Team Description Paper 2020
Team AutonOHM

Marco Masannek,
Sally Zeitler and Maurice Hufnagel

University of Applied Sciences Nuremberg Georg-Simon-Ohm,
Kesslerplatz 12, 90489 Nuremberg, Germany
masannekma61828 @th-nuernberg.de
http://www.autonohm.de

Abstract. This team description paper presents the team AutonOHM which won the RoboCup@Work world cup competition in 2017 and 2018. Detailed description of their hardware and software concepts are presented. The software section introduces the adopted solutions for the @work navigation, perception and manipulation tasks. Furthermore, improvements for future participation in the RoboCup world cup in Bordeaux 2020 are discussed.

1 Introduction

The RoboCup@Work league, established in 2012, focuses on the use of mobile manipulators and their integration with automation equipment for performing industrial-relevant tasks [6]. This paper presents our teams solution approaches and concepts. The main focus lies on the transition to a new base platform. This is necessary, because the current hardware is worn out and cannot be repaired due to inavailable spare components.

Chapter 3 shows the team's current system hardware and the future concept for this year. In chapter 4 the main software modules such as the state machine, localization and perception are presented. Finally, the conclusion provides a prospect to further work of team AutonOHM (chapter 5).

2 AutonOHM

The AutonOHM-@Work team at the University of Applied Sciences Nuremberg Georg-Simon-Ohm was founded in September 2014. In 2017, the team was able to win both the German (Magdeburg) and also the World Championship (Nagoya) title. With the knowledge and experience gained in the former tournaments, the team was also able to defend both of these titles in 2018.

As most of the teammembers of the sucessfull former team are not taking part in RoboCups anymore, the main goal last year was the knowledge transfer to the new teammembers, while also trying to implement new ideas and improve the overall system.

The new team consists of Bachelor and Master students, supervised by a former teammember and Master of Applied Research student. In addition, former teammembers will be involved in the training process of the new teammembers during 2020.

Fig. 1: Team AutonOHM 2020

As mentioned earlier, we have to built a new robot system, as the KUKA youbot, which is commonly used in the @work league, was discontinued and is not available for purchase anymore.

Fortunately, we are collaborating with the company Evocortex GmbH, which produces customizable omnidirectional platforms. We started in early 2019 with their alpha version of the Evorobot R&D platform and have been contributing to the development process ever since. Our goal is to provide an alternative robot platform for industrial applications such as the @work tasks, so new teams can jump the first barrier of creating their own robot to participate in the league.

Over the course of the season 2019, we have implemented and tested new features regarding both hard- and software. This resulted in the robot which will be presented in the following.

3 Hardware Description

We are using a customized Evocortex R&D platform with the smallest formfactor available. The platform is equipped with an omnidirectional mecanum drive, a aluminum chassis capable of carrying loads up to 100kg and a Li-Ion Battery with a nominal voltage of 24V and roughly 12.5Ah capacity. In our configuration, the platform does not come any sensors, power management or computation units, which means it only serves as our base. Every further component needed, like the manipulator or the pc, was mounted in or on the chassis.

