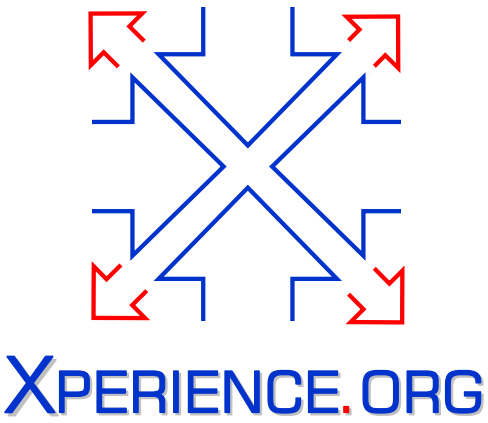
Project Acronym:	Xperience
Project Type:	IP
Project Title:	Robots Bootstrapped through Learning from Experience
Contract Number:	270273
Starting Date:	01-01-2011
Ending Date:	31-12-2015



Deliverable Number:	D5.3.5
Deliverable Title:	Final Demonstration on Scenario 2.
Type (Internal, Restricted, Public):	PU
Authors:	Tamim Asfour, Mikro Wächter, David Schiebener, Ekaterina Ovchinnikova, Simon Ottenhaus, Rüdiger Dillmann, Bojan Nemec, Aleš Ude, Wail Mustafa, Dirk Kraft, Norbert Kruger, Justus Piater, Emre Ugur, Chris Geib, Ron Patrick, Mark Steedman, Florentin Wörgötter, Alejandro Agostino, Giorgio Metta
Contributing Partners:	ALL

Contractual Date of Delivery to the EC:	31-12-2015
Actual Date of Delivery to the EC:	12-02-2016

Contents

Executive Summary

In D5.3.5 we now describe the final demonstration of the project which integrate methods and implementation from different workpackages and partners in the project. As such, the final demonstration combines contributions from WP5.2 and WP5.3. The contributions of WP5.2 to the main demo are described in more detail in D5.2.5.

The main demonstration is shown in the video **Xperience-Final-Video.mp4**, which emphasizes the link between the sensorimotor level and symbolic high level through the ArmarX memory architecture MemoryX.

We demonstrate the humanoid robot ARMAR-III performing complex tasks in interaction and collaboration with humans and employing mechanisms of structural bootstrapping towards efficient task execution. The robot is able to execute complex manipulation tasks, interact with humans, plan and reason about the world and perform actions to achieve given plan goals.

Together with the human, the robot is setting up the table for dinner for two people. Situation assessment and plan generation is performed on the fly based on previous experience. The demonstration highlights the aspects of the realization of integrated complete robot systems, and emphasizes the concept of structural bootstrapping on the levels of human-robot communication and physical interaction, sensorimotor learning, learning of object affordances, and planning in robotics.

Chapter 1

Main Structural Bootstrapping Demonstration

The scenario integrates several scientific methods developed in the project, which can be summarized as follows.

- Execution of complex manipulation tasks and plans based on the developed architecture and its implementation
- Automatic generation of domain descriptions for planning based on the robots experience
- Replanning on the fly in case of missing objects
- Replacing missing objects by employing different bootstrapping (replacement) strategies
- Replacing actions by adapting previously learnt actions to new context
- Human-robot communication in natural language including the robot's understanding of spoken commands, world descriptions, and feedback as well as the robot's ability to ask the human for help and information
- Handing over objects between the robot and the human

The demonstration elaborates on the dinner preparation scenario, which consists of the five main parts:

1. setting the table
2. preparing a salad
3. cleaning the sideboard
4. arranging chairs
5. bringing a drink

Figure 1.1 depicts different scenes from the demonstrations scenario showing the robot and the human preparing dinner together.

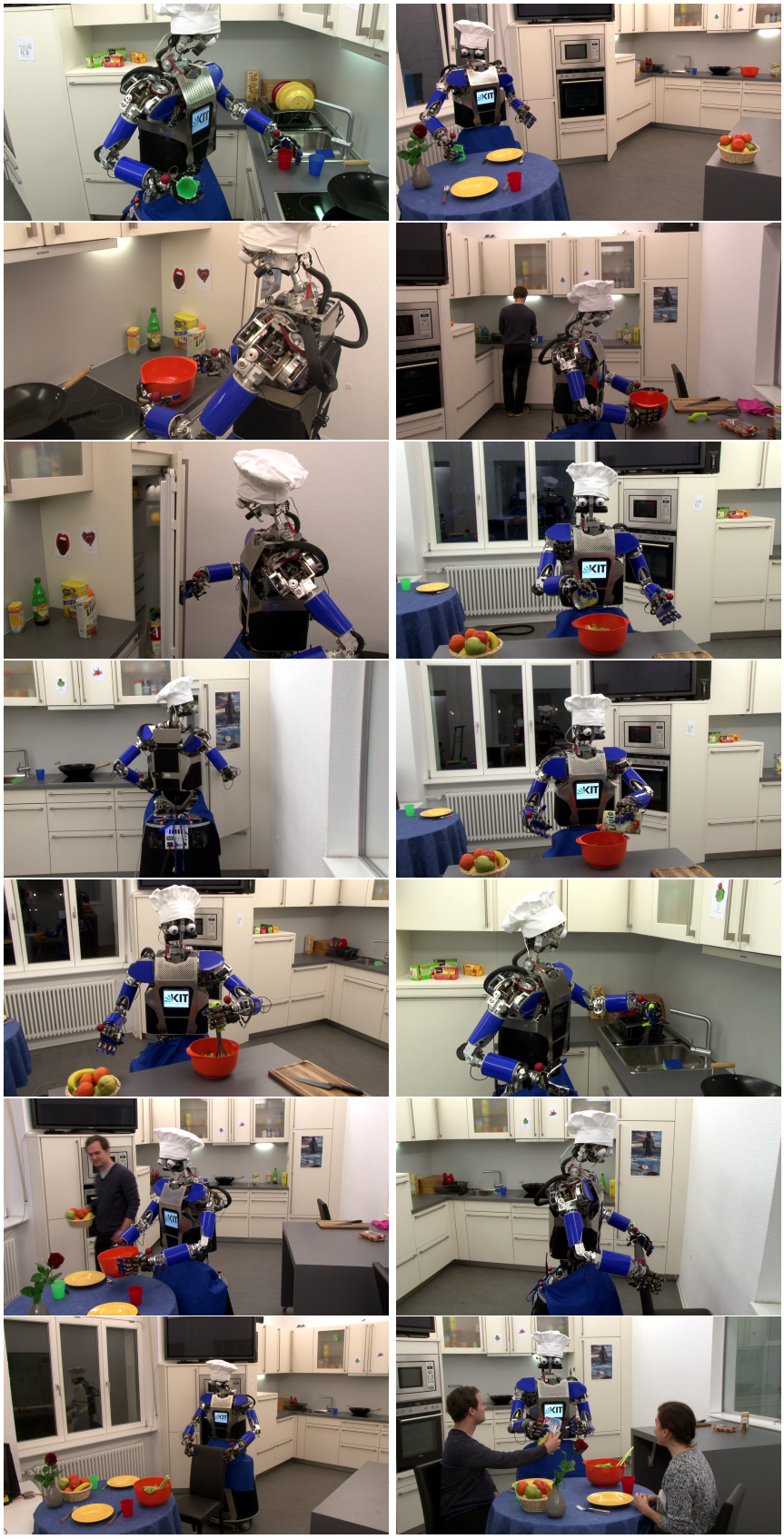


Figure 1.1: Snapshots of cooperative rearranging the room and preparing a dinner.

In the following sections we describe the individual scientific contributions in relation to this demonstration scenario.

1.1 Task Execution Architecture (KIT, UEDIN, UGOE)

The updated underlying system architecture for the demonstration is shown in Figure1.2. Given a spoken command, the Language Understanding component (KIT [OWWA15]) generates a goal for the PKS planner (UEDIN [10]). The goal is passed to the Replacement Management (RM) component (implemented at KIT using replacement strategies based on methods developed at KIT, SDU, UIBK) that checks for each object in the goal if this object and its location are in domain description generated from the robot's memory MemoryX (KIT [14]). This process is shown in Figure 1.3. If any replacements are required, then the goal is rewritten. The new goal is passed to the PKS planner together with the domain description. The PKS planner generates a plan. The execution of the plan is monitored by the Statechart framework of ArmarX (KIT [WOK+16]). If the plan execution fails because of the missing object, the RM component is evoked to make a replacement and PKS replans to generate a new plan. The LU component also directly interacts with MemoryX, when a world state description is processed. LU provides a command directly to the Plan execution & Monitoring component if a command that does not require planning has been uttered (e.g. *Stop*).

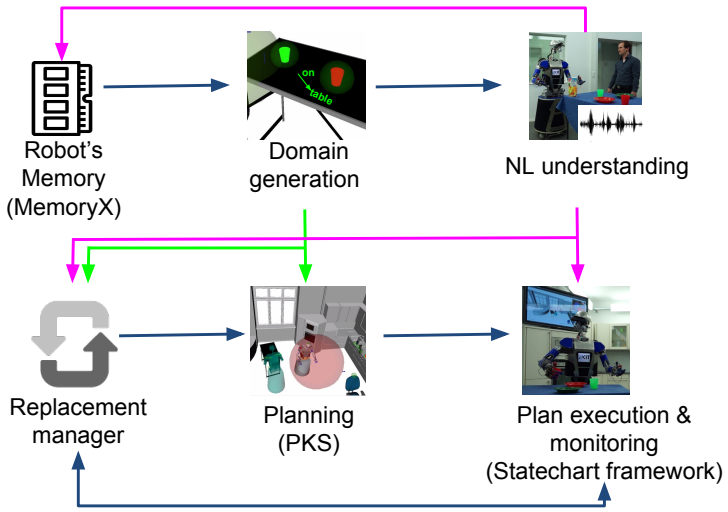


Figure 1.2: Interaction between Language Understanding, Replacement Manager, PKS planner, domain generation, and Statechart framework.

The setting the table part of the scenario corresponds to the demonstration in Y4 (UGOE [1]). This was extended by Language Understanding (LU) and MemoryX and integrated in the final demonstration. The human agent asks the robot to help him set the table for two people. The execution of the uttered command requires generation of the multi-step plan. The robot generates a plan resulting in putting two cups on the table, while the human agent puts forks, knifes, and plates on the table. The goal description for the planner is provided by the Language Understanding component. The domain description is generated from the working memory of the robot (MemoryX). The goal and domain descriptions constitute the input for the PKS symbolic planner, which generates a plan. The execution of the plan is monitored by the Statechart framework of ArmarX. The grasping and placing skills depend on object models and grasp definitions in MemoryX.

1.2 Gaze selection (KIT)

During manipulation action like grasping and placing, in particular when it is bimanual, the robot needs to visually localize the object(s) as well as its hands to reach the necessary degree of accuracy for the motion of the hands with relation to the object. Throughout the whole demonstration, the gaze direction is chosen autonomously by the robot depending on its uncertainty about the pose of the currently relevant

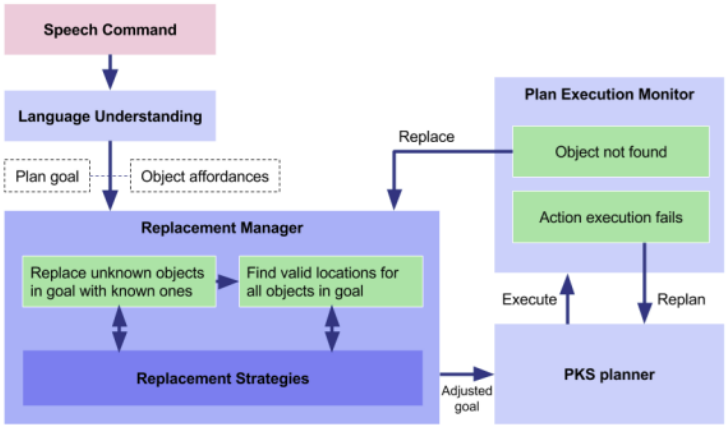


Figure 1.3: The Replacement Manager (RM) functions as a preliminary feasibility check of a given goal for the planning component. It checks the existence and their location of the requested objects. If the RM encounters a missing object or an unknown location it attempts to find a location for an object or replaces the object altogether (KIT [OWWA15]).

objects (KIT [13]). Figure 1.4 shows ARMAR-III during a bimanual grasping process and depicts the object position uncertainties.

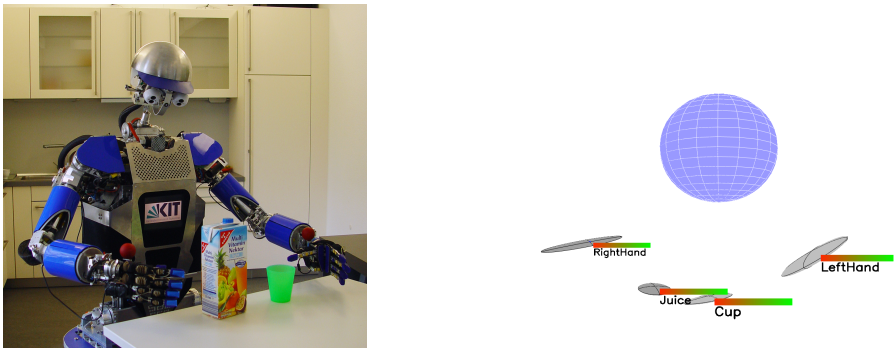


Figure 1.4: The robot selects its gaze direction depending on the position uncertainties of the currently relevant objects. In this example, two objects are grasped simultaneously, so both objects and the hands need to be localized continuously during grasp execution. The ellipsoids on the right visualize their respective position uncertainties.

The pose of the hands, while they are not in the current field of view, is updated based on forward kinematics, but the uncertainty is increased the further they move. Objects are typically considered to be static in the world, but their location relative to the robot, which is crucial for manipulation, becomes increasingly uncertain when the robot moves. Depending on those uncertainties a gaze direction is selected such that objects with a high uncertainty are brought into the field of view and can be localized. Correspondingly, manipulation skills that rely on precise pose information will wait for the uncertainty to be reduced by further localization results before continuing execution.

1.3 Shape-based affordance estimation for object replacement (SDU, KIT)

In the salad preparation part, the human agent first asks the robot to put the salad bowl on the sideboard. The command is processed by the LU component, the resulting goal and the domain description are processed by the PKS planner, which generates a plan. According to the plan, the robot moves to the location of the salad bowl, but does not find the required object at the location. The Plan Execution & Monitoring component reports the failure in the plan execution.

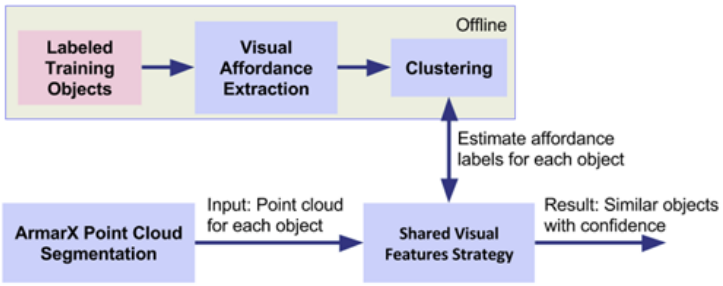


Figure 1.5: Object replacement based on visual shared affordances, which are deduced from shape similarities.

The Replacement Manager (RM) component is invoked. It calls the component for visual object replacement (see Figure 1.5) which detects another object that, comparing its shape to known objects with known affordances, appears to afford similar actions ([7], [8]). The RM suggests the new container as a replacement based on these shared visual features. The goal is rewritten accordingly and passed to the PKS planner, which produces a new plan that can be executed successfully.

1.4 Bimanual grasping (KIT, JSI)

The bowl needs to be grasped and lifted bimanualy using strategies developed in the project (KIT, JSI [4]). In this situation the gaze selection described above is particularly important, as for most of the grasp execution the robot can not see the bowl and both hands at once. Grasp success is checked using the force sensors in the wrist of the robot to ensure that the object is robustly fixed between the hands. During bimanual lifting and later placing, the force sensor is used as a feedback source to keep the grasp stable while moving.

1.5 Robot Skills: Opening and closing doors, grasping, pouring, placing, handover (KIT)

When the bowl is on the table, the human agent asks the robot for help in preparing a salad with corn and oil. The command is processed by LU and a goal is generated, which implies the corn and the oil being in the bowl and the salad being stirred in the bowl. RM finds that the location of the corn is unknown. Since other location replacement strategies fail, it generates a question for the human inquiring for the corn location. The human tells the robot that the corn is located in the fridge. The utterance is processed by LU and MemoryX is updated correspondingly. After obtaining the feedback from human, RM evokes the PKS planner that generates a plan. The robot moves to the fridge, opens it, moves to the sideboard, pours the corn into the bowl, puts the empty can into the sink, and returns to the fridge to close it. The actions of putting the can into the sink and closing the fridge are planned by PKS, since we introduce symbolic rules stating that dirty objects should go into the sink after being manipulated and that the fridge door should be closed at the end of each plan execution. After adding the corn, the robot moves to the oil location, grasps the oil, pours it into the bowl, and puts the oil bottle away. The latter action is also planned by PKS, since we define a symbolic rule requiring robot's hands to be empty at the end of each plan execution. In the meanwhile, the human is cutting other salad ingredients and pouring them into the bowl.

In order to stir the salad, the robot requires a stirrer. It moves to the assumed stirrer location, but it cannot grasp the stirrer. Instead of grasping, the planner plans a speech help request from the human. The human passes the stirrer to the robot. The robots returns to the bowl, stirs the salad based on the previously learnt wiping action, and puts the stirrer into the sink. Finally, the human asks the robot to put the bowl on the dining table, which is performed by the robot.

1.6 Force-adapted Dynamic Movement Primitives(KIT, JSI)

In the cleaning part of the scenario, the human asks the robot to clean the sideboard. According to the generated plan, the robot requires a sponge. It moves to the location of the sponge, but it cannot grasp it. Similarly to the situation with the stirrer, a speech request is produced. The human passes the sponge to the robot that returns to the sideboard and wipes it. The wiping skill trajectory was learned from human demonstration and is executed by the robot by incorporating the force-torque to adjust the pressure on the table (KIT [2], [3]). Since the wiping action labeled the sponge as dirty the robots puts the sponge into the sink afterwards.

1.7 Part-based chair localization, bimanual grasping and pushing (UGOE, KIT, JSI)

The arranging chairs part of the scenario corresponds to the demonstration in Y4. his was extended by Language Understanding (LU) and MemoryX and integrated in the final demonstration. The human asks the robot for help in arranging chairs. The robot pushes one chair to the table, while the human is arranging the second chair.

Chair recognition and localization is realized based on the segmentation of perceived point cloud data into convex segments (UGOE [12, 11]). The constituent parts of the chair are detected based on their size and geometric relation, in particular the backrest which is then grasped bimanually (KIT, see above). The chair is moved to its designated target pose next to the table (JSI).

1.8 Common sense object replacement (KIT, UIBK)

In the final part of the scenario, the human is asking for a drink saying *I'd like to drink something. Could you please bring me a soda.* This command is processed by the LU component and a goal of the soda being in the hand of the human is generated. Apart from generating the goal, LU extracts the affordance of drinking for the object "soda". The goal and the predicted affordances are passed to Replacement Manager. RM finds that the object "soda" is unknown and attempts a replacement by using the affordance "drink". It uses a replacement strategy that uses an affordance database, which was filled from text mining (KIT [5]) and post-processing by JointSVM (UIBK [6]). The process of this strategy is depicted in 1.6. The object "multivitamin juice" is proposed as a possible replacement. RM generates a confirmation question *Sorry, I have no soda. Would an orange juice be fine?* After the human confirms the replacement, RM rewrites the goal and passes it to the planner.

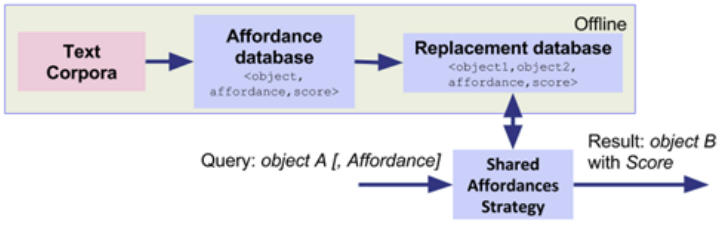


Figure 1.6: Object replacement based on shared affordances, which are deduced from text mining and machine learning with JointSVM.

1.9 Common object locations based on experience (KIT)

According to the generated plan, the robot moves to the assumed location of the juice, but does not find it there. This assumed location is always the location the robot has seen this object most often (see Figure 1.7) (KIT [14]). The locations were learned during past trials and exploration runs and then stored as density distributions. Due to the missing object the plan execution fails and RM is evoked

again. The RM component derives another potential location of the juice from the database of common locations. The robot finds the juice at the new location, grasps it, moves to the location of the human, and hands the juice over to the human.

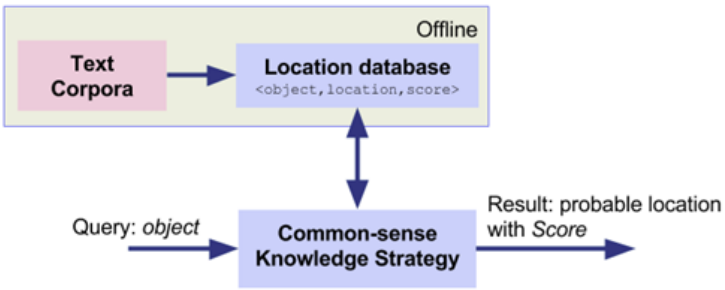


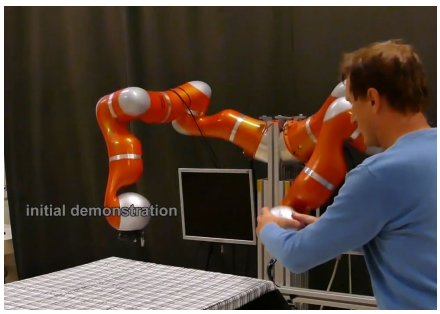
Figure 1.7: Object replacement based on the experience of the robot. The robot stores the locations where it has seen objects and generates a heat map of the most probable places.

Chapter 2

Further Contributions

2.1 Bimanual Programming by Demonstration and Cooperative Adaptation

Learning of dual arm behaviours based on bimanual programming by demonstration (PbD) and iterative learning control (ILC) framework is demonstrated in video **BimanualPegInHole.mov**. The first part of the video shows PbD of bimanual Peg In Hole operation. During the demonstration, only the relative motion of the two arms has been captured. During the execution, the robot optimizes its absolute coordinates in order to minimize joint velocities and consequently the PiH is not executed at the same absolute coordinates as demonstrated. After the initial demonstration, the right pole was displaced by 5mm in local y coordinate, which caused increased forces during the pole insertion. The robot successfully adapted to the new pole geometry in few repetitions using ILC framework. Adaptation to the new pole geometry together with the resulting relative forces are shown in the second part of video **BimanualPegInHole.mov**. This work is described in more detail in the attached paper [LNv+15].



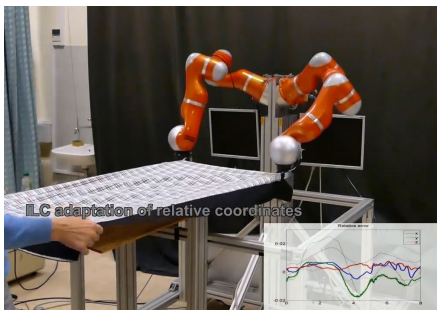


Table 2.1: Demonstration and cooperative execution of table cloth placement

Bimanual adaptation scheme applied to human-robot cooperation is presented in **TableCloth.mp4** and Figure 2.1. First, the initial demonstration of the table cloth placing was performed by the kinesthetic guidance of the bimanual system. Next, the learned motion was applied to place the table cloth in the same workspace configuration. Using the ILC adaptation scheme in absolute coordinates, the robot can effectively adapt to the new table orientation, as shown in the continuation of the video. Human-robot cooperation requires compliant robot operation for safety issues, which degrades the tracking performance in relative coordinates. The last part of the video shows, that this degradation can be effectively cancelled out using ILC framework.

References

[1] A. Agostini, M.J. Aein, S. Szedmak, E.E. Aksoy, J. Piater, and F. Wörgötter. Using Structural Bootstrapping for Object Substitution in Robotic Executions of Human-like Manipulation Tasks. In *IEEE/RSJ Int. Conf. on Intelligent Robots and Systems (IROS)*, pages 6479–6486, 2015.

[2] M. Do, J. Schill, J. Ernesti, and T. Asfour. Learn to wipe: A case study of structural bootstrapping from sensorimotor experience. In *IEEE International Conference on Robotics and Automation (ICRA)*, 2014.

