configuration. You will see the output of starting and monitoring the execution. A typical game will run for about 20 minutes. Note that the command will process all open jobs currently in the work queue. So running `create-sim-job` more than once will trigger multiple consecutive games.

15.1.2 Generating a Tournament Schedule

A tournament schedule takes a number of teams, and will create all permutations of the teams for a number of iterations. For a single iteration, teams will play each other twice (so each team has been assigned either color). The schedule is generated and stored to the work queue with the following command:

```
create-tournament --name TestTournament --iterations 3 \
  --template /opt/rcll-sim-ctrl/templates/game.yaml.j2 \
  Carologistics ROSPlan
```

This creates a schedule for the two given teams (add more teams as further positional arguments) playing 3 iterations. This results in 6 games to play (two pairings, three iterations).

Similar to single game generation, the `create-tournament` command accepts the `--debug` and `--dry-run` arguments.

15.2 Running a Tournament as a Batch Job

Kubernetes supports batch jobs that process a number of work items to completion. We use this to execute `run-sim-jobs` for the elements of the work queue. The manifest `runjob.yaml`[35] configures the job to run. There are two particularly relevant parameters. First, the `spec.parallelism` value determines the number of concurrently executed job pods. Each job pods processes a single work item (game) at a time until no more jobs are open. Since Kubernetes does not support closely coordinated pods by itself, our pod controller needs to handle this. However, this impedes the automatic resource management as Kubernetes does not know the overall resource requirements of a single job. Therefore, if two job pods compete for resources they may stall each other. Thusly, the number of concurrently processed work items must be chosen carefully.

Each job pod will create a distinct namespace for the created pods. This is why for each game, the refbox is reachable by the hostname "refbox", even though multiple are running. Each is in its respective namespace.

15.2.1 Watching Logs

To view the logs of the job pods, first determine the names of the pods:

```
kubectl get pods -l 'job-name=run-sim-jobs' -a
```

Then, the logs of each pod may be viewed like this (1a2bc is a random unique number shown in the command above):

```
kubectl logs -f run-sim-jobs-1a2bc
```

Other particularly interesting logs are from the run-game container in the rcll-sim pod. This is the per-game controller which instructs the refbox, transfers the game report, exports logging data, and prints the final result. Replace the namespace accordingly depending on the job.

```
kubectl logs -f --namespace=rcll-sim-0 -c run-game rcll-sim
```

15.3 Collecting Tournament Results

The results are stored as game reports in MongoDB. They are generated by the refbox of each game (in a local database) and transferred to the cluster database by the run-game container. The `print-results` script provides an aggregation pipeline to extract the results. It can be executed like the following:

```
print-results --tournament=TestTournament
```

This will print the aggregated results of the given tournament. You may add the flag `--teams` to list all participating teams, and `--individual-games` to see the scores of each game in the tournament.

[35] `https://github.com/timn/rcll-sim-cluster/blob/master/kubernetes/rcll-sim/runjob.yaml`

References

[1] Michael Cashmore, Maria Fox, Derek Long, Daniele Magazzeni, Bram Ridder, Arnau Carrera, Narcís Palomeras, Natàlia Hurtós, and Marc Carreras. "ROSPlan: Planning in the Robot Operating System". In: *25th International Conference on Automated Planning and Scheduling (ICAPS)*. 2015.

[2] *Kubernetes Concepts*. `https://kubernetes.io/docs/concepts/`.

[3] Tim Niemueller, Alexander Ferrein, and Gerhard Lakemeyer. "A Lua-based Behavior Engine for Controlling the Humanoid Robot Nao". In: *RoboCup Symposium 2009*. 2009.

[4] Tim Niemueller, Gerhard Lakemeyer, and Alexander Ferrein. "Incremental Task-level Reasoning in a Competitive Factory Automation Scenario". In: *AAAI Spring Symposium - Designing Intelligent Robots: Reintegrating AI*. 2013.

[5] Tim Niemueller, Gerhard Lakemeyer, and Alexander Ferrein. "The RoboCup Logistics League as a Benchmark for Planning in Robotics". In: *WS on Planning and Robotics (PlanRob) at Int. Conf. on Aut. Planning and Scheduling (ICAPS)*. 2015.

[6] Tim Niemueller, Gerhard Lakemeyer, and Siddhartha Srinivasa. "A Generic Robot Database and its Application in Fault Analysis and Performance Evaluation". In: *IEEE/RSJ International Conference on Intelligent Robots and Systems (IROS)*. 2012.

[7] Tim Niemueller, Sebastian Reuter, and Alexander Ferrein. "Fawkes for the RoboCup Logistics League". In: *RoboCup Symposium 2015 – Development Track*. 2015.

[8] Tim Niemueller, Tobias Neumann, Christoph Henke, Sebastian Schönitz, Sebastian Reuter, Alexander Ferrein, Sabina Jeschke, and Gerhard Lakemeyer. "Improvements for a Robust Production in the RoboCup Logistics League 2016". In: *RoboCup Symposium – Champion Teams Track*. 2016.

[9] Tim Niemueller, Sebastian Zug, Sven Schneider, and Ulrich Karras. "Knowledge-Based Instrumentation and Control for Competitive Industry-Inspired Robotic Domains". In: *KI - Künstliche Intelligenz* 30.289–299 (2016).

[10] Tim Niemueller, Erez Karpas, Tiago Vaquero, and Eric Timmons. "Planning Competition for Logistics Robots in Simulation". In: *WS on Planning and Robotics (PlanRob) at Int. Conf. on Automated Planning and Scheduling (ICAPS)*. London, UK, 2016.

[11] RCLL Technical Committee. *RoboCup Logistics League – Rules and Regulations 2016*. Tech. rep. RoboCup Logistics League, 2016.

[12] Frederik Zwilling, Tim Niemueller, and Gerhard Lakemeyer. "Simulation for the RoboCup Logistics League with Real-World Environment Agency and Multi-level Abstraction". In: *RoboCup Symposium*. 2014.