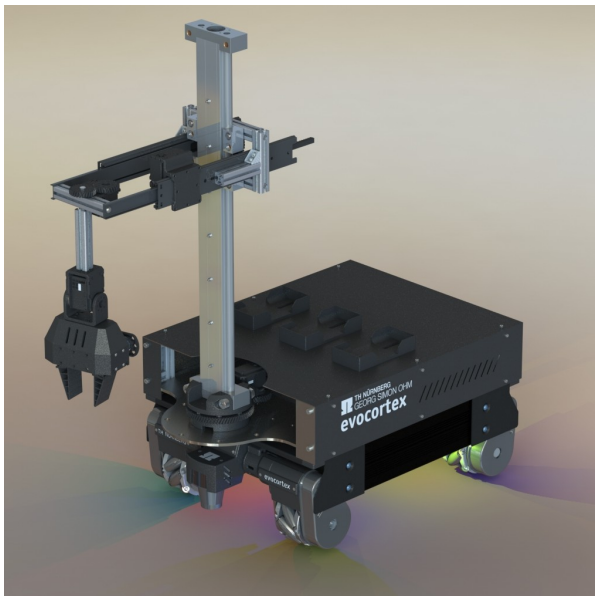

Fig. 2: CAD of our Ohmnibot

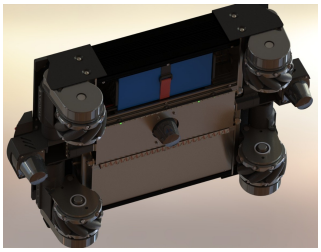

Fig. 3: Robot bottom

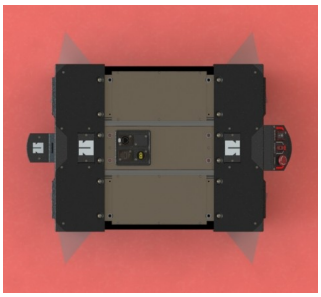

Fig. 4: Laser scan area

3.1 Sensors

Lidars We are using three SICK TiM571 2D lidars for mapping, navigation and obstacle avoidance. One each is mounted at the front and the back of the robot scanning 180°. The third lidar is mounted centered at the bottom of the robot scanning 90°to each side. With the third sensor we were able to eliminate the leftover dead zones, giving us a full 360°scan of the robots surroundings.

Cameras For image processing we use an Intel RealSense D435 3D-cameras. One is mounted on the manipulator so its field of view can be adjusted to the scene (e.g. object detection). One more each is mounted at the front and the back of the robot for the barriertape detection.

3.2 PC

In the season of 2019, we have been using two embedded PCs communicating via a LAN connection. The main PC was an Intel NUC i7 and was used for all main programs. The secondary support system was a Nvidia Jetson TX2 used for neural network computation. We replaced this combination with a custom single pc solution consisting of an AMD Ryzen 3700x processor and a Nvidia GTX 1650 graphics card. We faced issues with the two pc system, as its operation is much more complex and an autostart for the robot is much more difficult. Thats why we decided to come up with our custom solution, which is not only much more powerful, but also does it cost almost the same as the two smaller PCs combined.

3.3 PSU

We developed a custom psu circuit board containing emergency switches for the actuators, a main power switch and high efficiency voltage controllers for 5V and 12V. It is equipped with a custom designed plug system with selectable voltage, so every peripheral device can be connected using the same plug type. In addition to that, we use an adjustable DC-DC controller for the power supply of the manipulator, as its power consumption exceeds the limits of the onboard controllers. For the custom PC system we use a standard 250W automotive ATX power supply.

3.4 Manipulator

Arm As our robot platform does not include a manipulator, we had to search for alternative robot arms. Our enquirys showed that there are no alternatives to the youbot manipulator costing less than 10,000 EUR, which we are not able to afford. Thats why we had to develop a custom solution applicable in the @work league. Regarding the tasks mostly consisting of pick and place actions, we chose a kinematic concept similar to SCARA, which is commonly used in industrial scenarios for these tasks. Our aim was to develop an educational robot arm that is simple and affordable, so it can be manufactored and used by any student or research institute.

Our concept utilizes linear gears to control the z- and x-axis of the arm. Combined with the first rotational z joint, the TCP can be moved to every point (x,y,z) given within the operation area. For more flexibility, two additional rotational joints (y and z) were added between the TCP and the linear x axis to compensate for the object and arm orientation. The actuators we used are simple Dynamixel MX-106 and AX-64 motors which were available in our laboratory. They have enough power to control each axis, with the linear z axis being able to lift up to 5kg.

Most of the parts used were 3D printed using PETG material, including some of the main mounting parts and all gears. The main bearing, the linear rail and the full extension tray rails have to be purchased. Including the actuators, our current configuration sums up to about 2,500 EUR. We are planning to release the plans once the arm is fully developed.