[3] A. Gams, T. Petrič, M. Do, B. Nemec, J. Morimoto, T. Asfour, and A. Ude. Adaptation and coaching of periodic motion primitives through physical and visual interaction. *Robotics and Autonomous Systems*, 75, Part B:340 – 351, 2016.

[4] Andrej Gams, Bojan Nemec, Leon Žlajpah, Mirko Waechter, Auke Ijspeert, Tamim Asfour, and Aleš Ude. Modulation of motor primitives using force feedback: Interaction with the environment and bimanual tasks. In *IEEE/RSJ International Conference on Intelligent Robots and Systems (IROS)*, pages 5629–5635, Tokyo, Japan, 2013.

[5] Peter Kaiser, Mike Lewis, Ronald P. A. Petrick, Tamim Asfour, and Mark Steedman. Extracting common sense knowledge from text for robot planning. In *Proceedings of the IEEE International Conference on Robotics and Automation (ICRA 2014)*, pages 3749–3756, Hong Kong, China, 2014.

[6] Senka Krivić, Sandor Szedmak, Hanchen Xiong, and Justus Piater. Learning missing edges via kernels in partially-known graphs. In *European Symposium on Artificial Neural Networks, Computational Intelligence and Machine Learning*, 2015.

[7] Wail Mustafa, Dirk Kraft, and Norbert Krüger. Extracting categories by hierarchical clustering using global relational features. In Roberto Paredes, Jaime S. Cardoso, and Xosé M. Pardo, editors, *Pattern Recognition and Image Analysis*, volume 9117 of *Lecture Notes in Computer Science*, pages 541–551. Springer International Publishing, 2015.

[8] Wail Mustafa, Hanchen Xiong, Dirk Kraft, Sandor Szedmak, Justus Piater, and Norbert Krüger. Multi-label object categorization using histograms of global relations. In *International Conference on 3D Vision*, 2015.

[9] Ekaterina Ovchinnikova, Mirko Wächter, Valerij Wittenbeck, and Tamim Asfour. Multi-purpose natural language understanding linked to sensorimotor experience in humanoid robots. In *IEEE/RAS International Conference on Humanoid Robots (Humanoids)*, pages 0–0, 2015.

[10] Ronald P. A. Petrick and Andre Gaschler. Extending knowledge-level planning with sensing for robot task planning. In *Proceedings of the Workshop of the UK Planning and Scheduling Special Interest Group (PlanSIG 2014)*, Darlington, United Kingdom, December 2014.

[11] M. Schoeler and F. Worgotter. Bootstrapping the semantics of tools: Affordance analysis of real world objects on a per-part basis. *Autonomous Mental Development, IEEE Transactions on*, PP(99):1–1, 2015.

[12] S. Stein, F. Wörgötter, M. Schoeler, J. Papon, and T. Kulvicius. Convexity based object partitioning for robot applications. In *IEEE International Conference on Robotics and Automation (ICRA)*, pages 3213–3220, 05 2014.

[13] K. Welke, D. Schiebener, T. Asfour, and R. Dillmann. Gaze selection during manipulation tasks. In *IEEE International Conference on Robotics and Automation (ICRA)*, pages 652–659, Karlsruhe, Germany, May 2013.

[14] Kai Welke, Peter Kaiser, Alexey Kozlov, Nils Adermann, Tamim Asfour, Mike Lewis, and Mark Steedman. Grounded spatial symbols for task planning based on experience. In *Proceedings of the IEEE-RAS International Conference on Humanoid Robots (Humanoids)*, pages 484–491, Atlanta, GA, USA, 2013.

Attached Articles

[LNv+15] Nejc Likar, Bojan Nemec, Leon Žlajpah, Shingo Ando, and Aleš Ude. Adaptation of bimanual assembly tasks using iterative learning framework. In *IEEE-RAS International Conference on Humanoid Robots (Humanoids)*, pages 771–776, Seoul, Korea, 2015.

[OWWA15] Ekaterina Ovchinnikova, Mirko Wächter, Valerij Wittenbeck, and Tamim Asfour. Multi-purpose natural language understanding linked to sensorimotor experience in humanoid robots. In *IEEE/RAS International Conference on Humanoid Robots (Humanoids)*, pages 0–0, 2015.

[WOK+16] M. Wächter, S. Ottenhaus, M. Kröhnert, N. Vahrenkamp, and T. Asfour. The armarx statechart concept: Graphical programming of robot behaviour. *Frontiers - Software Architectures for Humanoid Robotics (submitted)*, 2016.

2015 IEEE-RAS 15th International Conference on Humanoid Robots (Humanoids)
November 3 - 5, 2015, Seoul, Korea

Adaptation of Bimanual Assembly Tasks using Iterative Learning Framework

Nejc Likar, Bojan Nemec, Leon Žlajpah, Shingo Ando and Aleš Ude

***Abstract*— The paper deals with the adaptation of bimanual assembly tasks. First, the desired policy is shown by human demonstration using kinesthetic guidance, where both trajectories and interaction forces are captured. Captured entities are portioned to absolute and relative coordinates. During the execution, small discrepancies in object geometry as well as the influence of an imperfect control can result in large contact forces. Force control can diminish the above mentioned problems only to some extent. Therefore, we propose a framework that iteratively modifies the original demonstrated trajectory in order to increase the performance of the typical assembly tasks. The approach is validated on bimanual peg in a hole task using two KUKA LWR robots.**

***Index Terms*— bimanual manipulation, iterative learning control, task adaptation, real time control**

I. Introduction

Bimanual arm architecture enables the performance of a variety of assembly tasks, is essential for carrying heavy and spacious objects and enables the transfer of many human skills to robots. However, additional flexibility requires more complex control algorithms. Dual arm manipulation has been extensively investigated in the nineties. Earlier control architectures exploited master-slave approach, hybrid-force-torque and impedance control approach to synchronize motion of both arms [1], [2]. Today, most of the bimanual control architectures are based on the concept of symmetric control [3], which enables portioning of the task to so called absolute coordinates and relative coordinates and underlying internal and external forces, which are orthogonal [4]. This formalism allows the learning, demonstration and adaptation of bimanual tasks in above mentioned orthogonal subspaces. Adaptation, being the one of the key features of new generation of service and humanoid robots, can be accomplished in several ways. In most cases the adaptation is required to refine the previously demonstrated motion to different robot embodiment. Additionally, adaptation is necessary to cope with non-modelled environment constraints. Often applied paradigm for motion adaptation is reinforcement learning (RL) applying probabilistic algorithms [5], which can deal with high dimensionality spaces induced by parameterised policies [6]. Despite of these advances, learning capabilities of modern robots are still far from the learning capabilities of humans. While humans can quickly adapt to new situations, robots often have to relearn the whole policy in a lengthy exploration process, even when a good initial policy approximation is provided. Therefore, researchers are trying to find effective solutions to speed up learning. One of promising paradigms is also Iterative Learning Control (ILC). The main objective of ILC is to improve the behavior of the control system that operates repeatedly by iterative refinement of the feed-forward control input [7]. Due to its simplicity, effectiveness and robustness when dealing with repetitive operations, ILC is often applied in robotics [8]. As many tasks in industry as well as in home environments need to be executed repeatedly, it represents a natural choice for adaptation of such tasks.

In this paper we propose a new learning controller that applies to bimanual task adaptation. Bimanual adaptation was studied also in [9], where both robot arms were independent agents coupled only at the force level. In this work, robot arms are coupled both on kinematical and force level and treated as a single agent. The structure of the proposed algorithm allows easy integration into the Dynamic Motion Primitives (DMP) framework [10]. The proposed algorithm is general and can be used with both types of bimanual movements that can be represented by DMPs, i. e. discrete and periodic movements. Our experimental setup was composed of two KUKA LWR arms equipped with Barret hands. The performance of the proposed algorithm was evaluated on long poles insertion task, which is related to the classical peg-in-hole problem [11], [12], [13], [14], [15]. The paper is organized as follow. In Section II we outline kinematics and dynamics of a bimanual system. In Section III, the main contribution of the paper, we extend our previously presented trajectory adaptation scheme based on demonstrated position and force profiles to a bimanual system. We discuss also the stability of the proposed adaptation scheme. In Section IV the experimental results and the effectiveness of the proposed algorithm are given. Discussion and future work regarding bimanual adaptation are summarized in conslusion.

*This research was partially supported by EU Seventh Framework Programme grant 270273, Xperience.

Nejc Likar, Bojan Nemec, Leon Žlajpah, and Aleš Ude are with Humanoid and Cognitive Robotics Lab, Department of Automatics, Biocybernetics and Robotics, Jožef Stean Institute, Ljubljana, Slovenia,(e-mails: nejc.likar@ijs.si, bojan.nemec@ijs.si, leon.zlajpah@ijs.si, ales.ude@ijs.si) Shingo Ando is with Robotics Technology Group, Yaskawa Electric Corporation, Kitakyushu, Japan, (e-mail: shingo.ando@yaskawa.co.jp)

978-1-4799-6885-5/15/$31.00 ©2015 IEEE

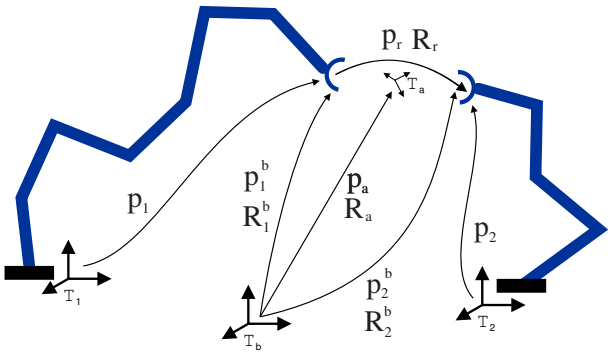


Fig. 1. Dual arm manipulator and the corresponding notation used in the paper.

II. Bimanual Task Kinematic Control

In this section we present a task-space control scheme for a bimanual system. This scheme is an extension of the previously proposed approach[16]. It fully characterizes a cooperative operational space and allows the user to specify the task in terms of geometrically meaningful motion variables defined at the position/orientation level [17], [16]. The resulting definition of the task variables in terms of the relative and absolute task motion of the cooperative system are mathematically well defined and have a clear physical meaning. Within this framework, both subspaces are orthogonal and thus decoupled - motion in relative coordinates does not affect absolute coordinates and vice versa. Consequently the control can be applied to both subsystems, relative and absolute, independently.

The key of our approach is in the definition of the common base coordinate systems for both subspaces, as illustrated in Fig. 1. According to this, the common base for the absolute coordinates is suitably chosen base $\mathcal{T}_b$ which applies to both robots, whereby the base for relative coordinates is placed in one of the robot's end effector, e.g. of the first robot. From now on we will use the notation where superscript j, $j \in \{1, 2, b\}$ denotes that the given quantity is specified relative to the coordinate system $\mathcal{T}_j$, while the subscript i, $i \in \{1, 2\}$ denotes the arm of a bimanual system and i, $i \in \{a, r\}$ denotes relative and absolute coordinates

According to this notation, absolute and relative task coordinates can be specified as

$$\begin{aligned} \mathbf{p}_r &= \mathbf{p}_2^1 = {\mathbf{R}_1^b}^T(\mathbf{p}_2^b - \mathbf{p}_1^b), && (1)\\ \mathbf{R}_r &= \mathbf{R}_2^1 = {\mathbf{R}_1^b}^T\mathbf{R}_2^b, && (2)\\ \mathbf{p}_a &= \frac{1}{2}(\mathbf{p}_1^b + \mathbf{p}_2^b), && (3)\\ \mathbf{R}_a &= \mathbf{R}_1^b\mathbf{R}_{\mathbf{k}_{21}^b}^b(\vartheta_{21}/2), && (4) \end{aligned}$$

where $\mathbf{p} \in \mathbb{R}^3$ applies to positions vector and $\mathbf{R} \in \mathbb{R}^{3\times3}$ to rotational matrices. $\mathbf{k}$ and ϑ_{21} are the axis and angle that realize the rotation $\mathbf{R}_1^b$ to $\mathbf{R}_2^b$. In quaternion notation, (2) and (4) are in the form

$$\begin{aligned} \mathbf{q}_r &= \mathbf{q}_2^1 = \bar{\mathbf{q}}_1^b * \mathbf{q}_2^b, && (5)\\ \mathbf{q}_a &= \mathbf{q}_1^b * \mathbf{q}_{\mathbf{k}_{21}^b}^b, && (6) \end{aligned}$$

where the quaternion $\mathbf{q}_1^b \in \mathbb{R}^4$ and $\mathbf{q}_2^b \in \mathbb{R}^4$ expresses the rotation of the TCP of the first and second robot in the common base coordinate frame $\mathcal{T}_b$, respectively. $\bar{\mathbf{q}}$ denotes conjugate quaternion and operator $*$ denotes quaternion product. $\mathbf{q}_{\mathbf{k}_{21}^b}^b$ denotes the unit quaternion corresponding to $\mathbf{R}_{\mathbf{k}_{21}^b}^b(\vartheta_{21}/2)$, which can be calculated from

$$\mathbf{q}_{\mathbf{k}_{21}^b}^b = \left(\cos\left(\frac{\vartheta_{21}}{4}\right), \mathbf{k}_{21}^b \sin\left(\frac{\vartheta_{21}}{4}\right)\right) \quad (7)$$

.

The corresponding relative and absolute forces and toques are

$$\begin{aligned} \mathbf{f}_r &= \frac{1}{2}(\mathbf{f}_1^1 - \mathbf{R}_r\mathbf{f}_2^2) && (8)\\ \mathbf{m}_r &= \frac{1}{2}(\mathbf{m}_1^1 - \mathbf{R}_r\mathbf{m}_2^2), && (9)\\ \mathbf{f}_a &= \mathbf{R}_1^b\mathbf{f}_1^1 + \mathbf{R}_2^b\mathbf{f}_2^2 && (10)\\ \mathbf{m}_a &= \mathbf{R}_1^b\mathbf{m}_1^1 + \mathbf{R}_2^b\mathbf{m}_2^2, && (11) \end{aligned}$$

where $\mathbf{f}_i^i \in \mathbb{R}^3$ and $\mathbf{m}_i^i \in \mathbb{R}^3$ denote the forces and torques measured at the i-th manipulator tool center point (TCP).

In order to control the robot, we have to map the desired relative and absolute task coordinates to the corresponding joint coordinates of both robots, denoted with $\boldsymbol{\theta} \in \mathbb{R}^{(N_1+N_2)}$, where N_1 and N_2 is the number of joints of the first and the second robot, respectively. This transformation is obtained through relative and absolute geometrical Jacobian, which maps the corresponding translational and angular velocities to the joint velocities

$$\begin{bmatrix} \dot{\mathbf{p}}_r \\ \boldsymbol{\omega}_r \end{bmatrix} = \mathbf{J}_r\dot{\boldsymbol{\theta}} \quad , \quad \begin{bmatrix} \dot{\mathbf{p}}_a \\ \boldsymbol{\omega}_a \end{bmatrix} = \mathbf{J}_a\dot{\boldsymbol{\theta}}. \quad (12)$$

Relative and absolute Jacobian matrices are obtained with the time derivation of the set of equations (1–4),

$$\mathbf{J}_r = \begin{bmatrix} -{\mathbf{R}_1^b}^T(\mathbf{J}_{1,p} + \mathbf{S}^T(\mathbf{p}_2 - \mathbf{p}_1)\mathbf{J}_{1,\omega}) & {\mathbf{R}_1^b}^T\mathbf{J}_{2,p} \\ -{\mathbf{R}_1^b}^T\mathbf{J}_{1,\omega} & {\mathbf{R}_1^b}^T\mathbf{J}_{2,\omega} \end{bmatrix} \quad (13)$$

and

$$\mathbf{J}_a = \begin{bmatrix} \frac{1}{2}\mathbf{J}_1 & \frac{1}{2}\mathbf{J}_2 \end{bmatrix}. \quad (14)$$

Subscript $(.)_p$ and $(.)_\omega$ denotes positional and rotational part of the Jacobian and $\mathbf{S}$ is anti-symmetric matrix [4].

If the task requires control only of the relative coordinates, the corresponding joint velocities are obtained from

$$\dot{\boldsymbol{\theta}} = \mathbf{J}_r^+(\mathbf{v}_{r,d} + \mathbf{K}_r\mathbf{e}_r) + (\mathbf{I} - \mathbf{J}_r^+\mathbf{J}_r)\dot{\boldsymbol{\theta}}_0, \quad (15)$$

where $\mathbf{J}_r^+$ is the Moore-Penrose pseudo-inverse of the relative Jacobian $\mathbf{J}_r$, $\mathbf{v}_{r,d} \in \mathbb{R}^6$ are the desired relative translational and rotational velocities, $\mathbf{I}$ is identity matrix, $\mathbf{K}_r \in \mathbb{R}^{6\times6}$ is a diagonal matrix with the kinematic gains and $\mathbf{e}_r \in \mathbb{R}^6$ is the error between the desired and actual relative task coordinates, calculated as

$$\mathbf{e_r} = \begin{bmatrix} \mathbf{p}_{r,d} - \mathbf{p}_r \\ \log(\mathbf{q}_{r,d} * \bar{\mathbf{q}}_r) \end{bmatrix}. \quad (16)$$

The rotational part of the error is calculated using logarithmic map $\log$, which maps the quaternion describing the rotation between the desired and current pose to the rotation error vector. This mapping is defined as

$$\log(\mathbf{q}) = \log(v, \mathbf{u}) = \begin{cases} \arccos(v)\dfrac{\mathbf{u}}{\|\mathbf{u}\|}, & \mathbf{u} \neq 0 \\ [0,0,0]^{\mathrm{T}}, & \text{otherwise} \end{cases}. \quad (17)$$

Vector $\dot{\boldsymbol{\theta}}_0 \in \mathbb{R}^{(N_1+N_2)}$ is an arbitrary vector of joint velocities that is projected in the null-space of the primary task, selected in such a way that it optimizes an additional secondary task, i.e. obstacle avoidance, joint limit avoidance, singularity avoidance, etc.

If we would like to control both relative and absolute task coordinates, this can be accomplished by solving the equation

$$\dot{\boldsymbol{\theta}} = \mathbf{J}_e^{+}(\mathbf{v}_{e,d} + \mathbf{K}_e\mathbf{e}_e) + (\mathbf{I} - \mathbf{J}_e^{+}\mathbf{J}_e)\dot{\boldsymbol{\theta}}_0, \quad (18)$$

where extended Jacobian is defined as

$$\mathbf{J}_e = \begin{bmatrix} \mathbf{J}_r \\ \mathbf{J}_a \end{bmatrix} \quad (19)$$

and $\mathbf{v}_{e,d}$ is the desired extended task space velocity composed of the desired relative and absolute task velocities. The $\mathbf{e}_e \in \mathbb{R}^6$ is the error between the desired and actual absolute task coordinates, calculated similar as in (16). Diagonal matrix $\mathbf{K}_e \in \mathbb{R}^{12\times12}$ contains suitably chosen positive kinematic control gains. Note that the dimension of the extended task defined with (19) can be ≤ 12, which allows to exploit the additional degrees of redundancy for secondary task(s).

III. Bimanual Task Adaptation

Assembly tasks performed by humans are usually accomplished with both hands. Humans are very good at performing bimanual assembly tasks that require compliance and force control and can quickly adapt to specific tools and environments. Therefore, we use human demonstration of the bimanual task as a starting point. Initial task is obtained with learning by demonstration (LbD) exploiting kinesthetic guidance. Unlike standard LbD approaches [18], we capture besides trajectories also forces and torques arising during the task execution as training data [19]. During the execution of learned skills, additional adaptation is often needed to cope with small differences induced by the environment, the inaccuracies in geometry of the manipulated objects, grasping tolerances, etc. It has been previously proved that on-line control alone can not in general provide successful adaptation in assembly tasks [20]. Therefore, various learning techniques were applied [21]. In our previous works we demonstrated that Iterative Learning Control (ILC) framework can efficiently adapt the previously demonstrated task to a new situation [9], [22], [19]. Therefore, ILC framework was applied also for bimanual task adaptation.

To learn the optimal control input, which is in our case the reference trajectory composed of position part vector $\mathbf{p}$ and rotation part quaternion $\mathbf{q}$, we applied ILC in the form

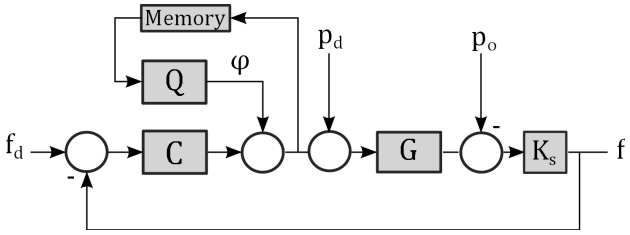


Fig. 2. Block diagram of force based adaptation scheme.

$$\mathbf{p}_l(k) = \mathbf{p}_d(k) + \boldsymbol{\varphi}_{p,l}(k) + \mathbf{C}_p(\gamma)\mathbf{e}_{f,l}(k) \quad (20)$$

$$\mathbf{q}_l(k) = \exp(\mathbf{C}_q(\gamma)\mathbf{e}_{m,l}(k)) * \exp(\boldsymbol{\varphi}_{q,l}(k)) * \mathbf{q}_d(k), \quad (21)$$

where l denotes the learning cycle, k is the time sample index, $\mathbf{C}_p(\gamma) \in \mathbb{R}^{3\times3}$ and $\mathbf{C}_q(\gamma) \in \mathbb{R}^{3\times3}$ are diagonal matrices composed of transfer function polynomials, γ is the backward shift operator, which delays a signal for one time sample and $\mathbf{e}_{f,l}(k)$ and $\mathbf{e}_{m,l}(k)$ are the force and torque tracking errors, respectively. Terms $\mathbf{p}_d(k)$ and $\mathbf{q}_d(k)$ denote the initially demonstrated trajectory in the form of the desired positions vector and rotations quaternion. Term $\boldsymbol{\varphi}_{p,l}(k)$ and $\boldsymbol{\varphi}_{q,l}(k)$ denotes position and rotation displacements, which will be learned by means of ILC. Remaining terms $\mathbf{C}_p(\gamma)\mathbf{e}_{f,l}(k)$ and $\mathbf{C}_q(\gamma)\mathbf{e}_{m,l}(k)$ belong to the admittance force controller, which minimizes force and torque tracking errors in the current iteration cycle. Therefore, the ILC scheme given by (20) and (21) is often refereed as "current iteration" ILC. Force and torque tracking errors are defined as $\mathbf{e}_{f,l}(k) = \mathbf{f}_d(k) - \mathbf{f}_l(k)$ and $\mathbf{e}_{m,l}(k) = \mathbf{m}_d(k) - \mathbf{m}_l(k)$, where $\mathbf{f}_d(k)$ and $m_d(k)$ are initially demonstrated forces and torques profiles, respectively. $\exp$ denotes the exponential map $\exp : \mathbb{R}^3 \mapsto \mathbb{S}$, defined as

$$\exp(\mathbf{r}) = \begin{cases} \cos(\|\mathbf{r}\|) + \sin(\|\mathbf{r}\|)\dfrac{\mathbf{r}}{\|\mathbf{r}\|}, & \mathbf{r} \neq 0 \\ 0, & \text{otherwise} \end{cases} \quad (22)$$

The compensation terms are learned with

$$\boldsymbol{\varphi}_{p,l}(k) = \mathbf{Q}_p(\gamma)(\boldsymbol{\varphi}_{p,l-1}(k) + \mathbf{C}_p(\gamma)\mathbf{e}_{f,l-1}(k)), \quad (23)$$

$$\boldsymbol{\varphi}_{q,l}(k) = \mathbf{Q}_q(\gamma)(\boldsymbol{\varphi}_{q,l-1}(k) + \mathbf{C}_q(\gamma)\mathbf{e}_{m,l-1}(k)), \quad (24)$$

where $\mathbf{Q}_p(\gamma) \in \mathbb{R}^{3\times3}$ and $\mathbf{Q}_q(\gamma) \in \mathbb{R}^{3\times3}$ are ILC transfer functions. Initial values of $\boldsymbol{\varphi}_{p,0}(k)$ and $\boldsymbol{\varphi}_{q,0}(k)$ are set to $0\ \forall\ k$. The same adaptation algorithm can be used for adaptation of both relative and absolute coordinates. In the above formulation we omitted indexes r and a, which define weather relative or absolute coordinates are subject of adaptation.

The stability of the proposed admittance force control law which iteratively updates the position compensation term will be analysed in the frequency domain of a time discrete system. For sake of simplicity, stability will be shown only for adaptation of the positional part of the trajectory. The overall ILC scheme for positional trajectories is outlined in

Fig. 2. Let uppercase letters denote one sided Z transform of the corresponding time-discrete signal denoted with low letter. Again, for the sake of simplicity, we will omit explicit dependence on z in transfer functions and Z transform of the signals. We define as a system output the force at the TCP of the robot, which is modelled assuming known environment stiffness $\mathbf{K}_s$,

$$\mathbf{F}_l = \mathbf{K}_s(\mathbf{G}\mathbf{P}_l - \mathbf{P}_o), \tag{25}$$

where $\mathbf{K}_s \in \mathbb{R}^{3\times3}$ is a diagonal positive definite environment stiffness matrix, $\mathbf{G} \in \mathbb{R}^{3\times3}$ is diagonal matrix containing transfer functions which map the desired position vector $\mathbf{P}_l$ into the actual position and $\mathbf{P}_o$ denotes the environment contact positions. In stability analyses we will consider the case where the robot dynamics is previously decoupled and linearized within the robot controller [23]. Therefore, $\mathbf{G}$ can be modelled as a diagonal matrix with transfer functions of second order. According to (20) and (23), Z transform of the error function $\mathbf{E}_l$, position update function $\mathbf{P}_l$ and learned offset function $\mathbf{\Phi}$ are

$$\mathbf{E}_l = \mathbf{F}_d - \mathbf{F}_l, \tag{26}$$

$$\mathbf{P}_l = \mathbf{P}_d + \mathbf{\Phi}_l + \mathbf{C}\mathbf{E}_l, \tag{27}$$

$$\mathbf{\Phi}_l = \mathbf{Q}(\mathbf{\Phi}_{l-1} + \mathbf{C}\mathbf{E}_{l-1}). \tag{28}$$

Now, let express the the error $\mathbf{E}_l$ as a function of the error in the previous learning cycle $\mathbf{E}_{l-1}$,

$$\begin{aligned}
\mathbf{E}_l &= \mathbf{F}_d - \mathbf{F}_l \\
&= \mathbf{F}_d - \mathbf{K}_s(\mathbf{G}\mathbf{P}_l - \mathbf{P}_o) \\
&= \mathbf{F}_d - \mathbf{K}_s(\mathbf{G}(\mathbf{P}_d + \mathbf{\Phi}_l + \mathbf{C}\mathbf{E}_l) - \mathbf{P}_o) \\
&= \mathbf{Q}(\mathbf{F}_d - \mathbf{K}_s(\mathbf{G}(\mathbf{P}_d + \mathbf{\Phi}_{l-1} + \mathbf{C}\mathbf{E}_{l-1}) - \mathbf{P}_o)) - \\
&\quad \mathbf{K}_s\mathbf{G}\mathbf{C}\mathbf{E}_l + (\mathbf{I} - \mathbf{Q})(\mathbf{F}_d - \mathbf{K}_s(\mathbf{G}\mathbf{P}_d - \mathbf{P}_o)) \\
&= \mathbf{Q}(\mathbf{F}_d - \mathbf{F}_{l-1}) - \mathbf{K}_s\mathbf{G}\mathbf{C}\mathbf{E}_l + \\
&\quad (\mathbf{I} - \mathbf{Q})(\mathbf{F}_d - \mathbf{K}_s(\mathbf{G}\mathbf{P}_d - \mathbf{P}_o)) \\
&= \mathbf{Q}\mathbf{E}_{l\text{-}1} - \mathbf{K}_s\mathbf{G}\mathbf{C}\mathbf{E}_l + (\mathbf{I} - \mathbf{Q})(\mathbf{F}_d - \mathbf{K}_s(\mathbf{G}\mathbf{P}_d - \mathbf{P}_o)).
\end{aligned} \tag{29}$$

In the above equation we added and subtracted the term $\mathbf{Q}(\mathbf{F}_d - \mathbf{K}_s(\mathbf{G}\mathbf{P}_d - \mathbf{P}_o))$ and used (25) – (28). Rearranging (29) we obtain

$$\frac{\mathbf{E}_l}{\mathbf{E}_{l-1}} = \frac{\mathbf{Q}}{I + \mathbf{K}_s\mathbf{G}\mathbf{C}} + \frac{\mathbf{I} - \mathbf{Q}}{I + \mathbf{K}_s\mathbf{G}\mathbf{C}} \frac{\mathbf{F}_d - \mathbf{K}_s(\mathbf{G}\mathbf{P}_d - \mathbf{P}_o)}{\mathbf{E}_{l-1}} \tag{30}$$

Asymptotic stability is assured iff $\frac{\mathbf{E}_l}{\mathbf{E}_{l-1}} < 1 \ \forall l$. Inserting again the z dependence into transfer functions and signals and substituting $z = e^{j\omega}$ in (30), the condition for asymptotic stability becomes [24]

$$\frac{\mathbf{Q}(e^{j\omega})}{\mathbf{I} + \mathbf{K}_s\mathbf{G}(e^{j\omega})\mathbf{C}(e^{j\omega})} + \frac{\mathbf{I} - \mathbf{Q}(e^{j\omega})}{\mathbf{I} + \mathbf{K}_s\mathbf{G}(e^{j\omega})\mathbf{C}(e^{j\omega})}\epsilon < 1, \forall\ \omega, \tag{31}$$

where

$$\epsilon = \frac{\mathbf{F}_d(e^{j\omega}) - \mathbf{K}_s(\mathbf{G}(e^{j\omega})\mathbf{P}_d(e^{j\omega}) - \mathbf{P}_o(e^{j\omega}))}{\mathbf{E}_{l-1}(e^{j\omega})} \tag{32}$$

and $\omega = [-\pi, \pi]$ is the frequency normalised with the sampling time of our time-discrete system.

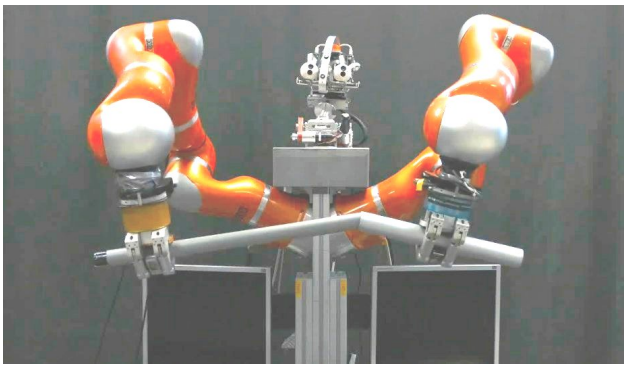


Fig. 3. Experimental platform.

Given the known transfer function $\mathbf{G}(z)$ and estimated environment stiffness $\mathbf{K}_s$ we have to design such admittance control law transfer function $\mathbf{C}(z)$ and learning function $\mathbf{Q}(z)$, that the learning error $\mathbf{E}(z)$ asymptotically decays to 0 when $l \to \infty$, Note that the nominator of the term ϵ (32) is 0, since $\mathbf{K}_s(\mathbf{G}\mathbf{P}_d - \mathbf{P}_o) = \mathbf{F}_d$, assuming ideal model for $\mathbf{K}_s$, $\mathbf{G}$ and $\mathbf{P}_o$. In practice, this is never true and the denominator of ϵ is small and bounded value, which depends only on the desired force $\mathbf{F}_d$, desired trajectory $\mathbf{P}_d$ and environment $\mathbf{P}_o$. Therefore, ϵ increases when the error $\mathbf{E}$ decreases. Consequently, zero learning error can be guaranteed only with the choice $\mathbf{Q}(z) = \mathbf{I}$. On the other hand, it is generally very hard to fulfill condition (31) with $\mathbf{Q}(z) = \mathbf{I}$. In most cases $\mathbf{Q}$ in the form of a low pass filter will assure the stability, but increase learning error [25]. Therefore, the design of $\mathbf{Q}(z)$ is a tradeoff between the robustness and stability and performance of the learning algorithm. The design of the learning algorithm can be thus summarised in the following steps:

1) calculate the upper bound of $\|\epsilon\|$ upon the admissible error $\mathbf{E}(e^{j\omega})$,
2) check if (31) is fulfilled, by e.g. Bode or Nyquist plot,
3) tune the parameters of the transfer function $\mathbf{C}$, $\mathbf{L}$ and $\mathbf{Q}$ until (31) is fulfilled.

IV. Experimental evaluation

We evaluated the performance of the proposed learning on bimanual peg in a hole tasks, where the robot has to insert one round pole into another, as show in Fig. 3. The outer diameter of one pole tightly fitted the inner diameter of the other pole. The experimental platform was composed of two KUKA LWR robot arms equipped with three finger Barret hands. The two arms were controlled with an external PC computer via FRI interface at sampling rate of 500 Hz. The Cartesian compliance of both robot arms was set to 1000 N/m for positions and to 300 Nm/rd for rotations. For this experiment, we have applied P type admittance control law with equal gains for $\mathbf{C}_p = 0.0002\ \mathbf{I}$. The KUKA built in controller decouples and linearizes the LWR robot dynamics, which can be approximated with discrete transfer function $\mathbf{G} = \frac{0.011z+0.01}{z^2-1.7z+0.7289}$. For the learning, we have chosen matrix $\mathbf{Q}$ as a diagonal matrices containing 2nd order low

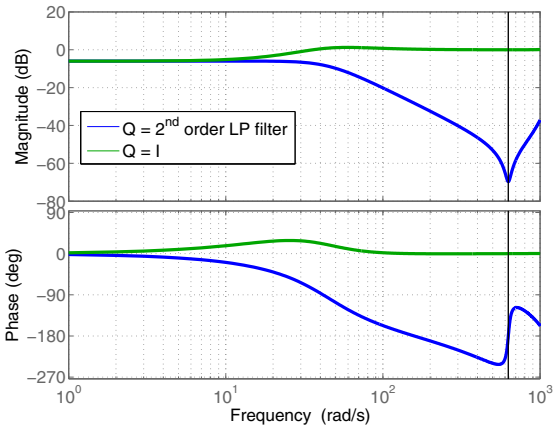


Fig. 4. Bode plot of the learning transfer functions for two choices of $\mathbf{Q}$.

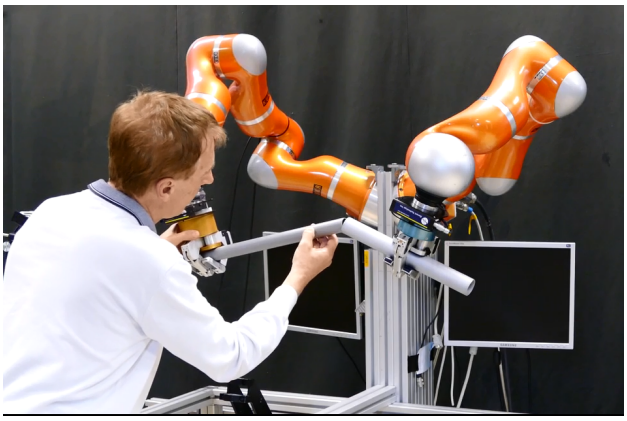


Fig. 5. Demonstration of the relative task with kinesthetic guiding of the right robot. The left robot does not move during the demonstration.

pass filter with zero at -0.9 and double poles at 0.85 in the Z plane. First, we verified the stability of the learning algorithm by drawing the Bode plot for the transfer function $\frac{\mathbf{Q}(z)}{\mathbf{I}+\mathbf{K}_s\mathbf{G}(z)\mathbf{C}(z)}$, where $\mathbf{K}_s$ was set to 500 N/m, which is exactly the mutual programmed stiffness of both robot arms. Fig. 4 shows Bode plot for two cases: in the first case $\mathbf{Q}$ is the 2nd order filter, and in the second case $\mathbf{Q}$ is unity matrix. According to (31), magnitude of the Bode plot should be below $(1-\epsilon)$ for all frequencies. It can be seen, that the Bode magnitude crosses the stability margin at high frequencies (above $20s^{-1}$) for $\mathbf{Q} = \mathbf{I}$, while it stays safe within the stability region when $\mathbf{Q}$ is in the form of a low pass filter. Note also that the magnitude plot uses logarithmic scale (dB) . The same result was verified also by the simulation, where scheme without the filtering starts to oscillate after it diminishes the tracking error in few initial learning cycles. The explanation of this phenomena is that at the beginning of the learning, low frequencies are dominant, whereas the Bode plot shows that both schemes are stable. When the algorithm diminishes tracking error, high frequencies become dominant and we have to use filtering to assure the stability of the learning.

As explained previously, small tolerances in position trajectory can result in high contact forces during assembly tasks. Force controller tries to diminish these forces instantly, while the learning minimizes them gradually during learning iterations. Force control might become inefficient or even unstable at rapid changes of the measured force and torque errors. Therefore, it is good idea to slow down the execution of the demonstrated policy when large deviation between the desired and actual forces and torques occurs. This flexibility to modulate the execution speed of the trajectory is provided by DMPs using phase stopping technique [10]. Therefore, time dependent trajectories $\mathbf{p}_d(k)$ and $\mathbf{q}_d(k)$ are replaced with the phase dependent DMP trajectories $\mathbf{p}_{DMP}(x)$ and $\mathbf{q}_{DMP}(x)$, where x is the phase variable [10]. Similarly, also the time dependent learned term $\boldsymbol{\varphi}_p(k)$ and $\boldsymbol{\varphi}_q(k)$ has to be replaced with phase dependent signals $\boldsymbol{\varphi}_{p,RBF}(x)$ and $\boldsymbol{\varphi}_{q,RBF}(x)$. Subscript RBF denotes, that the corresponding signal was encoded with radial basis functions (RBF), as explained in [19]. Accordingly, also force and torque tracking errors has to be encoded with RBF.

With the real experiment, we had to provide initial trajectories and force profiles in relative coordinates. As only the relative coordinates matter for this kind of the task execution, we can demonstrate the task by keeping one of the robots fixed, while the other was used for demonstration. This was accomplished with LbD, where we manually guided one of the robot arms using kinesthetic guidance and captured resulting relative coordinates calculation of relative coordinates from both robot poses using (1) and (5). One instance of the trajectory demonstration is show in Fig. 5. TCP forces and torques were captured from joint torque measurement, provided by the KUKA LWR. Within this setup, forces and torques can not be captured together with the position/orientation trajectories applying the kinesthetic guidance [19]. Therefore, we had to rerun the captured position/orientation trajectory in order to obtain non-distorted forces and torques. After that, we displaced one long peg in Barret hands in the local y direction for 5 mm and run 4 adaptation cycles. The results are shown in Fig. 6. Dotted line in graphs shows the insertion phase. As we can observe from plots, we have non-zero forces and torques even before contact. Part of this artefact is due to imprecise dynamics calculation when estimating TCP forces and torques from the joint torques. Another part is due to non-neglige inertia of long poles. However, the proposed adaptation algorithm effectively diminishes the force and torque tracking errors in just few learning cycles. For that reason, also the adaptation scheme with $\mathbf{Q} = \mathbf{I}$ in practice performs almost equally good as the scheme with $\mathbf{Q}$ in the form of the 2nd order filter, providing that the adaptation mechanism is switched off after few adaptation cycles.

V. Conclusions

In the paper we proposed a new force adaptation scheme for bimanual systems. The main advantage of the proposed algorithms is that adaptation act directly in relative and

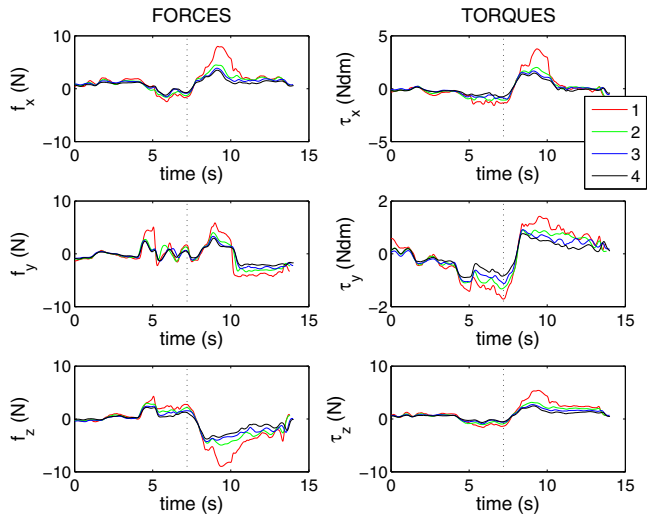


Fig. 6. Forces and torques during the long peg insertion tasks. Labels 1–4 show the adaptation cycle. We can observe extremely quick convergence of the ILC adaptation algorithm.

absolute coordinates. Subdivision of the motion of both arms to relative and absolute tasks is based on a modified definition of relative Jacobian, which assures also proper mapping to the Jacobian null-space, when additional tasks which exploit kinematic redundancy of the overall system are defined. The admittance based force adaptation is based on the ILC framework. The proposed controller belongs to a class of causal ILC controller, for which it was proved that the same steady state error can be obtained with standard feedback controller applying high gains [26]. However, high gain controllers are not suitable for robots interacting with humans [27], therefore we consider that ILC is still favourable. Stability of the proposed learning algorithm was discussed and proved together with practical notes how to choose the parameters of the learning algorithm and how to verify the stability. Theoretical results were verified with practical implementation of a dual arm peg-in-a-hole task. In our future work, we will extend the proposed algorithm to the impedance control law, which will modify control torques directly in contrast to the proposed algorithm, which adapts the position and orientation trajectory.

References

[1] Y. Zheng and J. Luh, "Joint torques for control of two coordinated moving robots," in *IEEE International Conference on Robotics and Automation. Proceedings.*, vol. 3, Apr 1986, pp. 1375–1380.

[2] T. Tarn, A. Bejczy, and X. Yun, "Coordinated control of two robot arms," in *IEEE International Conference on Robotics and Automation. Proceedings.*, vol. 3, Apr 1986, pp. 1193–1202.

[3] M. Uchiyama and P. Dauchez, "A symmetric hybrid position/force control scheme for the coordination of two robots," in *IEEE International Conference on Robotics and Automation (ICRA)*, vol. 1, Philadelphia, PA, 1988, pp. 350–356.

[4] P. Chiacchio and S. Chiaverini, *Kinematic control of dual-arm systems*. Springer, 1998.

[5] J. Kober, D. Bagnell, and J. Peters, "Reinforcement learning in robotics: A survey," *International Journal of Robotics Research*, no. 11, pp. 1238–1274, 2013.

[6] E. A. Theodorou, J. Buchli, and S. Schaal, "A generalized path integral control approach to reinforcement learning," *Journal of Machine Learning Research*, vol. 11, pp. 3137–3181, 2010.

[7] D. Bristow, M. Tharayil, and A. Alleyne, "A survey of iterative learning control," *IEEE Control Systems Magazine*, vol. 26, no. 3, pp. 96–114, 2006.

[8] M. Norrlöf, "An adaptive iterative learning control algorithm with experiments on an industrial robot," *IEEE Transactions on Robotics and Automation*, vol. 18, no. 2, pp. 245–251, Apr 2002.

[9] A. Gams, B. Nemec, A. Ijspeert, and A. Ude, "Coupling movement primitives: Interaction with the environment and bimanual tasks," *IEEE Transactions on Robotics*, vol. 30, no. 4, pp. 816–830, 2014.

[10] A. J. Ijspeert, J. Nakanishi, H. Hoffmann, P. Pastor, and S. Schaal, "Dynamical movement primitives: Learning attractor models for motor behaviors," *Neural Computation*, vol. 25, no. 2, pp. 328–373, 2013.

[11] J. F. Broenink and M. L. J. Tiernego, "Peg-in-Hole assembly using impedance control with a 6 DOF robot," in *Proceedings of the 8th European Simulation Symposium*, 1996, pp. 504–508.

[12] K. Hirana, T. Suzuki, and S. Okuma, "Optimal motion planning for assembly skill based on mixed logical dynamical system," in *7th International Workshop on Advanced Motion Control*, Maribor, Slovenia, 2002, pp. 359–364.

[13] Y. Li, "Hybrid control approach to the peg-in-hole problem," *IEEE Robotics and Automation Magazine*, vol. 4, no. 2, pp. 52–60, 1997.

[14] A. Stemmer, A. Albu-Schäffer, and G. Hirzinger, "An analytical method for the planning of robust assembly tasks of complex shaped planar parts," in *IEEE International Conference on Robotics and Automation (ICRA)*, Rome, Italy, 2007, pp. 317–323.

[15] T. Yamashita, I. Godler, Y. Takahashi, K. Wada, and R. Katoh, "Peg-and-hole task by robot with force sensor: Simulation and experiment," in *International Conference on Industrial Electronics, Control and Instrumentation (IECON)*, Kobe, Japan, 1991, pp. 980–985.

[16] F. Caccavale, P. Chiacchio, and S. Chiaverini, "Task-space regulation of cooperative manipulators," *Automatica*, vol. 36, pp. 879–887, 2000.

[17] B. Adorno, P. Fraisse, and S. Druon, "Dual position control strategies using the cooperative dual task-space framework," in *IEEE/RSJ International Conference on Intelligent Robots and Systems (IROS)*, Taipei, Taiwan, 2010, pp. 3955–3960.

[18] R. Dillmann, "Teaching and learning of robot tasks via observation of human performance," *Robotics and Autonomous Systems*, vol. 47, no. 2-3, pp. 109–116, 2004.

[19] F. Abu-Dakka, B. Nemec, J. Jørgensen, T. Savarimuthu, N. Krüger, and A. Ude, "Adaptation of manipulation skills in physical contact with the environment to reference force profiles," *Autonomous Robots*, vol. 39, no. 2, pp. 199–217, 2015.

[20] P. Pastor, L. Righetti, M. Kalakrishnan, and S. Schaal, "Online movement adaptation based on previous sensor experiences," in *IEEE/RSJ International Conference on Intelligent Robots and Systems (IROS)*, Sept 2011, pp. 365–371.

[21] L. Righetti, M. Kalakrishnan, P. Pastor, J. Binney, J. Kelly, R. Voorhies, G. Sukhatme, and S. Schaal, "An autonomous manipulation system based on force control and optimization," *Autonomous Robots*, vol. 36, no. 1-2, pp. 11–30, 2014.

[22] B. Nemec, F. J. Abu-Dakka, B. Ridge, J. A. Jørgensen, T. R. Savarimuthu, J. Jouffroy, H. G. Petersen, N. Krüger, and A. Ude, "Transfer of assembly operations to new workpiece poses by adaptation to the desired force profile," in *16th International Conference on Advanced Robotics (ICAR)*, Montevideo, Uruguay, 2013.

[23] J. Nakanishi, R. Cory, M. Mistry, J. Peters, and S. Schaal, "Operational space control: A theoretical and empirical comparison," *The International Journal of Robotics Research*, vol. 27, pp. 737–757, 2008.

[24] K. L. Moore, Y. Chen, and H.-S. Ahn, "Iterative learning control: A tutorial and big picture view," in *45th IEEE Conference on Decision and Control*, dec. 2006, pp. 2352 –2357.

[25] M. Norrlöf and S. Gunnarsson, "Experimental comparison of some classical iterative learning control algorithms," *IEEE Transactions on Robotics and Automation*, vol. 18, no. 4, pp. 636–641, 2002.

[26] P. B. Goldsmith, "Brief on the equivalence of causal lti iterative learning control and feedback control," *Automatica*, vol. 38, no. 4, pp. 703–708, Apr. 2002.

[27] E. Gribovskaya, A. Kheddar, and A. Billard, "Motion learning and adaptive impedance for robot control during physical interaction with humans," in *2011 IEEE International Conference on Robotics and Automation (ICRA)*, May 2011, pp. 4326–4332.

2015 IEEE-RAS 15th International Conference on Humanoid Robots (Humanoids)
November 3 - 5, 2015, Seoul, Korea

Multi-Purpose Natural Language Understanding Linked to Sensorimotor Experience in Humanoid Robots

Ekaterina Ovchinnikova, Mirko Wächter, Valerij Wittenbeck, Tamim Asfour[1]

***Abstract*— Humans have an amazing ability to bootstrap new knowledge. The concept of structural bootstrapping refers to mechanisms relying on prior knowledge, sensorimotor experience, and inference that can be implemented in robotic systems and employed to speed up learning and problem solving in new environments. In this context, the interplay between the symbolic encoding of the sensorimotor information, prior knowledge, planning, and natural language understanding plays a significant role. In this paper, we show how the symbolic descriptions of the world can be generated on the fly from the continuous robot's memory. We also introduce a multi-purpose natural language understanding framework that processes human spoken utterances and generates planner goals as well as symbolic descriptions of the world and human actions. Both components were tested on the humanoid robot ARMAR-III in a scenario requiring planning and plan recognition based on human-robot communication.**

I. INTRODUCTION

Significant research efforts in humanoid robotics have been focused on mimicking human cognition. This especially concerns the autonomous acquisition of knowledge and application of this knowledge in previously unseen situations. The concept of *structural bootstrapping* was introduced in the context of the *Xperience* project [1]. It addresses mechanisms relying on prior knowledge, sensorimotor experience, and inference that can be implemented in robotic systems and employed to speed up learning and problem solving in new environments. Earlier experiments demonstrate how structural bootstrapping can be applied at different levels of a robotic architecture including a sensorimotor level, a symbol-to-signal mediator level, and a planning level [2], [3].