Gripper Our gripper concept somehow relates to the arm concept, also using 3D printed linear gears. Its based on a single Dynamixel AX-12 motor connected to the driving gear. The power transmission enables the motor to grasp objects with its full torque rather than it being reduced by a lever with its length conditioned by the gripper fingers. The fin-ray fingers are custom printed out of rubber filament, making them soft and enabling them to close around grasped objects. They are also more wide than standard FESTO fin-ray fingers, so they have an enlarged attack surface and therefore have more tolerance for very small and/or moving objects.

3.5 Other

Inventory The robots inventory consists of three identical 3D printed slots. They are equipped with anti-slip pads, which prevent any movement of the objects, even with heavy robot vibrations. The individual slots are mounted on an adaptable rail system, which enables various mounting positions.

Network Setup Our network setup consists of multiple LAN switches onboard of the robot, connecting different mounting layers of the platform. A WLAN-router is used as a DHCP server while also giving wireless remote access to the robot.

RGB Feedback For additional feedback to its viewers, the robot is equipped with two RGB strips at its bottom. They are used to indicate the robots actions by blinking various color patterns.

4 Software Description

We use Linux Ubuntu 18.04 and ROS Melodic [4] as our operating systems. Over the last season, we refactored most of the old software architecture to regain flexibility and simplify the overall structure. Our new design is displayed in fig. 2.

The idea derives from the MVC pattern, which is adjusted to the ROS framework. Regarding the frequent use of hardware, an additional driver layer is added below the model layer. Models that need data from hardware, e.g. sensor data, can get them from the individual driver programs. The view layer is realized with each program using interfaces to RVIZ or simple console logging, which makes custom implementations obsolete. Components that require additional control features, such as the robot arm, have dedicated controllers providing simple interfaces for the brain layer, which is responsible for the actual task interpretation and execution. The individual layer components will be explained in the following sections.

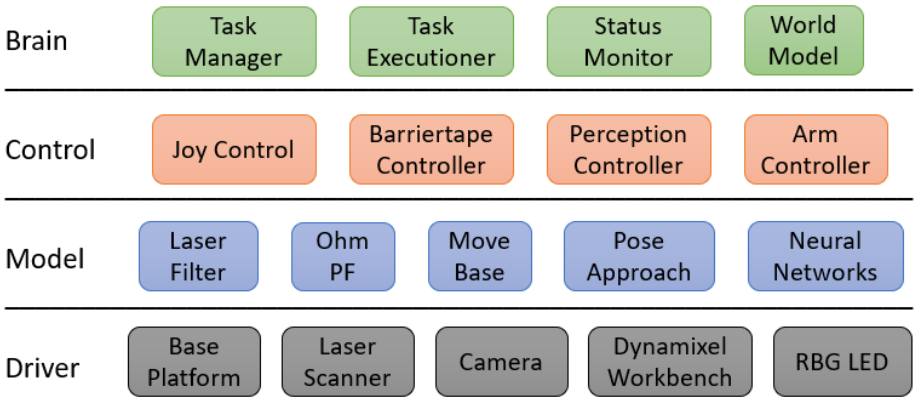

Fig. 2: Software Architecture - BCMD

4.1 Driver

The driver layer only contains actual hardware control programs, such as the sensor interfaces. The idea here is that the whole layer can be replaced with simulation tools such as Gazebo.

Base Platform The base platform driver converts incoming cmd_vel messages into wheel rpm and calculates the odometry from obtained rpm. It stops the robot automatically if the incoming commands time out to prevent uncontrolled movements. An additional twist_mux node throttles incoming commands from the joy_controller, move_base and the pose_approach.

Laser Scanner Three sick_tim nodes provide the interface to the scanners with given IP adress and scan area configuration. However, as the Lidar is prone to measurement errors such as shadows or reflections, custom laser filters are applied to the raw data for later computation.