In the context of structural bootstrapping, the interplay between the symbolic encoding of the sensorimotor information, prior knowledge, planning, and natural language understanding plays a significant role. Available robot skills and the world state have to be represented in a symbolic form for a planner to be able to operate with it. In the previous experiments, the symbolic representations of the objects in the world and their initial locations were created manually and hard-coded in the scenario settings, cf. [3]. In this paper, we show how the comprehensive descriptions of the world (domain descriptions) can be generated on the fly from the continuous robot's memory. Sensor data are mapped to symbolic representations required for linking the sensorimotor experience of the robot to language understanding and planning.

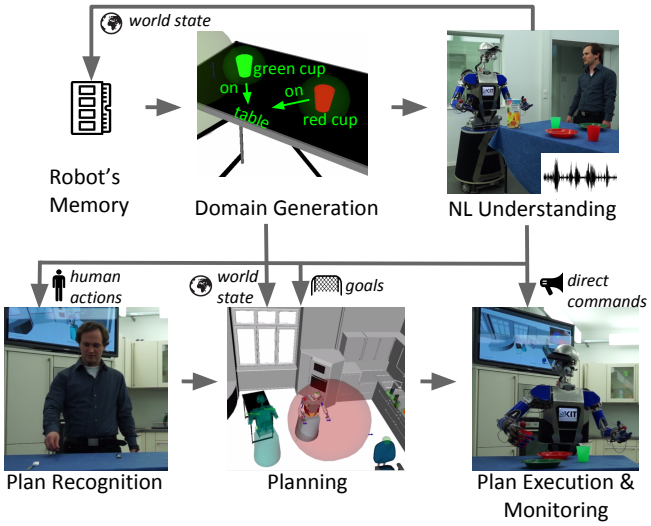


Fig. 1. System architecture.

We also introduce a natural language understanding (NLU) framework that allows us to generate goals for the planner as well as symbolic descriptions of the world and human actions given human spoken utterances. The framework is intended for a flexible multi-purpose human-robot communication. For example, if a robot's plan execution is failing because a required object is missing, we want to be able to communicate the location of this or an alternative object through natural language. In addition to the vision-based action recognition, we want to be able to comment on human actions using speech.

Natural language (NL) provides an effective tool for untrained users to interact with robots in an intuitive way, which is especially important for robots intended to perform collaborative tasks with people. One of the major challenges in application of NLU to robotics concerns grounding ambiguous NL constructions into actions, states, relations, and objects known to the robot. For example, the commands *Bring the milk from the fridge*, *Bring the milk. It's in the fridge*, *Take the milk out of the fridge* all imply that the milk is located in the fridge. Similarly, embedding the sensorimotor experience of the robot is crucial for understanding NL utterances. For example, if the robot is holding a cup, then it should interpret the command *Put the cup down* as probably referring to the cup it is holding rather

*The research leading to these results has received funding from the European Union Seventh Framework Programme under grant agreement Nº 270273 (Xperience).

[1]The authors are with the Institute for Anthropomatics and Robotics, High Performance Humanoid Technologies Lab (H^2T), at the Karlsruhe Institute of Technology (KIT), Karlsruhe, Germany. {ovchinnikova,waechter,asfour}@kit.edu

978-1-4799-6885-5/15/$31.00 ©2015 IEEE

than any other cup. Another important issue concerns the functionality of NL. Most of the literature on NLU for robotics focuses on instruction interpretation. At the same time, NL in human-robot interaction can also be used for describing the world, commenting on human actions, giving feedback, etc. These types of communication are especially important when performing collaborative tasks and in the situations when the robot cannot access the world state by using sensors. The proposed framework treats multiple types of NL input (commands, descriptions of the world and human actions) and interacts with related components such as the robot's memory, a planner, and a plan recognizer. It performs grounding of the symbolic representations into the sensorimotor experience of the robot and supports complex linguistic phenomena, such as ambiguity, negation, anaphora, and quantification without requiring training data.

We test the domain description generation and NLU on the humanoid robot ARMAR-III [4] in a scenario requiring planning and plan recognition based on human-robot communication.

The paper is structured as follows. After presenting the general system architecture in Sec. II, we describe the domain description generation from the robot's memory (Sec. III). Sec. IV introduces the natural language understanding pipeline. Sec. V briefly presents the planner and the plan recognizer employed in this study. Sec. VI discusses how plan execution and monitoring are organized in our framework. Experiments on the humanoid robot ARMAR-III are presented in Sec. VII. Related work is discussed in Sec. VIII. Section IX concludes the paper.

II. SYSTEM ARCHITECTURE

Fig. 1 shows the system architecture realized within the robot development environment *ArmarX* [5]. The system consists of six major building blocks: robot's memory, domain generation, NL understanding, plan recognition, planning, as well as plan execution and monitoring.

Domain descriptions are generated from the robot's memory (Sec. III). The robot's memory is represented within *MemoryX*, one of the main components of *ArmarX*. The domain description is used by the NL understanding component for grounding and generating the domain knowledge base (Sec. IV) as well as by the planner (Sec. V). The developed multi-purpose NLU framework can distinguish between a) direct commands that can be executed without planning (*Move to the table*), b) plan requiring commands that are converted into planner goals, which are processed by a planner (*Set the table*), c) descriptions of human actions that are used by a plan recognizer that recognizes human plans and generates corresponding robot goals further processed by the planner (*I'm grasping the knife*), d) descriptions of the world that are added to the robot's memory and used by both the plan recognizer and the planner (*The cup is on the table*), see Sec. IV. Plan execution is performed by the plan execution and monitoring component, which also verifies if the plan is executed correctly (Sec. VI). Each time an NL utterance is registered and processed by the NLU pipeline, the planner, and the plan recognizer, the robot's memory is updated and the required actions are added to the task stack to be processed by the plan execution component. Human comments can thus be used to update the world state in the robot's memory before or during action execution and are considered by the robot to adjust its plan accordingly.

III. GENERATION OF DOMAIN DESCRIPTIONS FROM ROBOT'S MEMORY

The challenge of mapping sensor data to symbolic representations lies in the diversity of each specific mappings, i.e. each symbol depends on a different combination of sensor data. We approach this challenge by designing the mapping procedure in a modular way. First, sensorimotor experience is processed and turned into continuous sub-symbolic representations (e.g., coordinates of objects, the robot, and robot's hands) that are added to the robot's memory. These continuous representations are mapped to object and location names or predicates. Finally, representations describing the world state are generated.

A. Memory Structure

The robot development environment employed in the described study contains a biologically inspired framework for storing and representing robot's knowledge [5]. In the described framework, the memory architecture *MemoryX* consists of the *prior knowledge*, the *long-term memory*, and the *working memory* that provide symbolic entities like actions, objects, states, and locations. The basic elements of the memory called *memory entities* are represented by name-value maps. The prior knowledge contains persistent data inserted by the developer that the robot could not learn by itself like accurate object 3D models [6]. The long-term memory consists of knowledge that has been stored persistently, e.g., common object locations that are learned from the robot's experience during task execution and persistently stored as heat maps [7]. The working memory contains volatile knowledge about the current world state, e.g., object existence and position or relations between entities. The working memory is updated by external components like the robot self-localization, object localization, or natural language understanding, whenever they receive new information. To account for uncertainties in sensor-data, each memory entity value is accompanied by a probability distribution. In case of object locations, new data is fused with the data stored in the memory using a Kalman-filter.

B. Mapping sensorimotor data to symbols

In this work, self-localization, visual object recognition, and kinematics of the robot where used for mapping sensorimotor experience to symbols. For the self-localization, we use laser-scanners and a representation of the world as a 2D map. The self-localization is used to navigate on a labelled 2D graph, in which location labels are associated with center coordinates and a radius. For visual object recognition, we use RGB stereo vision with texture-based [8] or color-based [6] algorithms.

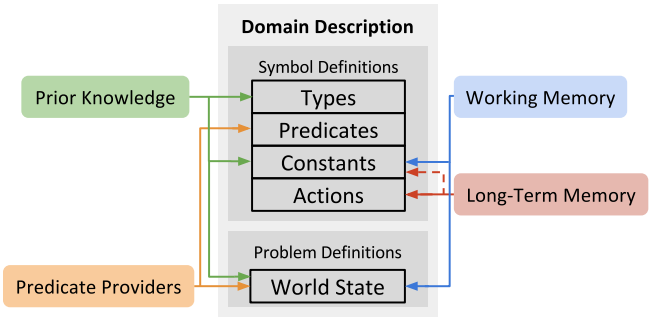


Fig. 2. Components involved in the domain description generation.

The conversion of continuous sensor data into discrete symbolic data is done by *predicate providers*. Each world state predicate is defined in its own predicate provider component, which outputs a predicate state (unknown, true, or false) by evaluating the content of the working memory or low-level sensorimotor data. In the experiment described below, we use the following predicate providers: `grasped` represents an object being held by an agent using a hand; `objectAt` and `agentAt` represent object and robot locations, correspondingly; `leftgraspable` and `rightgraspable` represent the fact that an object at a certain location can be grasped by the corresponding hand of the robot. Predicate providers can access other components (e.g., the working memory, robot kinematics, long-term memory) to assess the predicate state. For example, the `objectAt` predicate provider uses the distance between the detected object coordinates and the center coordinates of the location label.

Only those objects that are required for fulfilling a particular task are recognized during action execution. High-level components operating on a symbolic level generate requests for a particular object to be recognized at a particular locations. Other objects are not tracked to avoid false positive object recognition.

C. Domain Description Generation

The information contained in the robot's memory is used to generate a symbolic domain description consisting of static symbol definitions and problem specific definitions, see Fig. 2. The symbol definitions consist of types, constants, predicate definitions, and action descriptions, while the problem definitions consist of the symbolic representation of the current world state defined by predicates. Types enumerate available agents, hands, locations, and object classes contained in the prior knowledge. Constants represent actual instances, on which actions can be performed, and are therefore generated by using entities in the working memory. Each constant can have multiple types, such that one is the actual class of the corresponding entity, and others are parents of that particular class including transitive parentship. For example, instances of the type `cup` are also instances of `graspable` and `object`.

For some objects, the robot might not know yet where they are located, but their locations need to be defined for the planner to plan actions on them. In such cases, the domain generator uses the long-term memory to make assumptions about possible object locations. The domain generator derives action representations from the long-term memory, where they are associated with specific robot skills represented by statecharts [5].

The generated domain description is used by the NLU component as well as by the planning component. The NLU component uses domain descriptions to create a knowledge base and to ground NL references. The planning component uses it as the knowledge base for finding plans.

IV. MULTI-PURPOSE NL UNDERSTANDING

We intend to a) ground NL utterances to actions, objects, and locations stored in the robot's memory, b) distinguish between commands, descriptions of the world, and descriptions of human actions, c) generate representations of each type of the NL input suitable for the downstream components (planner, plan recognizer, action execution component). Our approach is based on the abductive inference, which can be used for interpreting NL utterances as observations by linking them to known or assumed facts, cf. [9]. The NL understanding pipeline shown in Fig. 3 consists of the following processing modules. The text produced by a speech recognition component[1] is processed by a semantic parser that outputs a logical representation of the text. This representation together with observations stored in the robot's memory and the lexical and domain knowledge base constitute an input for an abductive reasoning engine that produces a mapping to the domain. The mapping is further classified and post-processed. The pipeline is flexible in the sense that each component can be replaced by an alternative. We use the implementation of the abduction-based NLU that was developed in the context of knowledge-intensive large-scale text interpretation [11].

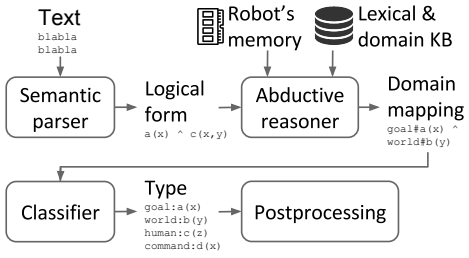


Fig. 3. Natural language understanding pipeline.

A. Logical form

We use logical representations of NL utterances as described in [12]. In this framework, a *logical form* (LF) is a conjunction of propositions and variable inequalities, which have argument links showing relationships among phrase constituents. For example, the following LF corresponds to the command *Bring me the cup from the table*:

[1]In the experiments described in this paper, we used the speech recognition system presented in [10].

$\exists e_1, x_1, x_2, x_3, x_4$ (*bring-v*$(e_1, x_1, x_2, x_3) \wedge$ *thing*$(x_1) \wedge$ *person*$(x_2) \wedge$ *cup-n*$(x_3) \wedge$ *table-n*$(x_4) \wedge$ *from-p*(x_3, x_4)),

where variables x_i refer to the entities *thing*, *person*, *cup*, and *table*, whereas variable e_1 refers to the eventuality of x_1 bringing x_2 to x_3; see [12] for more details. In the experiments described below, we used the *Boxer* parser [13]. Alternatively, any dependency parser can be used if it is accompanied by an LF converter as described in [14].

B. Abductive inference

Abduction is inference to the best explanation. Formally, logical abduction is defined as follows:

Given: Background knowledge B, observations O, where both B and O are sets of first-order logical formulas,
Find: A hypothesis H such that $H \cup B \models O, H \cup B \not\models \bot$, where H is a set of first-order logical formulas.

Abduction can be applied to discourse interpretation [9]. In terms of abduction, logical forms of the NL fragments represent observations, which need to be explained by the background knowledge. Where the reasoner is able to prove parts of the LF, it is anchoring it in what is already known from the overall context or from the background knowledge. Where assumptions are necessary, it is gaining new information. Suppose the command *Bring me the cup from the table* is turned into an observation o_c. If the robot's memory contains an observation of a particular instance of `cup` being located on the table, this observation will be concatenated with o_c and the noun phrase *the cup* will be grounded to this instance by the abductive reasoner.

We use a tractable implementation of abduction based on Integer Linear Programming (ILP) [11]. The reasoning system converts a problem of abduction into an ILP problem, and solves the problem by using efficient techniques developed by the ILP research community. Typically, there exist several hypotheses explaining an observation. In the experiments described below, we use the framework of *weighted abduction* [9] to rank hypotheses according to plausibility and select the best hypothesis. This framework allows us to define assumption costs and axiom weights that are used to estimate the overall cost of the hypotheses and rank them. As the result, the framework favors most economical (shortest) hypotheses as well as hypotheses that link parts of observations together and support discourse coherence, which is crucial for language understanding, cf. [15]. However, any other abductive framework and reasoning engine can be integrated into the pipeline.

C. Lexical and domain knowledge base

In our framework, the background knowledge B is a set of first-order logic formulas of the form

$$P_1^{w_1} \wedge ... \wedge P_n^{w_n} \rightarrow Q_1 \wedge ... \wedge Q_m,$$

where P_i, Q_j are predicate-argument structures or variable inequalities and w_i are axiom weights.[2]

[2]See [14] for a discussion of the weights.

Lexical knowledge used in the experiments described below was generated automatically from the lexical-semantic resources WordNet [16] and FrameNet [17]. First, verbs and nouns were mapped to the synonym classes. For example, the following axiom maps the verb *bring* to the class of giving:

action#give$(e_1, agent, recipient, theme) \rightarrow$
bring-v$(e_1, agent, theme) \wedge$ *to-p*$(e_1, recipient)$

Prepositional phrases were mapped to source, location, instrument, etc., predicates. Different syntactic realizations of each predicate for each verb (e.g., *from X*, *in X*, *out of X*) were derived from syntactic patterns specified in FrameNet that were linked to the corresponding FrameNet roles. See [18] for more details on the generation of lexical axioms. A simple spatial axiom was added to reason about locations, which states that if an object is located at a part of a location (*corner*, *top*, *side*, etc.), then it is located at the location.

The synonym classes were further manually axiomatized in terms of domain types, predicates, constants, and actions. For example, the axiom below is used to process constructions like *bring me X from Y*:

goal#inHandOf$(theme, Human) \wedge$
world#objectAt$(theme, loc) \rightarrow$
action#give$(e_1, Robot, recipient, theme) \wedge$
location#source(e_1, loc),

which represents the fact that the command evokes the goal of the given object being in the hand of the human and the indicated source is used to describe the location of the object in the world. The prefixes *goal#* and *world#* indicate the type of information conveyed by the corresponding linguistic structures. The framework can also handle numerals, negation, quantifiers represented by separate predicates in the axioms (e.g., *not*, *repeat*). The repetition, negation, and quantification predicates are further treated by the post-processing component. The hierarchy axioms (`red_cup`$\rightarrow$`cup`) and inconsistency axioms (`red_cup` *xor* `green_cup`) were generated automatically from the domain descriptions.

D. Object grounding

If objects are described uniquely, then they can be directly mapped to the constants in the domain. For example, *the red cup* in the utterance *Give me the red cup* can be mapped to the constant `red_cup` if there is only one red cup in the domain. However, redundant information that can be recovered from the context is often omitted in the NL communication, cf. [19]. In our approach, grounding of underspecified references is naturally performed by the abductive reasoner interpreting observations by linking their parts together, cf. Sec. IV-B. For example, given the text fragment *The red cup is on the table. Give it to me*, the pronoun *it* in the second sentence will be linked to *red cup* in the first sentence and grounded to `red_cup`. To link underspecified references to earlier object mentions in a robot-human interaction session, we keep all mentions and concatenate them with each new input LF to be interpreted. Predicates describing the world from the robot's memory

are also concatenated with LFs to enable grounding. Given *Bring me the cup from the table*, the reference *the cup from the table* will be grounded to an instance of `cup` observed as being located on an instance of `table`.

If some arguments of an action remain underspecified or not specified, then the first instance or the corresponding type will be derived from the domain description. For example, the execution of the action of putting things down requires a hand to be specified. In the NL commands this argument is often omitted (*Put the cup on the table*), because for humans it does not matter, which hand the robot will use. The structure `putdown(cup,table,hand)` is generated by the NLU pipeline for the first command above. The grounding function then selects the first available instance of the underspecified predicate. In future, we consider using a clarification dialogue, as proposed, for example, in [20].

E. Classifier

The classifier takes into account prefixes assigned to the inferred predicates. For example, the abductive reasoner returned the following mapping for the command *Bring me the cup from the table*:

$$\textit{action\#give}(e_1, x_1, x_2, x_3) \wedge \textit{location\#source}(e_1, x_4) \wedge x_1 = Robot \wedge x_2 = Human \wedge \textit{goal\#inHandOf}(x_3, Human) \wedge \textit{world\#objectAt}(x_3, x_4)$$

The classifier extracts predicates with prefixes and predicates related to the corresponding arguments. The following structures will be produced for the mapping above:

```
[goal: inHandOf(cup,Human),
world: objectAt(cup,table)]
```

Actions that do not evoke goals or world descriptions are interpreted by the classifier as direct commands or human action descriptions depending on the agent. For example, *action#grasp*(*Human*,*cup*) (*I'm grasping the cup*) will be interpreted as a human action description, while *action#grasp*(*Robot*,*cup*) (*Grasp the cup*) is a direct command.

The classifier can also handle nested predicates. For example, the utterances 1) *Help me to move the table*, 2) *I will help you to move the table*, 3) *I will help you by moving the table* will be assigned the following structures, correspondingly:

1) `[direct_command: helpRequest:[requester: Human, action: move(Robot,table)]`
2) `[human_action: help:[helpInAction: move(Robot,table)]`
3) `[human_action: help:[helpByAction: move(Human,table)]`

F. Post-processing

The post-processing component converts the extracted data into the format required by the downstream modules. The direct commands are immediately processed by the Plan Execution Monitor. Goals extracted from utterances are converted into a planner goal format, so that *not* predicate is turned into the corresponding negation symbol, predicates that need multiplication (indicated by the *repeat* predicate) are multiplied, and quantification predicates are turned into

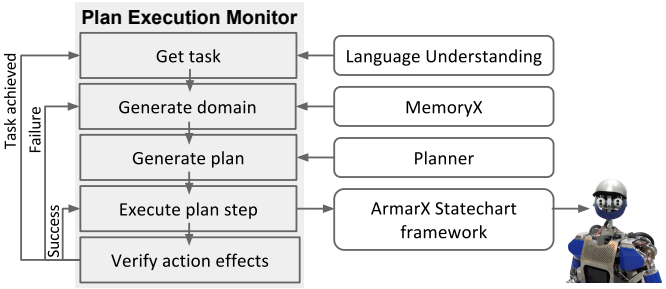


Fig. 4. Plan generation, execution, and monitoring.

quantifiers. For example, the commands 1) *Put two cups on the table* and 2) *Put all cups on the table* can be converted into the following goal representations in the PKS syntax [21], correspondingly:

1) `(existsK(?x1: cup, ?x2: table) K(objectAt(?x1,?x2)) & (existsK(?x3: cup) K(objectAt(?x3,?x2)) & K(?x1 != ?x3)))`
2) `(forallK(?x1: cup) (existsK(?x2: table) K(objectAt(?x1,?x2))))`

Similarly, human action descriptions are converted into the format required by the plan recognizer.

V. PLANNING AND PLAN RECOGNITION

We define a plan as a sequence of actions $P = \langle a_1, .., a_n \rangle$ with respect to the initial state s_0 and the goal G such that $\langle s_0, P \rangle \models G$. In the experiments described in Sec. VII, we used the *PKS* planner [21], which takes a domain description (Sec. III) and a goal (Sec. IV) represented in the *PKS* syntax as input and returns sequences of grounded actions with their pre- and post-conditions.

In the described experiments, we employed the probabilistic plan recognizer *ELEXIR* [22] that takes grounded human actions and domain descriptions in the *ELEXIR* syntax as input and computes the conditional probability of a particular human goal given the set of the human actions $Pr(g|obs)$.

We use the plan recognizer for recognizing human plans given observed human actions obtained by employing visual action recognition or NL understanding (when actions are commented). The recognized human plan triggers a scenario such that it is mapped to the tasks that the robot can perform to help the human to achieve their goal. Thus, the recognized human plan is mapped to a possible robot goal that is further transferred to the planner for generating a robot plan.

VI. PLAN EXECUTION AND MONITORING

The components described in the previous sections are employed by the Plan Execution Monitor component. The Plan Execution Monitor (PEM) is the central coordination unit for the execution of commands. A simplified control flow for execution of a planning task is shown in Fig. 4. To trigger PEM, a new task is sent from an external component (e.g., NLU component). Different types of tasks are accepted. For each type of task, PEM has an implementation of an *ControlMode* interface, which knows how to execute this task type. Currently, a task can be a single command or a list of

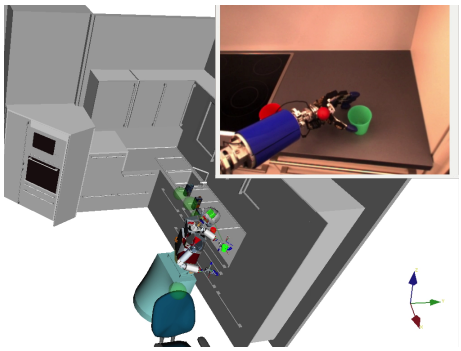


Fig. 5. Visualization of the robot's working memory during action execution and the current camera image.

goals that should be achieved. Here, we are focusing on the goal task type, which requires the planner to achieve the goal. After a new task was received, PEM calls *DomainGenerator* to generate a new domain description based on the current world state (Sec. III). This domain together with the received goal are then passed to the planner. If a plan could not be found, PEM synthesizes a feedback indicating the failure. A successful plan consists of a sequence of actions with bound variables passed back to PEM. These actions are executed one by one. Each action is associated with an ArmarX statechart [5], which controls the action execution. Action execution might fail because of uncertainties in perception and execution or changes in the environment. To account for the changes, preconditions of an action before the execution and its effects after the execution are verified by PEM. The world state is continuously updated. Fig. 5 shows the visualization of the robot working memory and the current camera image as seen by the robot. The world state observer component is queried for the current world state after each action. If any mismatches between a planned world state and a perceived world state are detected, the plan execution is considered to have failed and re-planning is triggered based on the current world state. Additionally, the statecharts report if they succeeded or failed; failing leads to re-planning. If an action was successfully executed, the next action is selected and executed. After the task completion the robot goes idle and waits for the next task.