Camera We use the Intel Realsense SDK with the provided ROS wrapper.

Dynamixel Workbench The tower arm is controlled with a controller instance of the dynamixel_workbench package. It provides a trajectory interface to control multiple motors at once, which we use for trajectory execution. As our gripper also uses a dynamixel motor, but needs extended access to motor variables (e.g. torque), a dedicated controller instance is used for the gripper controls and feedback.

RGB LED A Teensy Microcontroller runs a control loop for the LED strips. Depending on given brightness, color and mode, the individual LEDs are controlled to enlight, blink or fade.

4.2 Model

Our models contain all algorithms used to challenge the problems the tasks in the @work league. This includes localization, navigation and perception. The task planner is not included as a model but in the brain layer because it is more convenient to attach it directly to the task manager, as discussed in chapter 4.4.

Laser Filter As mentioned in 4.1, we filter the raw laser data before computing. The first filters are simple area filters to delete the robots wheels from the scan. Otherwise they would appear as obstacles in the navigation costmap. The second filter is a custom jumping point filter implementation. We faced problems with reflections of the alu profile rails used for the walls of the arena, which caused the robot to mark free space as occupied, even though the points were only single reflections. The filter calculates the x- and y-position for each scan point and checks if there are enough neighbors in close range to mark a point as valid. All points with less than n neighbors in the given range will be handled as measurement errors and therefore deleted.

Ohm PF For localization in the arena, we use our own particle filter algorithm. Its functionality is close to amcl localization, as described in [5] and [7]. The algorithm is capable of using multiple laser scanners and an omnidirectional movement model. Due to the Monte Carlo filtering approach, our localization is robust and accurate enough to provide useful positioning data to the navigation system. Positioning error with our particle filter is about 6 cm, depending on the complexity and speed of the actual movement.

Move Base We use the ROS navigation stack for global path planning and the local path control loops. For path cost calculation we use the costmap 2D plugins. Our base layer is a 2D laser map created with gmapping [3]. On top of that, we use a barriertape map layer which contains all detected barriertape points. For local obstacle avoidance, we added an obstacle layer which includes laser data from all three laser scanners. All layers are combined in the final inflation layer. Global pathplanning is computed by NavfnROS while the path is executed using the DWA local planner. As the local planner is not able to precisely navigate to a given goal pose, we set the goal tolerance relatively high. Once we reached our goal with move_base, we continue exact positioning with our custom controller, the pose_approach.

Pose Approach The pose_approach package utilizes a simple PID controller to move the robot to a given pose. It utilizes the robots localisation pose as input and the target pose as reference. As the controller does not consider costmap obstacles, the maximum distance to the target is 20 cm to prevent collisions. A laser monitor algorithm checks for obstacles in the current scan and stops the robot if necessary.

NN - Barriertape For the barriertape detection we use a U-Net with manually labelled datasets. The ROS node receives raw input images and returns a masked binary image with all barriertape pixels marked. We have ported the network node from Python to C++ to increase the detection rate from around 5Hz up to 20Hz.

NN - Objects The detection and classification of objects is done with a Tiny-YOLO-v3 network. The node receives a raw input image and returns a vector with the id, bounding box and weight of all objects that were found. As our dataset would require more than 10,000 labelled images, which would require a high amount of time to create, we have implemented an automated dataset creation method using Blender and Python. It basically changes objects, camera position and lighting for a modelled scene to render labelled images, which are quite similar to original scene images (fig. 3). Using a original to artificial image ratio of 1:10, we achieved a detection reliability of over 90% for most scenes. Our data generation scripts are public and free to use [1].

(a) (b) (c)

Fig. 3: Object Perception: real image (a) rendered image (b) result (c)

The trained network is converted to TRT-Engine using code from the TRT-YOLO-App from the Deepstream Reference Apps [2]. This increases performance as the CUDA cores will be used more efficient, and makes a detection rate of up to 60Hz possible.