VII. EXPERIMENTS

We tested our approach on the humanoid robot ARMAR-III [4] in a kitchen environment. In these experiments, robot skills were restricted to three primitive actions: moving, grasping, and placing. As shown in the accompanying video[3], the human agent asks the robot to help him to set the table for two people. The execution of the uttered command requires generation of a multi-step plan. The robot is supposed to generate a plan resulting in putting two cups on the table, while the human agent puts forks, knifes, and plates on the table. The task description is provided by the NLU component. The domain description is generated from the robot's memory. The task and domain descriptions constitute the input for the PKS planner, which generates a plan. The execution of the plan is monitored by the Plan Execution Monitor. The same set of actions can be triggered using plan recognition. The human agent starts setting the table and comments his actions (*I'm putting a fork on the table*). The robot recognizes the plan of the human agent and generates its own plan to help set the table, which involves putting the cups on the table. Finally, the human agent asks the robot to put a juice on the table. Given its memory, the robot has an assumption about the location of the juice. The planner generates a corresponding plan. The plan execution fails, because the juice cannot be found at the assumed location. The human agent suggests another location by saying: *The juice is at the dishwasher*. The robot updates its memory, re-plans, and executes the new plan. The video demonstrates a human-robot collaboration scenario. The robot not only generates and executes a plan given a human command, but adjusts it's plan during execution given new descriptions of the world and human actions.

Output type	Baseline				NLU pipeline			
	C	P	I	A	C	P	I	A
Planner goal	48	18	44	.66	56	25	19	.81
Human actions	43	24	33	.67	57	29	14	.86
World state	61	1	38	.62	88	1	11	.89
Total	152	43	115	**.65**	201	55	44	**.85**

TABLE I
NLU EVALUATION RESULTS.

In order to check, if our NLU framework can successfully handle the variability of natural language, we ran three experiments using the Amazon Mechanical Turk platform.[4] In the first experiment, subjects were presented with an image of a table set for two people and the following task description: *Imagine you want to have a dinner with a friend. Write a short command you would say to the robot to obtain the result shown in the image*. In the second experiment, subjects were presented with a video showing a person putting a fork and a knife on the table and the task description: *Imagine it's you performing the actions shown in the video. Describe the actions using the personal pronoun "I"*. In the third experiment, subjects were presented with a video showing a robot not finding a cup, which was located on the table, and the task description: *The robot in the video cannot find the green cup. Tell the robot where the cup is located*. 100 subjects were employed for each task.

Each of the 300 obtained utterances was processed by our NLU pipeline. We aimed at evaluating the correctness of the extracted goals, human action descriptions, and state of the world descriptions. We estimated the accuracy as the percentage of the correct and partially correct outputs. The results presented in Table I show the number of correct (C), partially correct (P), or incorrect outputs (I) as well as the accuracy (A) of a baseline method and the presented approach. As a baseline method, we extracted

[3] *http://youtu.be/87cbivmjfe8*

[4] *https://www.mturk.com*

tuples $\langle verb, agent, object, location, number \rangle$ from parsed utterances, such that $agent$ is the subject of $verb$, $object$ is its direct object, $location$ is the noun related to $verb$ by a location preposition, and $number$ is a numeral modifying $object$ or related to $verb$ by the preposition *for*. The tuples were mapped to goals, human action, and world state representations using a lookup table.

The overall baseline accuracy is .65, while the accuracy of the proposed approach is .85. The baseline method was unable to ground underspecified references, e.g., *it* in *Get the green cup! It is on the table*, which the abduction-based method was able to do, cf. Sec. IV-D. The domain and spatial axioms lacking in the baseline method also gave advantages to the abduction-based approach. Most of the errors of the abduction-based method resulted from a wrong parse. Processing of some of the utterances required deeper knowledge to make a correct inference. For example, our system did not recognize that the utterance *Robot, tonight I will dine with a friend, please set the table* implies that the table should be set for two. For human action descriptions, the errors were related to spatial inference. For example, given *I place the knife on the table. I place the fork to the left of the knife*, our system did not recognize that the knife is being placed on the table. For world descriptions, the errors were related to deictic expressions, e.g. *The cup is very straight to you, come forward*.

VIII. RELATED WORK

a) Grounding NL: Approaches to grounding NL into actions, relations, and objects known to the robot can be roughly subdivided into symbolic and statistical. Symbolic approaches rely on sets of rules to map linguistic constructions into pre-specified action spaces and sets of environmental features. In [23], simple rules are used to map NL instructions having a pre-defined structure to robot skills and task hierarchies. In [24], NL instructions are processed with a dependency parser and background axioms are used to make assumptions and fill the gaps in the NL input. In [25], background knowledge about robot actions is axiomatized using Markov Logic Networks. In [26], a knowledge base of known actions, objects, and locations is used for a Bayes-based grounding model. Symbolic approaches work well for small pre-defined domains, but most of them employ manually written rules, which limits their coverage and scalability. In order to increase the linguistic coverage, some of the systems use lexical-semantic resources like WordNet, FrameNet, and VerbNet [27], [25]. In this study, we follow this approach and generate our lexical axioms from Wordnet and FrameNet.

Statistical approaches rely on annotated corpora to learn mappings between linguistic structures and grounded predicates representing the external world. In [28], reinforcement learning is applied to interpret NL directions in terms of landmarks on a map. In [29], machine translation is used to translate from NL route instructions to a map of an environment built by a robot. In [30], Generalized Grounding Graphs are presented that define a probabilistic graphical model dynamically according to linguistic parse structures. In [19], a verb-environment-instruction library is used to learn the relations between the language, environment states, and robotic instructions in a machine learning framework. Statistical approaches are generally better at handling NL variability. An obvious drawback of these approaches is that they generate noise and require a significant amount of annotated training data, which can be difficult to obtain for each new application domain and set of action primitives.

Some recent work focuses on building joint models explicitly considering perception at the same time as parsing [31], [32]. The framework presented in this paper is in line with this approach, because abductive inference considers both the linguistic and perceptual input as an observation to be interpreted given the background knowledge.

b) Planning: With respect to the action execution, the existing approaches can be classified into those directly mapping NL instructions into action sequences [33], [34], [25] and those employing a planner [35], [27], [24], [36]. We employ a planner, because it allows us to account for the dynamically changing environment, which is essential for the human-robot collaboration. Similar to [36], we translate a NL command into a goal description.

c) Type of NL input: Although most of the NLU systems in robotics focus direct on instruction interpretations, there are a few systems detecting world descriptions implicitly contained in human commands [24], [37], [26]. These descriptions are further used in the planning context, as it is done in our approach. In addition, we detect world descriptions not embedded into the context of an instruction and process human action descriptions.

d) Linking planning, NL, and sensorimotor experience: Interaction between NL instructions, resulting symbolic plans, and sensorimotor experience during plan execution has been previously explored in the literature. In [33], symbolic representations of objects, object locations, and robot actions, are mapped on the fly to the sensorimotor information. During the execution of the predefined plans, the plan execution monitoring component evaluates the outcome of each robot's action as success or failure. In [24], the planner knowledge base is updated each time a NL instruction related to the current world state is provided and the planner re-plans taking into consideration the new information. In [35], symbolic planning is employed to plan a sequence of motion primitives for executing a predefined baking primitive given the current world state. In line with these studies, plan execution monitoring is a part of our system.

IX. CONCLUSIONS AND FUTURE WORK

We presented a symbolic framework for integrating sensorimotor experience, natural language understanding, and planning in a robotic architecture. We showed how domain descriptions can be generated from the robot memory and introduced an abduction-based NLU pipeline processing different types of linguistic input and interacting with related components such as the robot's memory, a planner, and a plan recognizer. The experimental results suggest that the

developed framework is flexible enough to process the input of untrained users and that the interaction with the human in the scenario setting allows the robot to successfully perform the task.

The main limitation of the proposed NLU framework concerns the need to define the domain mapping rules manually (Sec. IV-C). We plan to approach this limitation by employing bootstrapping from unannotated corpora to learn command-goal relations that are represented by the causation relations in texts, e.g. setting the table causes cups being on the table, cf. [38].

Concerning the domain description generation, the future work includes incorporating more information into predicate providers like, for example, object shapes or tactile sensor feedback. Another future improvement concerns switching from static predicate providers to dynamic ones that could access action outcomes and learn from successful and failed actions. Both these extensions can potentially to improve robustness and precision of the sensor-to-symbol mapping.

ACKNOWLEDGMENT

We would like to thank M. Do, C. Geib, M. Grotz, P. Kaiser, M. Kröhnert, D. Schiebener, and N. Vahrenkamp for their various contributions to the underlying system, which made this paper possible.

REFERENCES

[1] "Xperience Project." Website, available online at `http://www.xperience.org`.

[2] M. Do, J. Schill, J. Ernesti, and T. Asfour, "Learn to wipe: A case study of structural bootstrapping from sensorimotor experience," in *Proc. of ICRA*, 2014.

[3] F. Wörgötter, C. Geib, M. Tamosiunaite, E. E. Aksoy, J. Piater, H. Xiong, A. Ude, B. Nemec, D. Kraft, N. Krüger, M. Wächter, and T. Asfour, "Structural bootstrapping - a novel concept for the fast acquisition of action-knowledge," *IEEE Trans. on Autonomous Mental Development*, vol. 7, no. 2, pp. 140–154, 2015.

[4] T. Asfour, K. Regenstein, P. Azad, J. Schroder, A. Bierbaum, N. Vahrenkamp, and R. Dillmann, "ARMAR-III: An integrated humanoid platform for sensory-motor control," in *Proc. of Humanoids*, pp. 169–175, 2006.

[5] N. Vahrenkamp, M. Wächter, M. Kröhnert, K. Welke, and T. Asfour, "The ArmarX Framework - Supporting high level robot programming through state disclosure," *Information Technology*, vol. 57, no. 2, pp. 99–111, 2015.

[6] P. Azad, D. Münch, T. Asfour, and R. Dillmann, "6-dof model-based tracking of arbitrarily shaped 3d objects," in *Proc. of ICRA*, pp. 5204–5209, 2011.

[7] K. Welke, P. Kaiser, A. Kozlov, N. Adermann, T. Asfour, M. Lewis, and M. Steedman, "Grounded spatial symbols for task planning based on experience," in *Proc. of Humanoids*, pp. 484–491, 2013.

[8] T. A. P. Azad and R. Dillmann, "Combining harris interest points and the sift descriptor for fast scale-invariant object recognition," in *Proc. of IROS*, pp. 4275–4280, 2009.

[9] J. R. Hobbs, M. E. Stickel, D. E. Appelt, and P. A. Martin, "Interpretation as abduction," *Artif. Intell.*, vol. 63, no. 1-2, pp. 69–142, 1993.

[10] H. Soltau, F. Metze, C. Fügen, and A. Waibel, "A one-pass decoder based on polymorphic linguistic context assignment," in *Proc. of ASRU*, pp. 214–217, 2001.

[11] N. Inoue, E. Ovchinnikova, K. Inui, and J. R. Hobbs, "Weighted abduction for discourse processing based on integer linear programming," in *Plan, Activity, and Intent Recognition*, pp. 33–55, 2014.

[12] J. R. Hobbs, "Ontological promiscuity," in *Proc. of ACL*, pp. 60–69, 1985.

[13] J. Bos, "Wide-Coverage Semantic Analysis with Boxer," in *Proc. of STEP*, Research in Computational Semantics, pp. 277–286, 2008.

[14] E. Ovchinnikova, R. Israel, S. Wertheim, V. Zaytsev, N. Montazeri, and J. Hobbs, "Abductive Inference for Interpretation of Metaphors," in *Proc. of ACL Workshop on Metaphor in NLP*, pp. 33–41, 2014.

[15] E. Ovchinnikova, A. S. Gordon, and J. Hobbs, "Abduction for Discourse Interpretation: A Probabilistic Framework," in *Proc. of JSSP*, pp. 42–50, 2013.

[16] C. Fellbaum, *WordNet: An Electronic Lexical Database*. 1998.

[17] C. F. Baker, C. J. Fillmore, and J. B. Lowe, "The Berkeley FrameNet project," in *Proc. of COLING-ACL*, pp. 86–90, 1998.

[18] E. Ovchinnikova, *Integration of world knowledge for natural language understanding*. Springer, 2012.

[19] D. K. Misra, J. Sung, K. Lee, and A. Saxena, "Tell me dave: Context-sensitive grounding of natural language to manipulation instructions," *Proc. of RSS*, 2014.

[20] R. Ros, S. Lemaignan, E. A. Sisbot, R. Alami, J. Steinwender, K. Hamann, and F. Warneken, "Which one? grounding the referent based on efficient human-robot interaction," in *Proc. of RO-MAN*, pp. 570–575, 2010.

[21] R. P. Petrick and F. Bacchus, "A Knowledge-Based Approach to Planning with Incomplete Information and Sensing," in *Proc. of AIPS*, pp. 212–222, 2002.

[22] C. W. Geib, "Delaying Commitment in Plan Recognition Using Combinatory Categorial Grammars," in *Proc. of IJCAI*, pp. 1702–1707, 2009.

[23] P. E. Rybski, K. Yoon, J. Stolarz, and M. M. Veloso, "Interactive robot task training through dialog and demonstration," in *Proc. of HRI*, pp. 49–56, 2007.

[24] R. Cantrell, K. Talamadupula, P. W. Schermerhorn, J. Benton, S. Kambhampati, and M. Scheutz, "Tell me when and why to do it!: run-time planner model updates via natural language instruction," in *Proc. of HRI*, pp. 471–478, 2012.

[25] D. Nyga and M. Beetz, "Everything robots always wanted to know about housework (but were afraid to ask)," in *Proc. of IROS*, pp. 243–250, 2012.

[26] T. Kollar, V. Perera, D. Nardi, and M. Veloso, "Learning environmental knowledge from task-based human-robot dialog," in *Proc. of ICRA*, pp. 4304–4309, 2013.

[27] D. J. Brooks, C. Lignos, C. Finucane, M. S. Medvedev, I. Perera, V. Raman, H. Kress-Gazit, M. Marcus, and H. A. Yanco, "Make it so: Continuous, flexible natural language interaction with an autonomous robot," in *Proc. of the Grounding Language for Physical Systems Workshop at AAAI*, 2012.

[28] A. Vogel and D. Jurafsky, "Learning to follow navigational directions," in *Proc. of ACL*, pp. 806–814, 2010.

[29] C. Matuszek, D. Fox, and K. Koscher, "Following directions using statistical machine translation," in *Proc. of ACM/IEEE*, pp. 251–258, 2010.

[30] T. Kollar, S. Tellex, M. R. Walter, A. Huang, A. Bachrach, S. Hemachandra, E. Brunskill, A. Banerjee, D. Roy, S. Teller, *et al.*, "Generalized grounding graphs: A probabilistic framework for understanding grounded language," *JAIR*, 2013.

[31] J. Krishnamurthy and T. Kollar, "Jointly learning to parse and perceive: Connecting natural language to the physical world," *Trans. of ACL*, vol. 1, pp. 193–206, 2013.

[32] C. Matuszek, N. FitzGerald, L. Zettlemoyer, L. Bo, and D. Fox, "A joint model of language and perception for grounded attribute learning," *arXiv preprint arXiv:1206.6423*, 2012.

[33] M. Beetz, U. Klank, I. Kresse, A. Maldonado, L. Mosenlechner, D. Pangercic, T. Ruhr, and M. Tenorth, "Robotic roommates making pancakes," in *Proc. of Humanoids*, pp. 529–536, 2011.

[34] C. Matuszek, E. Herbst, L. Zettlemoyer, and D. Fox, "Learning to parse natural language commands to a robot control system," in *Experimental Robotics*, pp. 403–415, Springer, 2013.

[35] M. Bollini, S. Tellex, T. Thompson, N. Roy, and D. Rus, "Interpreting and executing recipes with a cooking robot," in *Experimental Robotics*, pp. 481–495, 2013.

[36] J. Dzifcak, M. Scheutz, C. Baral, and P. Schermerhorn, "What to do and how to do it: Translating natural language directives into temporal and dynamic logic representation for goal management and action execution," in *Proc. of ICRA*, pp. 4163–4168, 2009.

[37] F. Duvallet, M. R. Walter, T. Howard, S. Hemachandra, J. Oh, S. Teller, N. Roy, and A. Stentz, "Inferring maps and behaviors from natural language instructions," in *Proc. of ISER*, 2014.

[38] Z. Kozareva, "Cause-Effect Relation Learning," in *Proc. of TextGraphs-7*, pp. 39–43, ACL, 2012.

frontiers

The ArmarX Statechart Concept: Graphical Programming of Robot Behaviour

Mirko Wächter *, **Simon Ottenhaus, Manfred Kröhnert, Nikolaus Vahrenkamp and Tamim Asfour**

High Performance Humanoid Technologies Lab (H^2T), Institute for Anthropomatics and Robotics (IAR), Karlsruhe Institute of Technology (KIT), Karlsruhe, Germany

Correspondence*:
Mirko Wächter
High Performance Humanoid Technologies Lab (H^2T), Institute for Anthropomatics and Robotics (IAR), Karlsruhe Institute of Technology (KIT), Karlsruhe, Germany,
mirko.waechter@kit.edu

ABSTRACT

Programming sophisticated robots such as service robots or humanoids is still a complex endeavor. Although programming robotic applications requires specialist knowledge, a robot software environment should support convenient development while maintaining full flexibility needed when realizing challenging robotics tasks. In addition, several desirable properties should be fulfilled, such as robustness, reusability of existing programs, and skill transfer between robots. In this work, we introduce the ArmarX statechart concept, which is used for describing control- and data-flow of robot programs. The event-driven statechart approach of ArmarX helps realizing important features such as increased robustness through distributed program execution, convenient programming through graphical user interfaces, and versatility by interweaving dynamic statechart structure with custom user-code. We show that using hierarchical and distributed statecharts increases reusability, allows skill transfer between robots, and hides complexity in robot programming by splitting robot behavior into control-flow and functionality.

Keywords: Robot Software Framework, Robot Programming, Statecharts, Graphical User Interfaces, Distributed Processing

1 INTRODUCTION

Programming complex robots like humanoids is challenging and is often divided into at least two domains. One being low-level control which is essential for smooth execution, system stabilization, safety, and consideration of dynamic effects. High-level robot programming on the other hand copes with perception, task and motion planning, user interaction, memory concepts, and reusability of robot skills. Well-designed robot software frameworks should support the development of complex robot programs on all system levels. Therefore, a framework needs to provide well-defined interfaces for all available robot components and the flexibility to additionally implement application- or task-specific behaviors. In addition, a basic set of robot skills (i.e. robot programs for a special behavior) should be available which can be used to assemble more complex robot programs. One challenge in building a robot framework is to provide means for doing this in a robust and convenient way.

In this work, we focus on high-level robot programming and discuss how using hierarchical, distributed statecharts for encoding robot skills aids in achieving convenient programming and reusable, transferable robot behaviours. Possible candidates of statechart implementations must meet the following requirements to be considered eligible: full control over data-flow and control-flow, local scoping of data similar to encapsulation in programming, runtime-reconfigurability as well as runtime introspection. It should also not be necessary to recompile programs upon structural or control-flow changes. Furthermore, a graphical user interface is desirable in order to reduce the unavoidable complexity of describing robot behavior and to minimize development and comprehension efforts. This convenience feature should provide means for defining and parametrizing both control- and data-flow, online visualization of active states and transitions in running programs, and a convenient way to incorporate custom user code. Additionally, a code generator should be provided for enforcing type-safety and catching errors in user code as early as possible as well as allowing source code auto-completion in development environments of statechart related datatypes and functions.

We will discuss the statechart concept of the robot development environment ArmarX Vahrenkamp et al. (2015) in detail and show how it provides both reusability of high-level robot skills realized as distributed, hierarchical statecharts and the possibility to add user code with access to the external robot components.

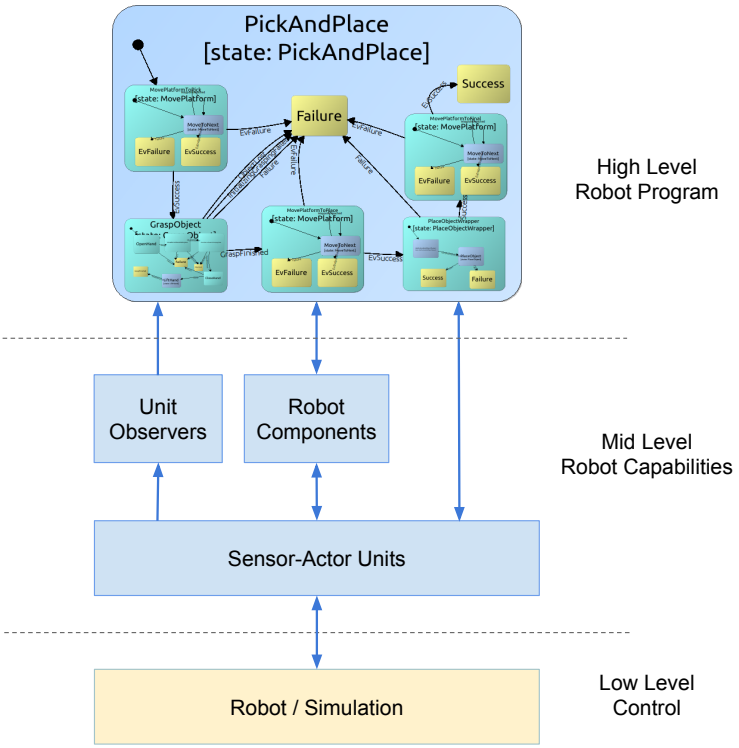


Figure 1. Basic structure of ArmarX. The low level part is abstracted through the Sensor-Actor Unit concept. These framework components realize hardware access and simulation and hide the low level communication from higher level layers of the robot software. On the mid level, robot capabilities such as perception, planning and motion generation, are implemented in a network transparent way. The high level layer comprises a set of robot skills, realized as statecharts, which are used for assembling complex robot programs.

2 RELATED WORK

Robot Development Environments (RDEs) have co-evolved with the increasing complexity and capabilities of modern robots. Taking a closer look at recent RDEs, there has been an agreement on the necessity of distributed processing for complex robotic systems (e.g. Metta et al. (2006); Quigley et al. (2009); Bruyninckx et al. (2003); Scholl et al. (2001); Ando et al. (2008)). Communication in such distributed systems is often performed via middlewares such as CORBA (2006) or Ice Henning (2004). In other cases, specialized middleware systems or messaging protocols have been developed based on task specific requirements.

2.1 Robot Development Environments

Apart from distribution and communication, RDEs differ depending on which part of robot programming they target. MiRPA Finkemeyer et al. (2007) for example provides a low-level message oriented realtime communication middleware. OpenRTM Ando et al. (2008) is situated on the lower control level and provides a component model with input, output, and configuration interfaces as well as basic execution statemachines (inactive, active, error states). MOOS Newman (2008) is located on a similar level than OpenRTM and provides a publish-subscribe based communication and data exchange between MOOS applications via a central database. OpenRDK Calisi et al. (2008) is also a low-level framework and uses agents as main abstraction which dynamically instantiate modules containing functionality. Modules communicate through a blackboard type mechanism and can access input, output, and parameter data of any other module. YARP Metta et al. (2006), being used for the iCub robots Metta et al. (2008), provides low level communication as a basis for higher level robot capabilities implemented in the iCub software. Last, ROS Quigley et al. (2009) and Orocos Bruyninckx et al. (2003) lean towards the implementation of higher level system capabilities. In ROS, software modules called *nodes* span a peer to peer network and send messages, whereas Orocos provides an explicit component model and separates the structure of the control system from its functionality.

2.2 Statecharts

Besides the original paper of Harel (1987) there are countless other publications Samek (2002); Coleman et al. (1992); Von der Beeck (1994) and software projects Angermann et al. (2014); Yakindu (2015); EasyCODE (2015) on statecharts for a variety of different use cases. Here, we focus on software realizations of statecharts and statecharts in the context of robotics. There are several projects offering frameworks for developing your own statecharts. The well-known de-facto extension of C++ Boost Huber (2007) contains a subproject called the Boost Statechart Library, which offers a statechart implementation close to the original formalism of Harel. It has the unique feature of specifying the statecharts with C++ templates and achieving compile-time statechart validation. While this is a valuable feature to ensure valid statecharts, it does not fit our requirements. For our purposes, we require runtime-reconfigurability and no recompilation on layout changes as well as runtime introspection, which is difficult to achieve if the structure is specified implicitly with C++ templates. On the side of graphical tools, the statechart graphical modeling tool QM Quantum Leaps (2015) provides means for designing and implementing event-driven low-level statecharts for embedded systems with a strong focus on traceability at the code level. The complete statecharts are generated into C++ code, meaning that for statechart structure changes recompilation is necessary. In our statecharts, we aim to generate code only to catch errors in the user-code as early as possible and for IDE auto-completion purposes. In Yakindu (2015) another graphical statechart modeling tool is presented, aiming at usability and assistance inside the editor during typing. Though, it seems to target low-level

statecharts like QM with limited data-flow control, which is of high importance in the ArmarX statecharts as described later.

Statecharts are widely used in robotics to control behavior on a high level Klotzbücher and Bruyninckx (2012); Merz et al. (2006); Billington et al. (2010); Bohren and Cousins (2010); Nilsson and Center (1973) since they address several of the problems of robotics like state-based control and event-triggered execution. In the well-known RDE ROS Quigley et al. (2009), an approach called SMACH Bohren and Cousins (2010) is employed that focuses on data-flow in statecharts. However, scope of data-flow in ROS SMACH is handled differently. In ROS SMACH, a child state can access all data used by its parent state. This eases programming because it is easy to operate with data on several levels, but also violates the principle of modularity of states and creates implicit data dependencies between states. A state using datafields of a parent state cannot easily be reused in another state since it depends on the availability of specific datafields in a parent state. Due to this, we do not allow data scopes over several state levels in ArmarX and require explicit mapping of data between state levels. Also, ROS SMACH only supports graphical online visualization of states, but does not provide any tools for graphical programming. In many aspects, the statecharts of ArmarX are similar to the restricted Finite State Machine (rFSM) Klotzbücher and Bruyninckx (2012) from Orocos Bruyninckx et al. (2003). However, the statecharts in Orocos focus on coordination of components but offer only very limited data-flow control.

2.3 Graphical Robot Programming

When developing high level software on a robotic platform, it is desirable to configure and connect existing components using a graphical user interface to prevent writing repetitive and therefore error prone source code. This allows new as well as experienced users to intuitively and efficiently combine mid and high level components in order to create a functional system structure. Since writing software is one of the main challenges in robotics for beginners, such as students, Graphical Robot Programming offers a great entry point. It removes the obstacle presented by syntax and control flow of a conventional programming language Rahul et al. (2014). Graphical software development often combines complexity hiding by connecting modular components on a macroscopic scale with the option to write low level software facilitating control tasks on joint level or performing motions in Cartesian space Pot et al. (2009). Graphical and tabular representations are an accessible way to model system behavior in the context of simulation, validation and consistency checking of a system design before final implementation MathWorks (2015c). Hirzinger and Bauml (2006) are using Simulink MathWorks (2015b) in conjunction with MATLAB MathWorks (2015a) to graphically model subsystems to later generate executables running on a realtime target. The Microsoft Visual Programming Language Microsoft (2012b), as part of Microsoft Robotics Developer Studio Microsoft (2012a), proposes developing the complete logic and program flow in a visual development environment as it lowers the bar for beginner programmers. However, we decided to limit the visual development in ArmarX to the definition of structure, used data types and data-flow in our statecharts for the benefit that the user can write unrestricted C++ code. The RDE YARP Metta et al. (2006) also offers means of graphical programming with the *gyarpbuilder* Paikan (2014), yet on another level. With *gyarpbuilder* it is possible to connect continuous input and output data of components graphically and to insert arbitrators in these connections to manipulate data-flow easily. *RtcLink* AIST (2015) from the OpenRTM project offers a GUI to operate on RT-Components existing in a network. It can activate and deactivate components as well as connect their ports. It leverages the capabilities of an established IDE by providing the GUI as an Eclipse plugin.

3 ARMARX STATECHARTS

The complexity of multi-component systems can be challenging in terms of program and data flow. Hence, only skilled experts are capable of designing and realizing highly connected software systems as they are needed on humanoid robots. With the hierarchical statechart paradigm one can easily visualize the whole application by hiding the complexity through hierarchical views on the application. In the following, we will introduce the basic concepts of hierarchical statecharts and the extensions in the robotics context, and we will discuss the realization of distributed hierarchical statecharts in the robotics programming environment ArmarX. Therefore, we discuss the original statechart formalism by Harel (1987). Following this, we present the statechart principles in ArmarX and the differences to Harel's formalism, and we explain the technical details of the ArmarX statecharts.

3.1 Statechart Concept

Harel (1987) proposed in his work a new formalism to represent and describe complex systems. The limitations of finite state machines (FSM, Gill et al. (1962)) motivated him to develop the more powerful *statecharts* representation. Like FSMs, statecharts consist of states and transitions between these states, but Harel introduced several new notation features compared to FSMs. The main aspect of statecharts is to reduce the complexity for the human developer. First of all, Harel introduced a *hierarchy* of states, meaning states can contain a full statechart themselves. However, these state levels are not completely separated from upper or lower levels, since he proposed *inter-level-transitions* to directly jump into sub-states. Further, *orthogonality* is used to parallelize the execution of different states on one statechart level. Each orthogonality level of one state is executed in parallel, independently of the other levels. A *history*-connector was added to give states a memory, controlling which sub-state is entered when a state is re-entered. Based on the fulfillment of conditions, *Condition*-connectors control which state a transition leads to. At last, each state can be connected to actions, being triggered during different phases of the state: entering, leaving and an action that is executed repeatedly as long as the state is active.

3.2 Statecharts in ArmarX

With ArmarX, we provide a generic robotics software programming environment which combines event-driven programming with distributed component-based robot applications. A robot framework in ArmarX consists of several distributed components providing access to sensors and actors (i.e. the hardware), offering computation functionality, and realizing a robot memory system as a common data source for the robot software. On top of these robot components, the ArmarX statechart mechanism can be employed to define the structure of the mid- to high-level robot behavior (i.e. the program flow). In order to gain full flexibility within the robot applications, the programmer can use well-defined entry points to implement user-specific source code. By separating structure from behavior, the task of building new robot software applications can be supported through graphical user interfaces while maintaining full flexibility on source code level.
ArmarX provides means of designing such statecharts textually and graphically with the possibility to link them with user-code to perform custom operations. The graphical way is presented in section 4.1 in detail.

3.2.1 Differences to Harel's formalism

The statecharts in ArmarX differ in several points from Harel's original formalism. We omitted some of Harel's features to comply with our design principles (explained in the next paragraph) and to simplify the statechart design process for the developer. We added one important aspect to our statechart, which is not covered in Harel's formalism: data-flow specification and control. The *hierarchy* and *condition*-connectors

are available like in the original statecharts. We do not directly allow *inter-level-transitions* to not violate the principle of modularity. The *history*-connector is not available since it conflicts with the data-flow specifications, and to reduce side-effects during execution as well as to simplify the comprehension of the current state of the system during introspection. Each entering of a state must provide the same internal state. *Orthogonality* is currently only available in a smaller scope. Each active state can have an asynchronous user-code function executed in a separate thread. Thus, the different hierarchy levels can run in parallel.

3.2.2 Design Principles

Key principles of the ArmarX statecharts are: modularity, re-usability, runtime-reconfigurability, decentralization and state-disclosure.

- *Modularity* in our statecharts comes naturally through the individual states and explicitely specified input and output. There is no direct interaction allowed between sub-states of different parent states.
- *Re-usability* is ensured, since every state can be used as a sub-state in any other state and has a specific interface for interaction. The interface is specified with the state parameters like the parameters of a function.
- *Runtime-reconfigurability* means that a statechart can be defined in configuration files and that the statechart structure can be changed completely at runtime.
- *Decentralization* means that a statechart does not need to be resided in one process, but can be spread over several processes and hosts. This enables load balancing and robustness. A crashed distributed state component would not crash the whole statechart, but would just create an event for higher layers that this specific state has failed (see 5.1 for an example of crash recovery).
- *State-disclosure* means that the current state and all its parameters can be inspected at runtime and logged for future behavior adaptation via a network interface (see section 4.5).

3.3 Conceptual Details

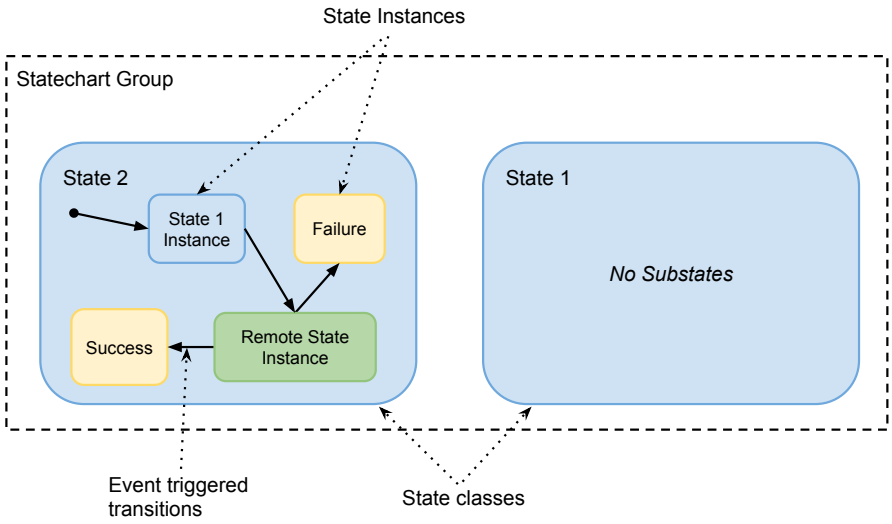


Figure 2. Statecharts in ArmarX are organized in groups. States are defined as state classes (State 1 & 2) and instances thereof can be used in other states. States are changed through transitions triggered by (conditional) events. Statecharts end with end-states (Failure, Success) or external events. State instances can also point to remotely located state classes (green state) for distributed processing. A statechart is a state itself and can be used again in other states.

In the following, we are describing the main technical aspects of the ArmarX statecharts: sub-states, transitions, state phases, data-flow, interfacing with external components, distributed statecharts, and the dynamic statechart structure.
Following the composite pattern, a statechart in ArmarX is a state itself (see Figure 2). A state can contain sub-states and transitions between these sub-states. Every sub-state is a state, so that all states can be used as sub-states on a higher hierarchy level. States and sub-states can be compared to classes and instances in object-oriented programming. When the states are defined they are not yet used anywhere, like classes. After instantiation, states are always sub-states of another state. Transitions between sub-states are triggered by events. Transitions do not only specify control-flow but also data-flow by attaching a parameter mapping to each transition. This mapping contains instructions on how to fill the input-parameters of the next state (explained in detail in paragraph 3.3.4).

3.3.1 State Instance Types

States are always of the same type. But there are a few different types of state instances, i.e. sub-states:

- *LocalState*
  Local states are normal state instances with no special features.
- *End-state*
  End-states trigger leaving the parent state immediately. They cannot contain sub-states or execute any user code. End-states are one way to specify outgoing transitions of the parent state. The name of an end-state specifies the name of the transition.
- *RemoteState*
  Remote states behave like local states but point internally to a specific state in another process.
- *DynamicRemoteState*
  Dynamic remote states are similar to remote states, but are like generic pointers. On entering, a dynamic remote state morphs into a specific remote state based on parameters mapped during the transition.

3.3.2 Transitions

Transitions describe connections between states. The developer specifies start- and end-state for each transition and which on event the transition is triggered. The associated event to a transition can be fired immediately when the state is entered or when a specified condition is fulfilled. It is possible to construct arbitrary complex conditions with a boolean algebra and pre-defined or custom literals. Boolean terms are installed in sensor-observers and evaluated on each associated sensor update. For example, one could specify a smaller-condition on the distance between the hand and an object. This condition is installed in the *ObjectMemoryObserver* since object relations stored in the object memory are observed here. Then, an event would be sent to the state that installed the condition when the distance goes below a threshold.
Additionally, there is the initial transition, which does not have a start-state. The initial transition is triggered immediately when the parent state is entered and leads to entering the initial sub-state of a state. Thus, when the top-level state of a state hierarchy is entered all initial sub-states of each level are subsequently entered.
End-States have the special ability to specify outgoing transitions in the parent state where they are used. These transitions are triggered immediately when an end-state is entered. Thus, when an end-state is entered, this end-state and the parent state are left.

Data flow is realized through a parameter mapping definition which is attached to transitions (see paragraph 3.3.4). One important detail to mention is that transitions can only created between sub-states of the same parent state, unlike in Harel statecharts. We decided to create this restriction to keep the modularity principle of states. If states would have transitions to other hierarchy levels or other parent states, the parent state could not be reused without destroying that transition.

3.3.3 State Phases

During the visit of a state different phases are passed through: *OnEnter*, *running*, *onBreak*, and *onExit*. To enable developers to execute own code in a state, each phase is linked to a user code function, i.e. C++ code. The functions *onEnter*, *onBreak*, and *onExit* are designated for setup and clean up. The *running* phase is meant for complex, long running computations. The order of execution of the phases is as follows: *onEnter*, *running* and then *onBreak* or *onExit*. Before entering a state (i.e. phase *onEnter*), the parameters (explained in next paragraph) are mapped or set to default values. In the *onEnter* phase local variables can be set to be mapped into sub-states or prepared for later phases. When a transition is triggered, the *onExit* or *onBreak* phase is entered. Which phase is executed depends on the level where the transition was triggered: as aforementioned, statecharts are hierarchical. Thus, it is possible for a higher state to receive an event, although its sub-states are not finished yet. In this case, the sub-statecharts cannot finish in an expected manner. To give the developer an option to deal with this unexpected behavior, each state provides the *onBreak* phase. If no extra behavior is specified for the *onBreak* phase, the user-code function of the *onExit* phase is executed.
Whenever a state is entered, its initial sub-state is entered as well. This means that after executing the *onEnter* phase of a state, the *onEnter* phase of the initial sub-state is also executed immediately.
During each of these phases, the developer can access different parameter dictionaries in the user-code functions, which are explained in the next paragraph.

3.3.4 Data-flow

All states are equipped with input- and output-parameter dictionaries to decouple states from external global data storage. Input-parameters are read-only in user-code functions and specify all parametrization the state needs for its computations. Output-parameters must be set in the user-code functions, contain the results of a state, and can be used as source for input-parameters of the next state.
Additionally, so called local-parameters are provided and accessible by the user code. Local-parameters are intended to be used for temporal local storage of parameters that are passed down to sub-states' input, passed up from sub-states' output, or passed between different state phases. Once a state is left, all parameters are reset in order to avoid side effects of previous visits.
Each parameter dictionary field consists of a string identifier and a variant data-type that can manifest itself into arbitrary types. ArmarX already provides the basic types boolean, integer, float, double and string as well as several types associated with robotics like vectors, matrices, 3D poses or probability distributions. If needed, developers can implement new types easily.
These parameter dictionaries are defined by the developer and specify the interface of each state, i.e. which data it needs for execution. Each parameter can be optional, can have a default value[1], and/or can be filled from several sources. We call this parameter mapping. When a state is used, its non-optional input-parameters without default values need to be connected with other parameters of the same type. Thus, a parameter mapping for each of these input-parameters needs to be created for each state instance. The developer can choose between mapping from the output of a previous state from the same hierarchy

[1] Consequently, if parameters have a default value, the optional flag does not make much sense any more, thus these two boolean flags basically form a tri-state.

level, the input- or local-parameters of the parent state, or from a parameter attached to the transition-event. Additionally, developers can map values from the output of a state to the local- or output-parameters of the parent state. Later, when another substate needs the calculated value as an input parameter, the local parameter is mapped to that input parameter. For example, generic counter states can be implemented following this pattern, so that counting loop sequences of states can be defined without writing any additional specialized custom code. With this, it is possibly to pass data from a sub-state to another state later in the chain more easily. Otherwise the parameters would need to be mapped from state to state. Figure 3 shows the different types of mappings during transitions.

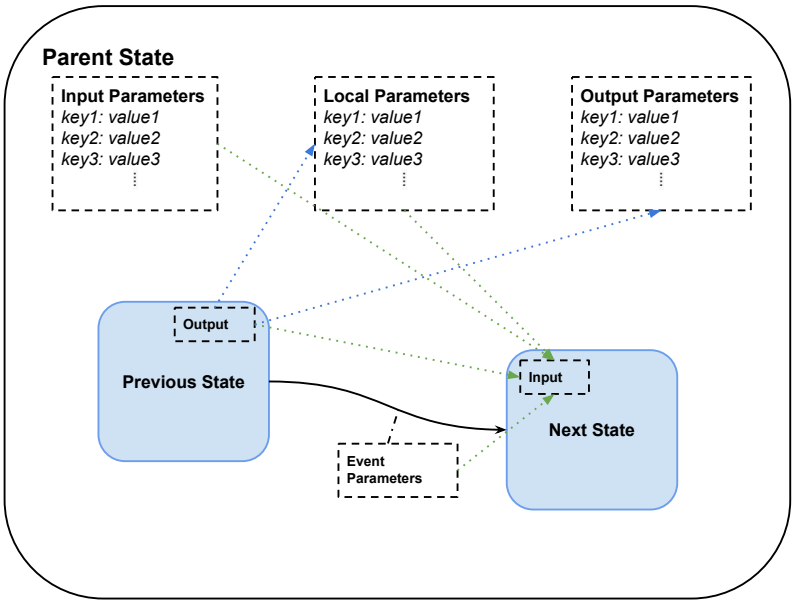


Figure 3. Available types of parameter mapping during transitions. The green arrows show possible mappings to the input parameters of the next state. The blue arrows show possible mappings from the output of the previous state to the local and output parameters of the parent state. These mappings happen after leaving the previous state and before entering the next state.

3.3.5 Interfacing with External Components

Statecharts that can only access functionality and data of themselves are not particularly useful for robotics. Therefore, they must be able to access all available components. Since ArmarX is a heavily distributed system, it cannot be assumed that required components are running in the same process or on the same host. Hence, states require network proxies to these components and it should be ensured that a state is only started if all required components are available. Dependencies for a group of states can therefore be defined in a so called *StatechartContext*, which manages dependencies and enables states to communicate with external components.

3.3.6 Distributed Statecharts

To our knowledge, one unique feature of ArmarX is the possibility to distribute statecharts over several processes or hosts. To this end, states in ArmarX are organized in groups containing states that are semantically similar (e.g. states needed for direct motion control) and share the same dependencies to external components. Each group is executed as one component in a so called *RemoteStateOfferer*.

These *RemoteStateOfferers* offer states to be used by others states as *RemoteStates* over the network. For robustness, each *RemoteStateOfferer* is located in its own process. Thus, a *RemoteState* is inserted whenever a state uses a state of another group as a sub-state. This process is completely transparent for the developer. The only difference to a local state is that the *RemoteStateOfferers* name needs to be specified in addition to the state name. Theoretically, each state could have its own group for maximized robustness. Since distributed statecharts are slower than local statecharts, developers need to decide carefully when to split statecharts in more than one group. Another advantage of distributed statecharts is the possibility to deploy them close to their components. A statechart that makes heavy use of the robot's memory should ideally be located on the same host as the database servers, whereas a visual servoing statechart should be close to the vision system and the host where joint-level control takes place. Figure 4 depicts the linkage between different statechart groups and *RemoteStates*.

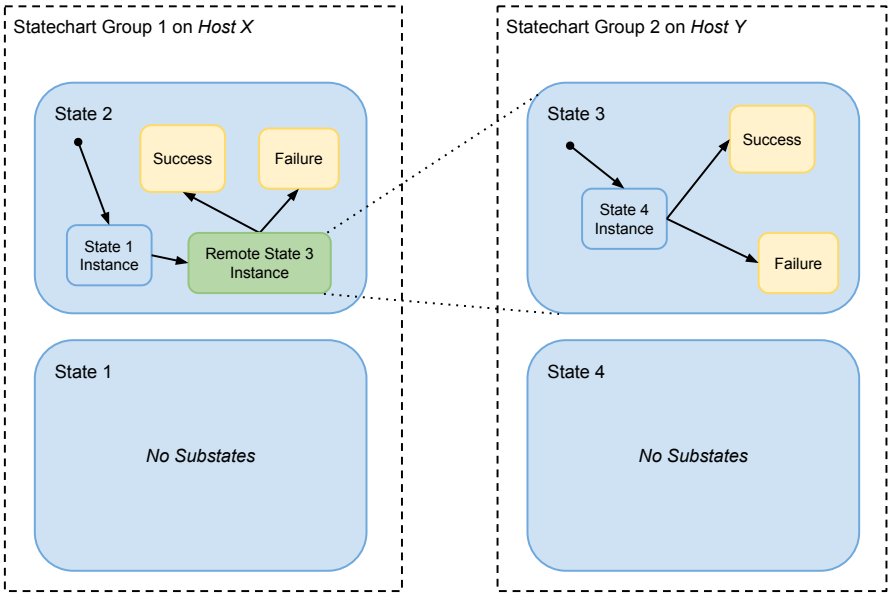


Figure 4. Statecharts in ArmarX are organized in groups which can be distributed over several processes and hosts. Each statechart group resides by default in one process. By creating *RemoteState* instances it is possible to incorporate states of another group transparently into a statechart.

3.3.7 Dynamic Structure

Most statechart frameworks fix the statechart structure to the structure designed by the developer. For a learning robot or a robot with a planning component it is desirable to have a dynamically changeable statechart structure. A planning component might want to interchange one state in a statechart with the currently planned skill. To this end, it is possible to specify so called *DynamicRemoteStates* which behave like generic pointers in the C programming language. As the name suggests, a *DynamicRemoteState* connects to a state in another (or its own) process. It decides upon entering, which state it loads into itself based on specific parameters passed by the transition. Additionally, it can specify more parameters that are mapped into the loaded state. The correctness and completeness of the parameters is checked at runtime, when the state is loaded.

3.4 Textual Statechart Specification

While the advised method to create statecharts is to use the Statechart Editor (see section 4.1) it is also possible to specify statecharts textually:

```
void defineSubstates()
{
    //add sub-states
    setInitState(addState<InitialState>("Initial"));

    StateBasePtr finalSuccess = addState<SuccessState>("Success");
    StateBasePtr finalFailure = addState<FailureState>("Failure");

    // add transitions
    addTransition<Next>(getInitState(),
                        getInitState());
    addTransition<TimerExpired>(getInitState(),
                                finalFailure);
    addTransition<Success>(getInitState(),
                           finalSuccess);

}
```

First, each state needs to be added with its state class (TemplateParameter) and the instance name (parameter of *addState()*). Afterwards, transitions between these sub-states can be created by specifying the start and end-state and on which event this transitions should be triggered.

4 THE STATECHART CONCEPT EMBEDDED INTO ARMARX

Statecharts can be implemented in various ways by using a lookup table for transitions, by implementing transition tables via switch-case statements, by implementing an object oriented state pattern, etc. Since all these approaches are based on writing code to perform the state transitions, a lot of repetitive textual description is usually necessary to define large statecharts. This textual description becomes rapidly incomprehensible for other developers. To overcome this tedious and error prone work, a graphical statechart editor was developed for ArmarX statecharts.

4.1 Statechart Editor: Defining Control Flow and Data Types

The goal of the statechart editor is to enable all users to create new statecharts with sub-states, to define input and output parameters, and to connect states with transitions. The editor covers all major use cases related to editing a statechart: creation of structure, definition of control flow and definition of data-flow during transitions. The user is not required to write any custom code to create a functional statechart. We decided to store the statechart definition in a custom xml-based format.

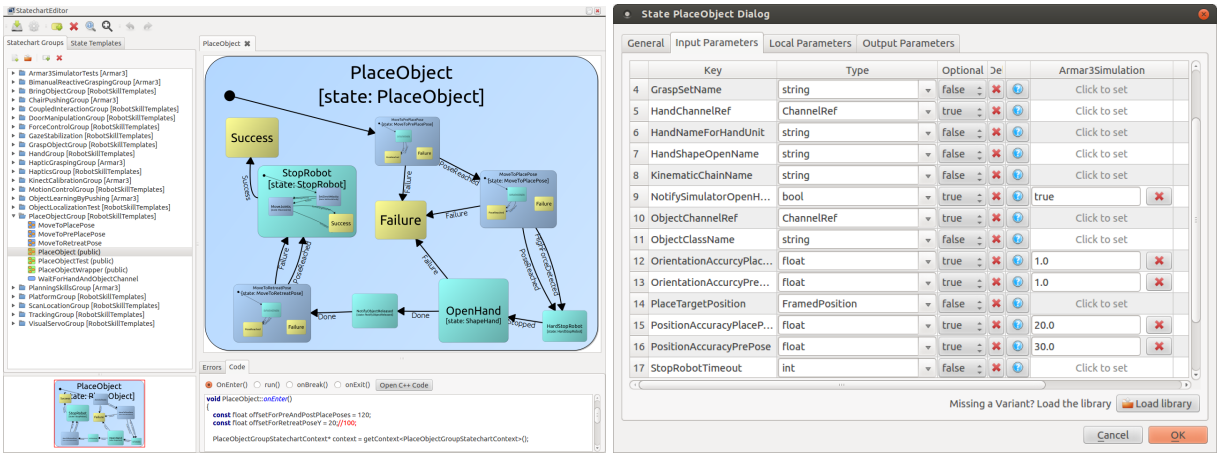
Figure 5a shows the main window of the statechart editor. Statecharts are organized in statechart groups. A statechart group can contain multiple statecharts and sub-states. For further organization of statecharts, folders and sub-folders are available. All statechart groups are listed on the left side of the main window.

The user can open any statechart from the state library for graphical editing or reuse a statechart by including it as a sub-state within another statechart.