4.3 Controller

Model nodes that require additional control features are connected to control nodes which then provide interfaces for the brain layer. They use our robot_custom_msgs interfaces to share information about the subtask, workstation or objects. Nodes may have specific subtask types implemented into their behaviour to react optimized.

Joy Control We use a PS3 joystick to move our robot manually (e.g. for mapping). For this, we have implemented a custom teleop_joy_node with similar functionality.

Barriertape Control The barriertape controller is a custom mapping implementation for visual obstacles. It throttles the input images to the barriertape network and computes the masked images. It is possible to connect multiple cameras to the controller, which will then be iterated in a loop.

Received masked images are converted into a point cloud with a predefined density. This pointcloud is then transformed from the individual camera frame into the global map frame. Afterwards, all new points are compared to the existing map points. New barriertape points that mark cells which are already occupied are ignored to save computation. As we faced problems with image blur and therefore resulting non precise barriertape detection, we also compute Pixels that mark free space (no barriertape detected). They are compared to existing points, which get deleted if they overlap.

The whole map is converted into a occupancy grid and then published periodically, so it can be included in the costmap of the move_base node. The node is controlled via service calls, which enable or disable the detection loop. The map is always published once the node finished the init process.

Arm Control As the kinematic model of the tower arm has only one solution for a given TCP position, we developed a custom arm controller node instead of using moveIt. It is possible to adjust the amount and type of joints and links via ROS parameters, only the inverse kinematics solution has to be adjusted for new arms. Using predefined danger zones, the arm executes a self calculated trajectory to the target pose considering the individual motor parameters. The arm is controlled via ROS services or a development GUI for debugging. When using the services, the arm executes a full task using the given information, which means, in case of a pick task, it moves the tcp to the object position, closes the gripper, and stores the object. After the subtask finishes, feedback of the exit status is returned to the caller.

Perception Control The perception control node is responsible for the workstation analysis and object detection. A given scene (3D Pointcloud and RGB image) is analyzed in multiple steps. First, the surface equation of the workstation is calculated using the RANSAC algorithm. If a valid result is obtained, raw images are sent to the object perception network (4.2). All found objects are then localized using the pinhole camera model, the workstation plane and the bounding box pixels. Finally, the position is transformed into the workstation frame and then saved for later usage. For moved objects, multiple positions are recorded and then used to calculate the movement equation with RANSAC.

4.4 Brain

Every node below the brain level needs external controls to fulfil tasks. The brain layer provides nodes which contain the intelligence of the robot, which means the tracking of itself, its environment and the received tasks.

Worldmodel All data obtained about the robots environment is stored in the worldmodel database. This includes the map, all workstation positions and all detected objects on the workstations. The data can be accessed using service calls.

Status Monitor The status monitor keeps track of the robot itself. It saves the current pose, inventory and state. The associated color code is sent to the RGB LED driver node.

Task Manager When the robot receives a new transportation task, it must be analyzed and planned before the execution starts. All extracted subtasks are managed by the task_manager node, which uses all information available to replan the order of all subtasks. With the increasing numbers of transportation tasks in the competition, high efficiency is crucial to achieve perfect runs.

The score of a single subtask is calculated considering expected duration, points and the risk of failure. These factors may change if certain conditions are met, for example, the navigation time is set to zero if the robot already is at the given position.

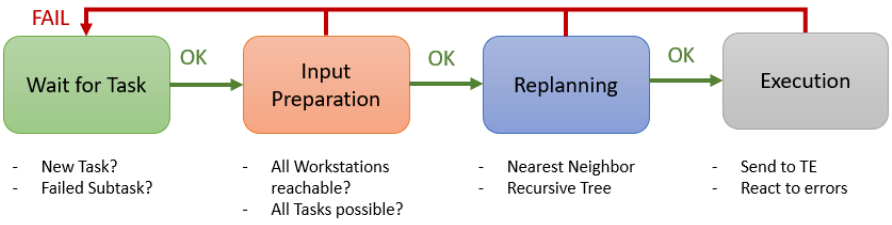

Fig. 4: Task Manager States

Before even starting the planning of subtasks, the received task is analyzed for impossible tasks. This would be the case if the target Workstation is unknown or unreachable, or an object is lost. All subtasks that cannot be executed are moved to a deletion vector. A self developed planning algorithm then calculates the raw score of the remaining subtask vector, followed by a simple nearest neighbor search (NN). This result is then used as input for a recursive tree calculation method which checks all possible combinations for the subtask and searches for the optimal solution. A branch is only fully calculated if the score sum does not exceed the best solution found with the NN. This way the have achieved a overall planning time for the final challenge (20 subtasks) of around 10s. For subtask numbers below 18 the planning only takes 2s. After that, the planned subtask vector is iterated and every subtask is sent to the task exectutioner (4.4). If the execution was not successful, the actual task is moved to a failed subtask vector and deleted from the current working STV. The short planning times enable us

to replan everytime a subtask fails or new data is available. This is necessary because even simple changes can cause serious errors in the intentional plan. If certain paths are blocked, the navigation time for transportation tasks can increase dramatically, causing a huge loss of efficiency. A final garbage collection checks all deleted and failed subtasks for plausibility again and adds retrys for possible subtasks.

Task Executioner Subtasks that are sent to the Task Executioner get run trough an interpreter to extract the actions that are necessary for the task execution. All actions are performed in custom states which can be adjusted via parameters at creation. The interpreter uses information form the status monitor, the worldmodel and the given subtask to create every substate. The resulting state vector is iterated until its finished or failed. While executing, the node reads and modifies the data in the status monitor and worldmodel package. This way every change is immediately available for all other nodes too.

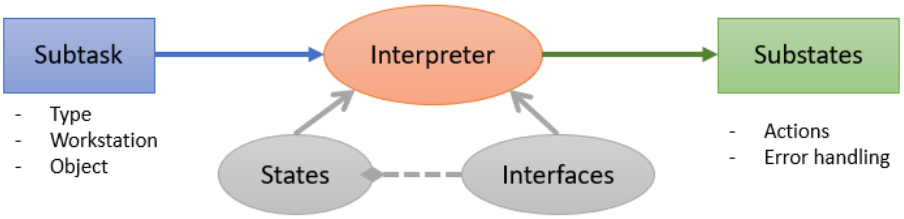

Fig. 5: Task Executioner - Subtask Interpretation

5 Conclusion and Future Work

The last season was primarily used to build the new team and the new robot from ground up while trying to transfer as much knowledge from the old systems as possible. As new challenges of the @work league made some of the old solutions inapplicable for future use, we had to develop and test new ideas. This caused us having a lot of trouble with prototypes and small mistakes, which led to the robot system being totally unreliable over the whole season. The ideas presented in this paper are the result of the last years effort. We have described our new system design, which we look forward to be competing with in the 2020 championships.

References

1. Github: Itsmethebee (sally zeitler).
2. Github: Nvidia deepstream reference apps.
3. Ros gmapping package.
4. Robot Operating System (ROS), 2016.
5. F Dellaert, D Fox, W Burgard, and S Thrun. Monte Carlo localization for mobile robots. *Proceedings 1999 IEEE International Conference on Robotics and Automation (Cat. No.99CH36288C)*, 2(May):1322–1328, 1999.
6. Gerhard K. Kraetzschmar, Nico Hochgeschwender, Walter Nowak, Frederik Hegger, Sven Schneider, Rhama Dwiputra, Jakob Berghofer, and Rainer Bischoff. RoboCup@Work: Competing for the Factory of the Future. pages 171–182. Springer, Cham, 2015.
7. S Thrun, W Burgard, and D Fox. *Probablistic Robotics*. Massachusetts Institute of Technology, 2006.