When a statechart is opened for graphical editing, it is displayed on the right side of the editor. The editor offers a variety of options to edit a statechart including specialized dialogs and context menus.

- *Sub-states*
  By dragging a statechart from the statechart library into the right editing area a sub-state is created. A state can be reused multiple times as a sub-state within a statechart. The editor displays sub-states in two different colors: states from within the same statechart group are colored blue; states from different statechart groups are displayed in turquoise (*RemoteState*). *DynamicRemoteStates* are violet.
- *End-states*
  End-states are special sub-states, which are colored yellow. Each end-state implicitly creates an outgoing event/transition. When the statechart transitions to an end-state, the execution within the statechart is terminated. Additionally, the corresponding event/transition is triggered so that control flow moves back to the parenting statechart where execution is continued. When transitioning to an end-state, a statechart completes by terminating its execution entirely if no parenting statechart is present, i.e. the statechart in question is the top-level statechart.
- *Events & Transitions*
  As mentioned before, an end-state implicitly creates an event, which in turn implicitly creates an outgoing transition. When a statechart is initially added as a sub-state, all outgoing transitions of this sub-state are displayed as detached transitions. Transitions can be connected to other sub-states by dragging them onto the target sub-state. To create a valid statechart, all transitions have to be connected from a source state to a target state. The target state can be another sub-state, end-state or the source state itself in case of a reflexive transition. When no detached transitions remain, the transitioning behavior of the statechart is fully defined, which implies that no event is left unhandled. Additional events can be specified in the state properties, which are fired from the code directly or on fulfilled conditions.
- *State Parameters*
  Each state has a list of input, output and local parameters. A parameter is defined by its name, data type and an optional default value. Figure 5b displays the input parameters of the *PlaceObjectSkill* as it is used in ArmarX. Role and usage of the three parameter types is similar to those of parameters, return values and local variables of functions in imperative programming languages.
- *Data-Flow*
  A transition can be accompanied by several data mappings that define the data-flow within the statechart during this transition. The statechart editor does not support global data storage since global data storage breaks the concept of data encapsulation. Data is only passed between states during a transition. Data can be passed in 6 different ways as depicted in Figure 3. The editor ensures that only parameters of the same type are mapped, while parameter mappings are edited in the transition dialog. An example is given in Figure 5c .

For many use cases it is possible to compose a complete statechart by combining the capabilities listed above and by using existing states from the state library. Writing any additional source code in C++ is not required in these cases.



(a) Statechart Editor

(b) Input Parameters

(c) Transition Dialog

Figure 5: Dialogs of the Statechart Editor

More complex applications may require implementing custom behavior of states using source code. For these cases the editor offers the option to jump directly into the source code of any state. Additionally, the source code of a state can be viewed in the bottom panel below the graphical editing area (see Figure 5a).

4.2 Linking Implementation and Control Flow

Using a graphical definition for statecharts implies that all parameters, parameter types, parameters mappings, events and transitions are identified via names. Since states are reusable and do not store any information about previous or following states, all states have to share the same basic interface for passing input and output data. We decided to define this interface using string-Variant maps as describe in paragraph 3.3.4. Additionally, the state functions *OnEnter*, *Run*, *OnBreak* and *OnExit* can be implemented in C++. Since C++ is a statically typed language without reflection, accessing an input parameter would look similar to this:

```
float myInput = ((FloatContainer*)getInput("MyValue"))->get();
```

The resulting code overhead to access input parameters and to write output parameters is substantial, if one takes into consideration that not only basic types, but also lists and maps of any data type are supported. Furthermore, the identification of parameters by strings and run time casts can lead to run time errors that could have been detected during compile time. Instead, accessing an input without self written overhead code should look like this:

```
float myInput = getMyValue();
```

To achieve this type safe and auto-completion friendly interface we employ a code generator that generates custom wrapper functions to access inputs and outputs. Inside a generated function the parameter is referenced by name and necessary casts are applied. Since these functions are generated automatically, access by name and the casts will never lead to run time errors. Instead, all possible errors related to parameter accessing occur at compile time. Detecting this kind of errors before executing the statechart saves a lot of time during development.

4.3 Connecting Statecharts and ArmarX Components

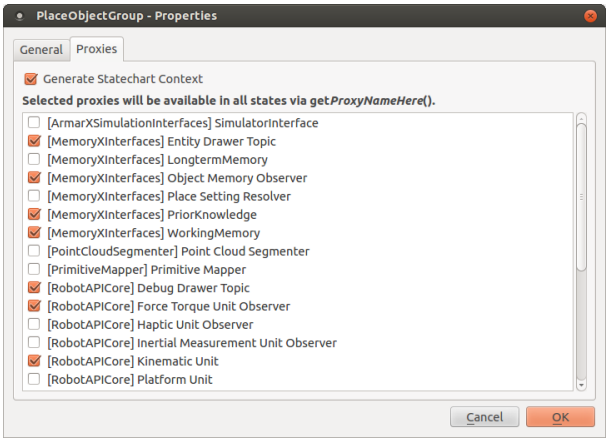
One of the main aspects of statecharts in ArmarX is to interact with components. Since different statecharts for different tasks often require different sets of components, each statechart depends on a set of components. The statechart editor generates a complete list of all available ArmarX components from component meta information. The user can pick any number of components from this list and add them to the dependencies of the statechart as shown in Figure 6a. Every selected component can then be accessed inside the states via a proxy object. Also, additional code is automatically generated so that the statechart registers these dependencies within the ArmarX framework before start-up. Then, the dependency resolver in ArmarX ensures that all necessary components are running before the statechart starts execution.

The list of component proxies for a state can be interpreted as the interface of this state to the ArmarX framework. Similar to object oriented development our goal is to keep these interfaces small. For example, a pick and place statechart requires components to operate the robotic platform, the arms, the hands, do visual servoing, etc. Without encapsulation of proxies this would lead to a very wide interface for high level tasks.

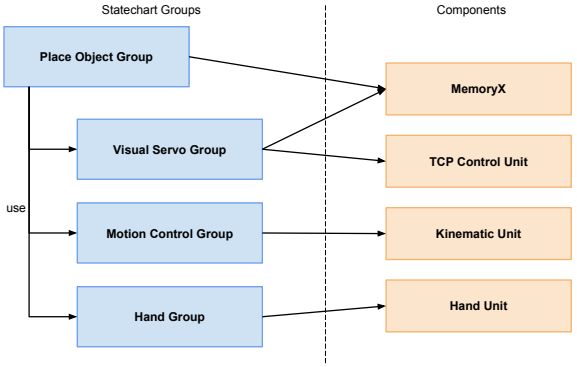
To approach this challenge, we offer a wrapping statechart group for each important component. Each state within a group encapsulates a common task of the encapsulated component. For example, the *HandGroup* offers states to open or close the hands. A high level statechart can then use these wrapper states to indirectly interact with components without the need of a direct dependency on all components. E.g., the *PlaceObjectGroup* needs to control the arms and hands as well as to perform visual servoing to increase accuracy. This demands interaction with the *KinematicUnit*, *HandUnit* and *MemoryX* amongst others. Each of these units is encapsulated by a statechart group, namely the *MotionControlGroup*, *HandGroup* and *VisualServoGroup*. The *PlaceObjectGroup* uses these statechart groups to indirectly interact with the encapsulated components as shown in Figure 6b.

4.4 Statechart Profiles and State Cloning: Reusing Statecharts for different Robots

When developing a new skill for a robot we usually start in simulation. During the transfer of the statechart to the real robot, a lot of parameters usually need to be adapted. For example, when picking up objects from a table, the height of the table might be different in simulation and in reality or the force torque sensor thresholds differ. But these are just differences in parameterization and not on source code level. Thus, our goal is to have the same sourcecode working in simulation as well as on the real robot.



(a) Dialog for selecting components

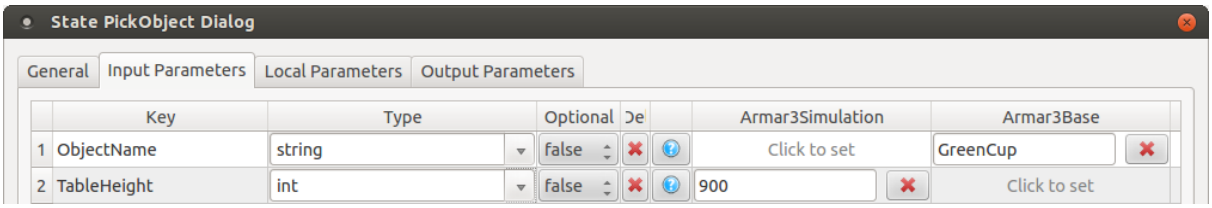


(b) Statechart groups encapsulating components to reduce interface width on high level statecharts

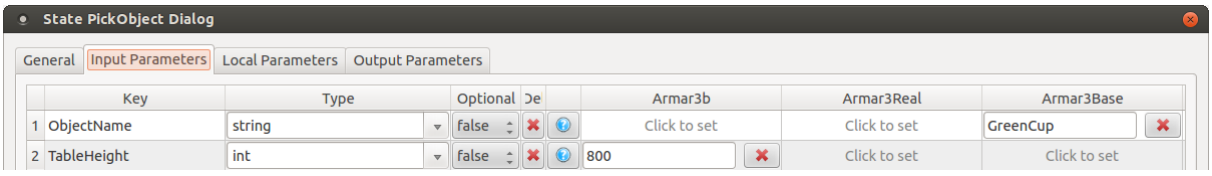
Figure 6: Interaction of Statecharts and Components in ArmarX

To meet this requirement, we introduce the concept of profiles. When working with the statechart editor, the user first selects which profile he or she wants to work with. Every parameter of every state can have specialized values in different profiles, but it is also possible to define default values that apply to multiple profiles if no specialized value is set.

Figure 7 displays the parameter edit dialog for the place object example mentioned above. The parameter *ObjectName* is set to 'GreenCup' and is applied in simulation as well as on the real robot. The parameter *TableHeight* is set to 900mm for simulation and to 800mm for the real robot. The statechart will be executed with the appropriate parameters depending on the selected profile.



(a) Parameters for Simulation



(b) Parameters for the real Robot ARMAR-III B

Figure 7: Statechart Profiles

When reusing statecharts for new robots, simple parameter adjustment is often not sufficient. The underlying behavior implementation of states might require adaptation or the statecharts need to communicate with different components altogether. To cover these cases, the statechart editor offers the option of cloning

complete statecharts, including all sub-states as well as cloning all state-dependencies of the statechart in question. Dependencies are determined by finding all external statecharts that are used in the statechart to be cloned. This process is applied recursively until the list of dependencies is complete. In addition, the statechart editor checks if some of the dependencies have already been cloned for the target robot and omits these states while cloning accordingly. When cloning states, it is possible to apply a prefix to all new states to avoid later confusion. Additionally, all necessary C++ source code files are copied, renamed and modified to match the new names. Statecharts yielded by the cloning process can be compiled and executed without any manual adaptations or amending of source code.

Figure 8 shows an exemplary usecase, in which the statechart group for placing objects (*PlaceObjectGroup*) is cloned to be adapted for the iCub robot. In this example the *HandGroup* has already been cloned previously and has been adapted for the iCub under the name *ICubHandGroup*. The editor recognizes that the *ICubHandGroup* already exists inside the ArmarX iCub package. All newly cloned states that have a dependency to the *HandGroup* will use the adapted *ICubHandGroup* instead of the original.

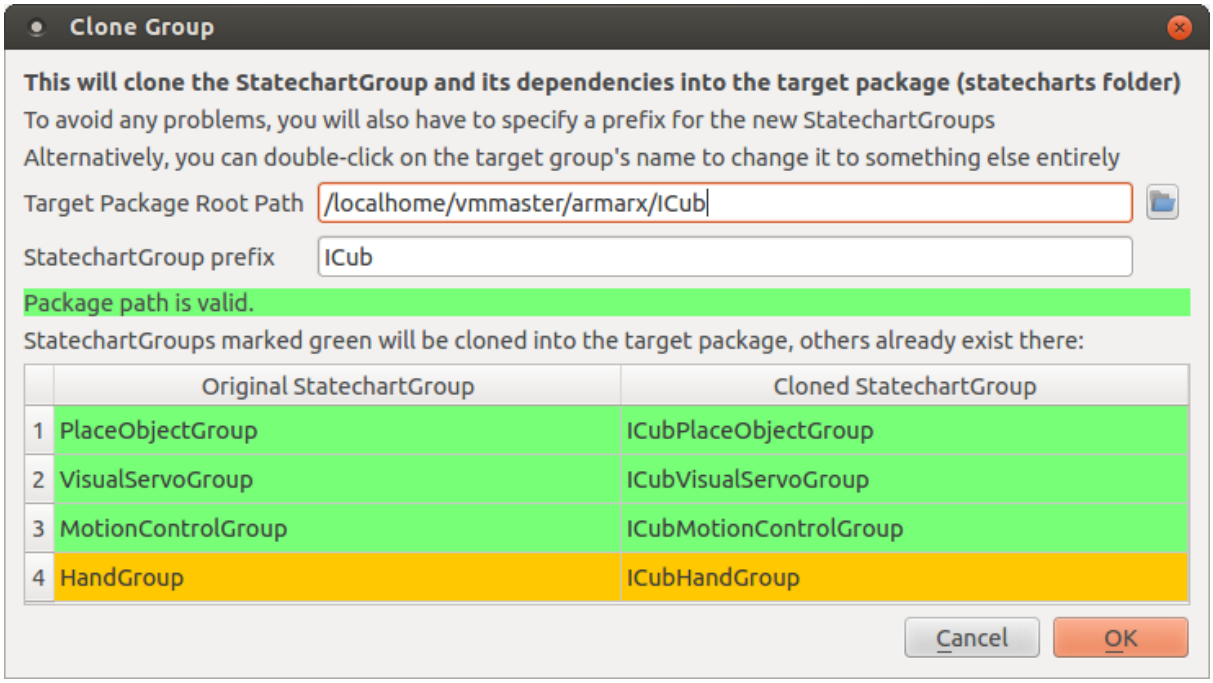


Figure 8. Cloning the *PlaceObjectGroup* for the iCub

4.5 System State Disclosure

Disclosing the state of a robotic system is one of the key features of ArmarX for diagnosing problems at runtime and inspecting the internal state during development. Programmers are able to access data of many parts of the system required for debugging, monitoring and profiling purposes. Different built-in framework mechanisms provide this information, which includes sensor data, conditions, statechart related events, as well as component dependencies and the execution state of statecharts and components. Specialized visualizations are available for presenting and inspecting these different aspects. Textual output is presented as a timestamped log, memory contents is displayed in a 3D view, and a plotter is provided for one dimensional sensor data. Statecharts, their control-flow, and active states are visualized in the StatechartViewer (see Figure 10). Within statecharts, conditions are used to generate events based on sensor data and can be viewed as boolean expression trees as shown in Figure 9.

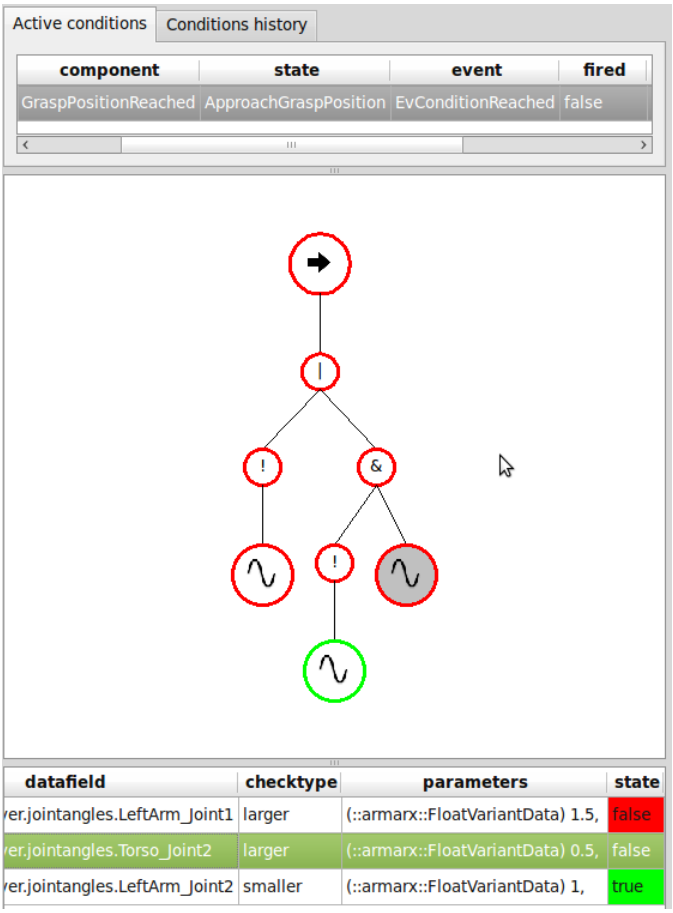


Figure 9. Plugin for visualizing active and already expired conditions. Each subterm of the expression tree is colored green upon fulfillment and red otherwise.

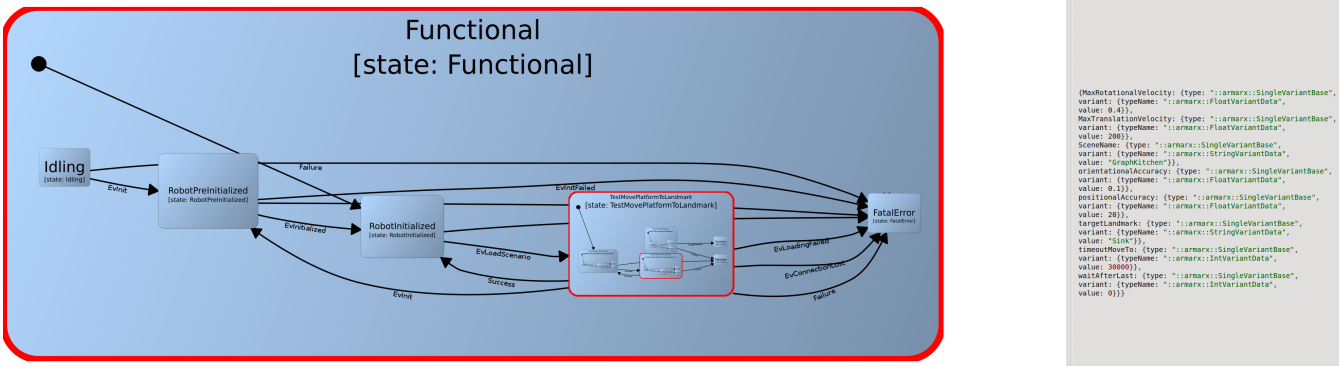


Figure 10. The current state of an executed statechart can be inspected live in the StatechartViewer. The statecharts are layouted on the fly. The red state border signals that this state is active. On the right, current state parameters of the selected state can be examined.

Additionally, ArmarX discloses the system state on a very low level for determining bottlenecks or providing hints for partitioning the distributed application. On the component level, CPU-, memory-, and network utilization data is accessible via the observer mechanism (see Vahrenkamp et al. (2015)) for easy visualization with the graphical plotter. On the statechart level, state transitions and timing information about state durations are available. To enable later processing and evaluation, this low level data can be stored persistently in the memory structure provided by ArmarX.

5 APPLICATIONS AND USE CASES

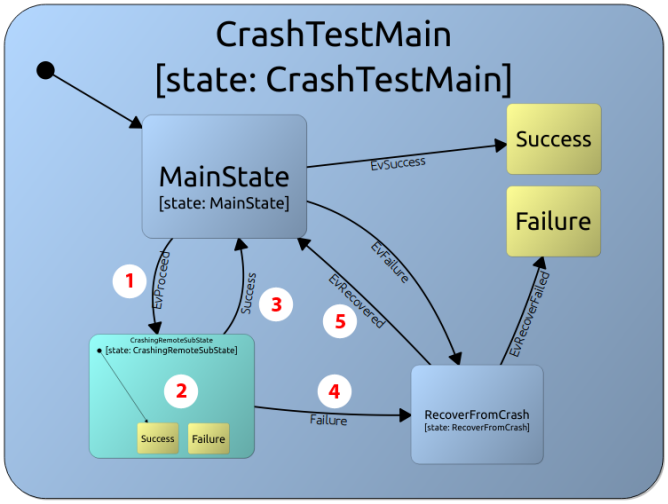
In this section, several applications and use cases realized with ArmarX will be presented which show how distributed statecharts support robustness and provide both convenient usage and flexibility.

5.1 Robustness and Fault Recovery

In this use case, we show how fault recovery concepts are realized within ArmarX. This is important, since most robotics software is written in C++, which allows writing program crashing implementations easily. Hence, a robust robot framework must be able to deal with crashing applications in a way that other components are informed but not affected by a component fault. Further, fault recovery mechanisms should be provided for high and low level robot control.

Several concepts support robustness in ArmarX:

- **Dependency Management**: Due to the distributed nature of ArmarX, crashing components do not affect other components in a non-deterministic way. If component A depends on another component B, the dependency manager of ArmarX only sets A to the state *connected* after B is fully initialized and connected. If component B stops working (i.e. crashes), A is informed and reset to its prior *initialized* state. If the system is capable of restarting B, A will be set to *connected* again.
- **Automatic Restart**: The deployment mechanisms of the distributed Ice middleware can be used to automatically check for running applications. In case an application (an ArmarX component) stopped working, it can be automatically started again.
- **High-Level Fault Recovery**: If an implementation of a robot statechart is erroneous and causes the statechart to crash, the encapsulating statechart is automatically informed that the execution of its sub-state resulted in a failure. Hence, the high-level robot program can consistently handle defective parts in the robot program which could result in a non deterministic behavior of the robot otherwise.




```
void CrashingRemoteState::onEnter()
{
    if (rand()%100 == 0)
    {
        int *i = NULL;
        int j = *i;      // CRASH !!!
    }
}
```

Figure 11. Left: The *CrashTestMain* state encapsulates a remote sub-state which faults from time to time due to a *segmentation fault*. Right: The C++ code of the enter method of the state *CrashingRemoteState* which causes a *segmentation fault* error in a non deterministic way. The error results in an immediate termination of the application that executes the sub-state.

In the following we will show how a crashing sub-statechart can be handled by the robot program. In Figure 11 a statechart is depicted on the left. The execution of the statechart starts with the *MainState*

which emits the event *EvProceed* (1 in Figure 11) causing the execution to pass to the *CrashingRemoteState* statechart (2 in Figure 11). A normal execution would result in a success event (3 in Figure 11), but as shown in Figure 11 on the right, the statechart crashes in a non-deterministic way due to a segmentation fault. Such a segmentation fault results in an immediate termination of the application executing *CrashingRemoteState*. The encapsulating statechart *CrashTestMain* automatically gets informed by the ArmarX runtime system via the *Failure* event (4 in Figure 11) and can recover from this faulty behavior in a deterministic manner (5 in Figure 11).

5.2 Generic Robot Skills

ArmarX provides a library of generic skills, implemented as statecharts, which can be configured and used for a wide variety of robots. The skills cover most basic capabilities needed to setup a robot skill library. In addition to these skills, robot-specific statecharts can be implemented to account for specific features of the platform. The set of generic skills currently provided by the ArmarX framework is listed in Table 1.

Generic skills can be applied to a specific robot by configuring their parameters and by providing robot-specific components on the mid-level of the ArmarX architecture (see Figure 1). Hence, statecharts provide a dependency list of components which must be running before execution is possible. For example, the *ShapeHand* skill needs a *HandUnit* to be running and the skill parameters must specify which shapes are available for execution.

5.2.1 Usecase: Generic skills on different Robots

To show how skills can be applied to different robots, we present a usecase for YouBot Kuka (2015) and ARMAR-4 Asfour et al. (2013), showing the required steps to use the skills *MoveTCP* and *MoveJoints* on different robots.

In general, two steps are needed to program a robot platform with ArmarX. First, a basic set of (robot) components must be configured in order to realize the mid-level structure of the robot software as shown in Figure 1. Second, the initial set of skills has to be configured, defining the basic capabilities the robot programmer can use to build robot applications.

Table 1. Generic set of skills available for use with different robots.

Skill	Description
MoveJoints	Moves joints either in position or velocity control mode
MoveTCP	Moves the tool center point to a Cartesian target
VisualServo	Implements a position-based visual servo approach
MovePlatform	Moves a platform-based robot along a graph or to a specific point
LookTo	Centers a Cartesian position with the head
GraspObject	Picks up an object with an end effector
BringObject	Picks up an object and delivers it to a specified location
ZeroForce	Enables zero force control for an end effector
StopRobot	Stops all movements
PlaceObject	Puts down a grasped object
ScanForObject	Applies a scanning strategy to search for an object
TrackObject	Tries to track an object
ViewSelection	Changes view direction, according to an automatic attention mechanism
Open/Close/Shape Hand	Move hand to specific shapes

a) Robot Components
Initially, several components must be realized for the different robots. Beforehand, the robot's visualization, kinematics and dynamics properties must be defined. In ArmarX, these properties are specified with the Simox Vahrenkamp et al. (2012) robot file format. The minimal set of components needed for the *MoveJoints* and *MoveTCP* skills is listed below:

- *KinematicUnit*: Encapsulates access on joint level. In the following examples, the robots are simulated with kinematic simulation units provided by ArmarX. On a real robot, this component is connected to the robot's hardware layer. In case of ARMAR-4, the *KinematicUnit* connects to the ArmarX-RT layer to communicate with the motors and sensors Vahrenkamp et al. (2014).
- *KinematicUnitObserver*: Observes the raw joint data in order to trigger events.
- *RobotStateComponent*: A network transparent representation of the robot used for forward and inverse kinematics.
- *TCPControlUnit*: Allows control of the tool center point (TCP) in cartesian space.

Access to the real robot (i.e. to the drivers) needs to be implemented via the *KinematicUnit* component, while all these components are already available in simulation and can be configured for use with a new robot. Hence, a basic framework can be quickly realized by configuring provided ready-to-use components of ArmarX.

b) Robot Skills
Once all components are set up for the specific robot, high-level robot program can be implemented. As a starting point, several skills can be taken from the ArmarX skill template library and configured to be used on the robot. In this example, the *MoveTCP* and *MoveJoints* skills are used and a waving statechart is programmed via the *Statechart Editor* tool. As shown in Figure 12 and Figure 13, the realization can take advantage of the ready-to-use skill library of ArmarX on such different robots as ARMAR-4 and YouBot. In addition, the waving statechart that is used in Figure 13 at the top can be directly executed on the real ARMAR-4 as shown in Figure 13 at the bottom. Such a skill transfer for the complex reactive grasping skills (similar to 5.3) is also shown in the work presented by Paikan et al. (2015).

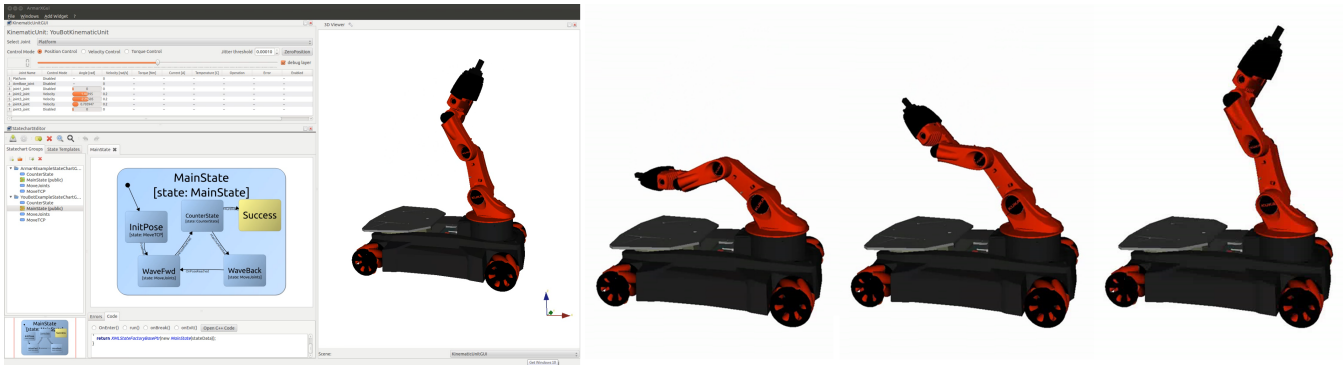


Figure 12. The waving statechart executed on YouBot, while running a kinematic simulation.

5.3 Reactive Grasping of Unknown Objects

In the context of the Xperience (2011) Project, we developed a statechart and extended accompanying components to perform Reactive Grasping based on vision and haptics on the humanoid robot ARMAR-III Asfour et al. (2006). This use case demonstrates reusability of ArmarX statecharts through extension

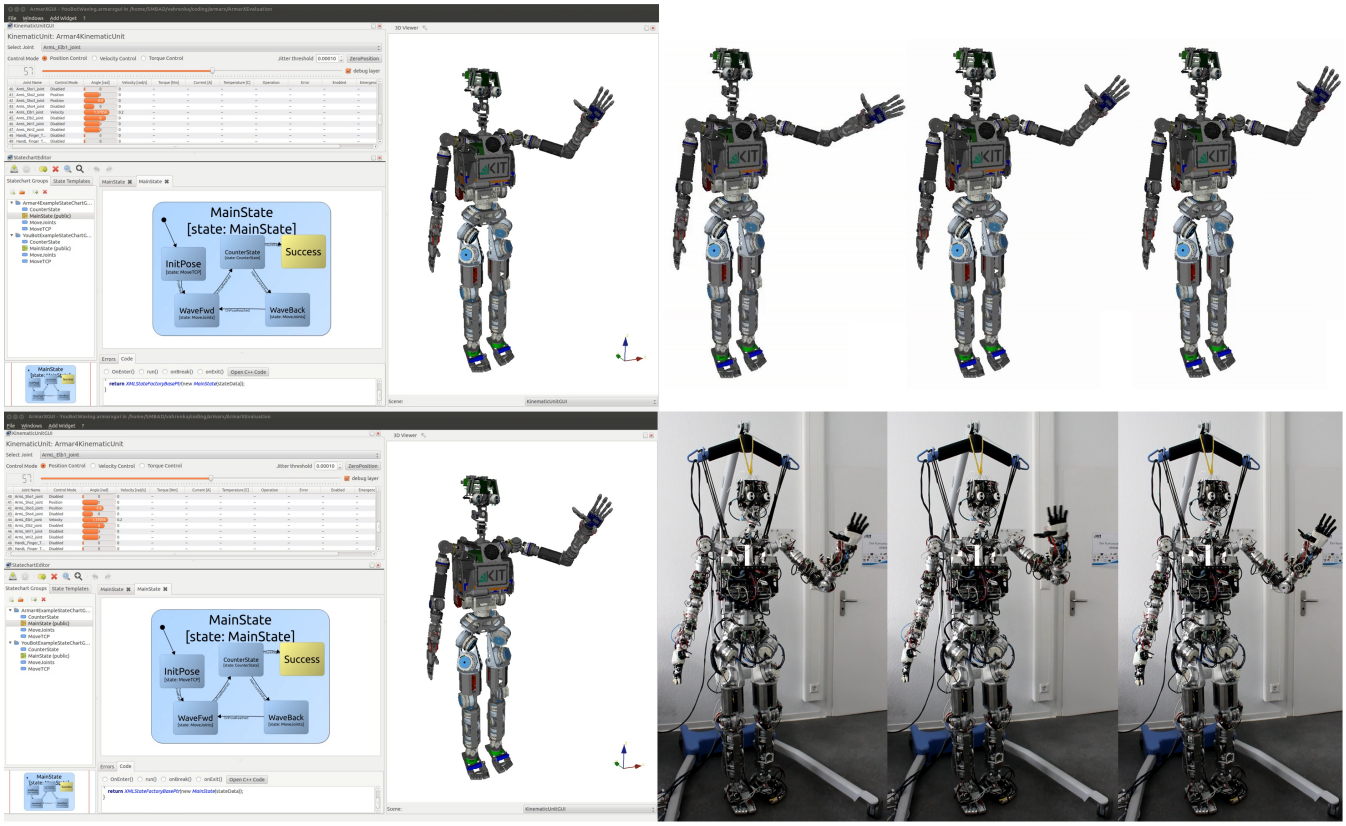


Figure 13. ARMAR-4 executing a waving motion with the same statechart in simulation and on the real robot.

of the programmed behavior and the incorporation of sensor feedback on different hierarchy levels. The approach presented in Schiebener et al. (2011) was used to initially learn an object hypothesis and pose. The pose as well as the forward kinematics are not perfectly exact. Therefore, correcting actions during grasping are necessary. Guidance of the hand during the grasping approach phase is based on visual servo. To accommodate for inaccuracies, we need an extended visual servoing that reacts on collisions of the hand with the object. Instead of implementing a specialized version of visual servoing we created a wrapping statechart called *Visual Servo with Collision Detection*, which is used in our Reactive Grasping (see Figure 14).

In parallel to the statechart execution, three different collision detection components are running to detect visual collisions, tactile collisions and collisions inferred from proprioceptive data. These components run independently, monitor different sensors of the robot, and offer event notifications usable in statecharts. The wrapping statechart *Visual Servo with Collision Detection* monitors the output of the collision detection components by installing conditions with given thresholds on the output data. Then, the visual servoing statechart is started as a sub-state. If any of the conditions is met during servoing the appropriate event is fired. The wrapping state *Visual Servo with Collision Detection* is exited and the execution of all sub-states is stopped. Hereby, the visual servoing is interrupted and the collision can be handled appropriately by correcting the grasp pose. After correcting the pose, the statechart transitions back to the extended visual servoing.

By wrapping the visual servoing skill in a statechart, we can reuse and extend the visual servoing without modifying it. The used visual servoing skill is the standard visual servoing from the ArmarX statechart library.

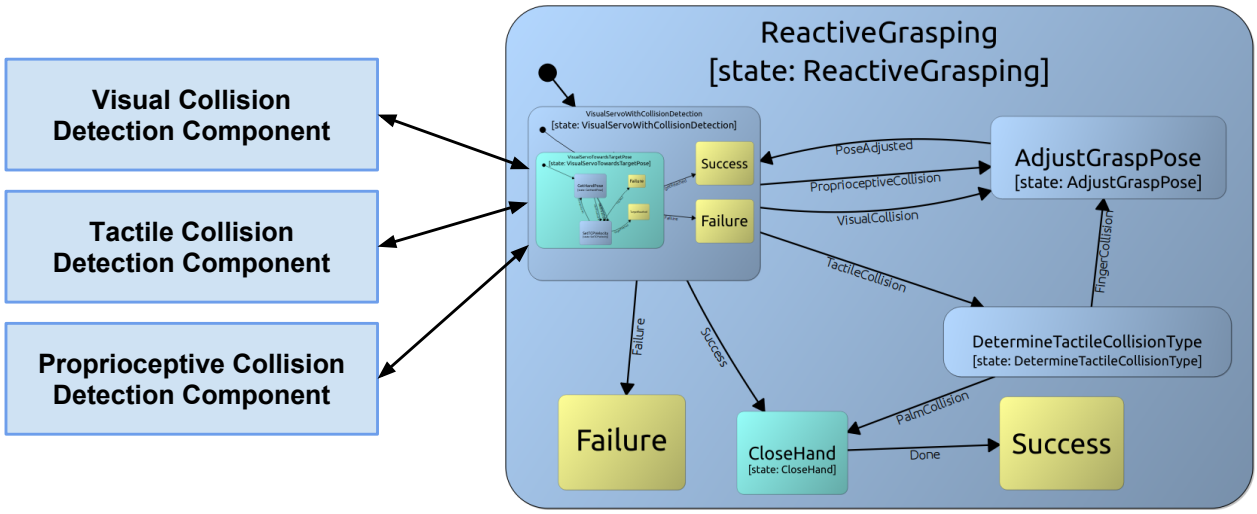


Figure 14. Simplified statechart for Reactive Grasping in the Xperience project

5.4 Dynamic State Replacement

One use case for the dynamic state replacement feature of ArmarX is the combination of a symbolic task planning system with ArmarX statecharts for execution. To connect the planning system to statecharts, a control statechart, as shown in Figure 15, was build around one *DynamicRemoteState* (depicted in violet). Since statecharts do not offer an interface for remote procedure calls, it is not possible to communicate with states directly. States react on external changes by observing changes in datafields. Thus, we inserted an additional component, the plan step observer, on which the statechart can install conditions to receive an event (*EvNextStepPlanned*) on changes related to the current planning step. The planning system manages this datafield, containing the current action and its parameters. After the event was received, the desired skill statechart is loaded into the *DynamicRemoteState* and is directly executed. With this powerful mechanism it is possible to implement interactive and dynamic robotic applications in a consistent and robust way.

6 CONCLUSION

We presented the statechart concept of the robot development environment ArmarX and showed how high level robot programming can be realized in a robust and convenient way. The event-driven statechart approach of ArmarX helps realizing important features such as increased robustness through distributed program execution, convenient programming through graphical user interfaces, and versatility by interweaving dynamic statechart structure with custom user-code. These features build a solid base for higher-level robot program implementations which is accompanied by advanced framework capabilities such as reusable robot programs and the presented ability to transfer skills to different robots.

In future work, we will improve the framework in terms of high-level robot program development, validation and debugging. Therefore we will introduce orthogonality into the statechart concept to enable parallel statechart structures. Currently, parallel execution is only supported between hierarchy levels, but there are applications in which orthogonal skill execution eases the design of the high level robot program. In addition, we will work on automatic parameter and statechart validation in order to reduce potential faults in robot programming and to speed up the development process. Furthermore, we plan to offer debugging features on statechart level such as break points in statecharts, which will reduce the time spent for development and debugging.

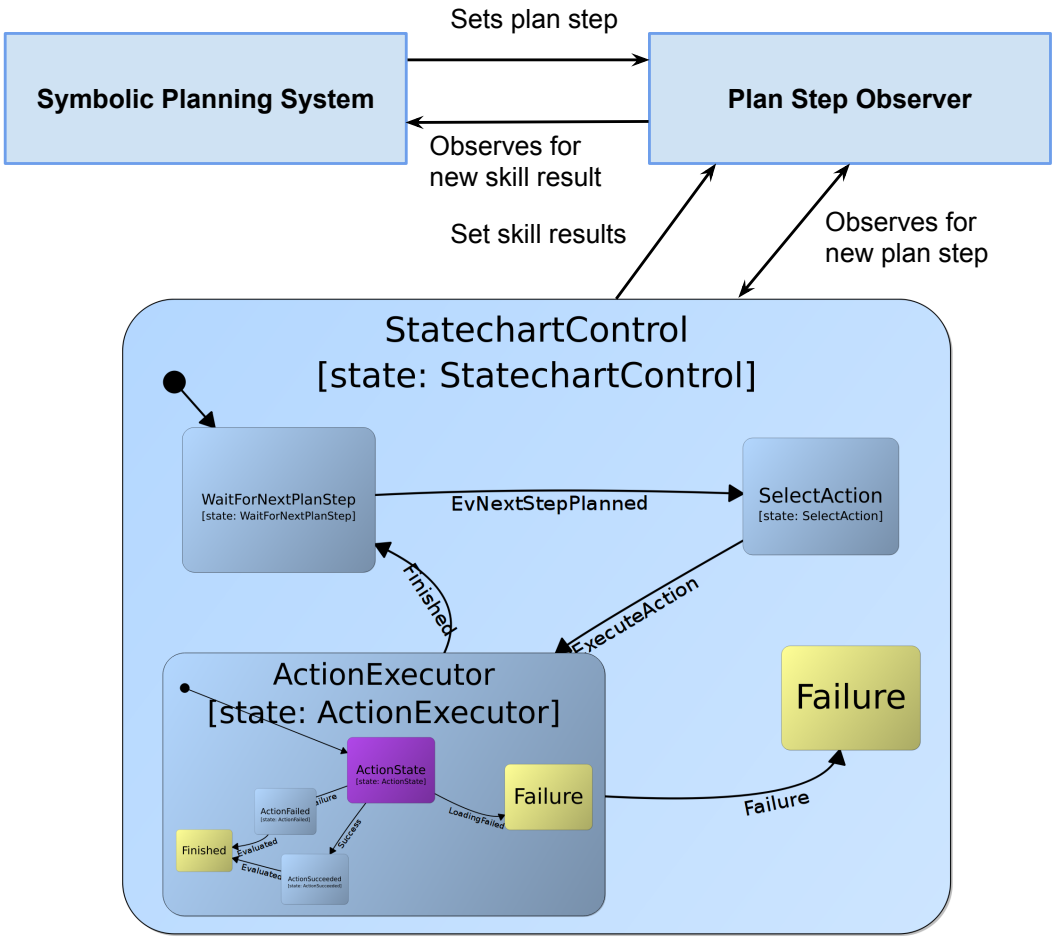


Figure 15. Planning statechart with a *DynamicRemoteState* (violet) that can be changed at runtime.

7 ACKNOWLEDGEMENT

The research leading to these results has received funding from the European Union's Seventh Framework Programme under grant agreement no. 270273 (Xperience) and from European Union's Horizon 2020 Research and Innovation Programme under grant agreement no. 643950 (SecondHands).
The authors would like to thank all members and students of the Humanoids group at KIT for their various contributions to this work.

REFERENCES

AIST (2015). RtcLink. `http://openrtm.org/openrtm/en/content/rtclink-0`. Accessed: 2015-09-18

Ando, N., Suehiro, T., and Kotoku, T. (2008). A software platform for component based rt-system development: Openrtm-aist. In *Proceedings of the 1st International Conference on Simulation, Modeling, and Programming for Autonomous Robots* (Berlin, Heidelberg: Springer-Verlag), SIMPAR '08, 87–98. doi:10.1007/978-3-540-89076-8_12

Angermann, A., Beuschel, M., Rau, M., and Wohlfarth, U. (2014). *MATLAB-Simulink-Stateflow: Grundlagen, Toolboxen, Beispiele* (Walter de Gruyter)

Asfour, T., Regenstein, K., Azad, P., Schröder, J., Vahrenkamp, N., and Dillmann, R. (2006). ARMAR-III: An Integrated Humanoid Platform for Sensory-Motor Control. In *IEEE/RAS International Conference on Humanoid Robots (Humanoids)*. 169–175

Asfour, T., Schill, J., Peters, H., Klas, C., Bücker, J., Sander, C., et al. (2013). Armar-4: A 63 dof torque controlled humanoid robot. In *IEEE/RAS International Conference on Humanoid Robots (Humanoids)*. 390–396

Billington, D., Estivill-Castro, V., Hexel, R., and Rock, A. (2010). Modelling behaviour requirements for automatic interpretation, simulation and deployment. In *SIMPAR* (Springer), vol. 6472 of *Lecture Notes in Computer Science*, 204–216

Bohren, J. and Cousins, S. (2010). The SMACH High-Level Executive [ROS News]. *Robotics Automation Magazine, IEEE* 17, 18–20

Bruyninckx, H., Soetens, P., and Koninckx, B. (2003). The real-time motion control core of the Orocos project. In *IEEE International Conference on Robotics and Automation (ICRA)*. 2766–2771

Calisi, D., Censi, A., Iocchi, L., and Nardi, D. (2008). Openrdk: A modular framework for robotic software development. In *Intelligent Robots and Systems, 2008. IROS 2008. IEEE/RSJ International Conference on*. 1872–1877

Coleman, D., Hayes, F., and Bear, S. (1992). Introducing objectcharts or how to use statecharts in object-oriented design. *Software Engineering, IEEE Transactions on* 18, 8–18

CORBA, O. M. G. (2006). *CORBA Component Model 4.0 Specification*. Specification Version 4.0, Object Management Group

EasyCODE (2015). EasyCODE. `http://www.easycode.de`. Accessed: 2015-08-15

Finkemeyer, B., Kröger, T., Kubus, D., Olschewski, M., and Wahl, F. M. (2007). MiRPA: Middleware for robotic and process control applications. In *Workshop on Measures and Procedures for the Evaluation of Robot Architectures and Middleware at the IEEE/RSJ International Conference on Intellegent Robots and Systems* (San Diego, CA, USA), 78–93

Gill, A. et al. (1962). Introduction to the theory of finite-state machines

Harel, D. (1987). Statecharts: A visual formalism for complex systems. *Science of computer programming* 8, 231–274

Henning, M. (2004). A new approach to object-oriented middleware. *Internet Computing, IEEE* 8, 66–75. doi:10.1109/MIC.2004.1260706

Hirzinger, G. and Bauml, B. (2006). Agile robot development (ard): A pragmatic approach to robotic software. In *Intelligent Robots and Systems, 2006 IEEE/RSJ International Conference on* (IEEE), 3741–3748

Huber, A. (2007). boost statechart library. `http://www.boost.org`. Accessed: 2015-08-15

Klotzbücher, M. and Bruyninckx, H. (2012). Coordinating Robotic Tasks and Systems with rFSM Statecharts. *JOSER: Journal of Software Engineering for Robotics* , 28–56

Kuka (2015). YouBot Webpage. `http://www.youbot-store.com/`. [Online; accessed 19-August-2015]

MathWorks (2015a). MATLAB. `http://www.mathworks.com/products/matlab/`. Accessed: 2015-08-18

MathWorks (2015b). Simulink. `http://www.mathworks.com/products/simulink/`. Accessed: 2015-08-18

MathWorks (2015c). Stateflow. `http://www.mathworks.com/products/stateflow/`. Accessed: 2015-08-18

Merz, T., Rudol, P., and Wzorek, M. (2006). Control system framework for autonomous robots based on extended state machines. In *ICAS* (IEEE Computer Society), 14

Metta, G., Fitzpatrick, P., and Natale, L. (2006). YARP: Yet another robot platform. *International Journal on Advanced Robotics Systems* 43–48

Metta, G., Sandini, G., Vernon, D., Natale, L., and Nori, F. (2008). The icub humanoid robot: An open platform for research in embodied cognition. In *Proceedings of the 8th Workshop on Performance Metrics for Intelligent Systems* (New York, NY, USA: ACM), PerMIS '08, 50–56. doi:10.1145/1774674.1774683

Microsoft (2012a). Robotics Developer Studio. `https://msdn.microsoft.com/en-us/library/bb648760.aspx`. Accessed: 2015-08-18

Microsoft (2012b). Visual Programming Language. `https://msdn.microsoft.com/en-us/library/bb483088.aspx`. Accessed: 2015-08-18

Newman, P. M. (2008). Moos-mission orientated operating suite. *Massachusetts Institute of Technology, Tech. Rep* 2299

Nilsson, N. J. and Center, A. I. (1973). *A hierarchical robot planning and execution system* (Stanford Research Institute)

Paikan, A. (2014). *Enhancing Software Module Reusability and Development in Robotic Applications*. dissertation, Istituto Italiano di Tecnologia

Paikan, A., Schiebener, D., Wächter, M., Asfour, T., Metta, G., and Natale, L. (2015). Transferring object grasping knowledge and skill across different robotic platforms. In *International Conference on Advanced Robotics (ICAR)*. 498–503

Pot, E., Monceaux, J., Gelin, R., and Maisonnier, B. (2009). Choregraphe: a graphical tool for humanoid robot programming. In *Robot and Human Interactive Communication, 2009. RO-MAN 2009. The 18th IEEE International Symposium on* (IEEE), 46–51

Quantum Leaps (2015). QM statemachines. `http://www.state-machine.com`. Accessed: 2015-08-15

Quigley, M., Conley, K., Gerkey, B. P., Faust, J., Foote, T., Leibs, J., et al. (2009). ROS: an open-source Robot Operating System. In *ICRA Workshop on Open Source Software*. vol. 3, 5

Rahul, R., Whitchurch, A., and Rao, M. (2014). An open source graphical robot programming environment in introductory programming curriculum for undergraduates. In *MOOC, Innovation and Technology in Education (MITE), 2014 IEEE International Conference on*. 96–100. doi:10.1109/MITE.2014.7020248

Samek, M. (2002). *Practical statecharts in C/C++: Quantum programming for embedded systems* (CRC Press)

Schiebener, D., Ude, A., Morimotot, J., Asfour, T., and Dillmann, R. (2011). Segmentation and learning of unknown objects through physical interaction. In *Humanoid Robots (Humanoids), 2011 11th IEEE-RAS International Conference on* (IEEE), 500–506

Scholl, K.-U., Albiez, J., and Gassmann, B. (2001). Mca-an expandable modular controller architecture. In *3rd Real-Time Linux Workshop*

Vahrenkamp, N., Kröhnert, M., Ulbrich, S., Asfour, T., Metta, G., Dillmann, R., et al. (2012). Simox: A robotics toolbox for simulation, motion and grasp planning. In *International Conference on Intelligent Autonomous Systems (IAS)*. 585–594

Vahrenkamp, N., Wächter, M., Kröhnert, M., Kaiser, P., Welke, K., and Asfour, T. (2014). High-level robot control with armarx. In *INFORMATIK Workshop on Robot Control Architectures*. 1–12

Vahrenkamp, N., Wächter, M., Kröhnert, M., Welke, K., and Asfour, T. (2015). The ArmarX framework - supporting high level robot programming through state disclosure. *Information Technology* 57, 99–111

Von der Beeck, M. (1994). A comparison of statecharts variants. In *Formal techniques in real-time and fault-tolerant systems* (Springer), 128–148

Xperience (2011). The Xperience Project. `http://www.xperience.org`. Accessed: 2015-08-15

Yakindu (2015). Yakindu Statechart Editor Tools. `http://www.yakindu.org`. Accessed: 2015-08-15