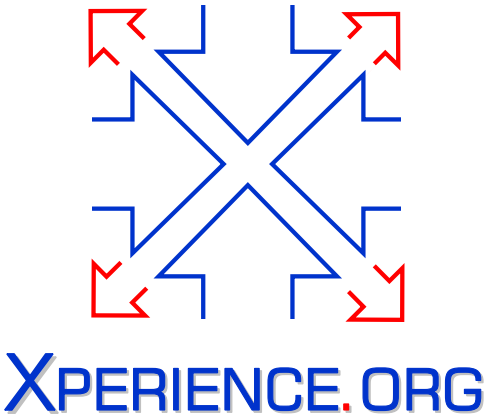
Project Acronym:	Xperience
Project Type:	IP
Project Title:	Robots Bootstrapped through Learning from Experience
Contract Number:	270273
Starting Date:	01-01-2011
Ending Date:	31-12-2015



Deliverable Number:	D1.2.1
Deliverable Title:	Blueprint for the Cognitive Architecture
Type (Internal, Restricted, Public):	PU
Authors:	G. Metta, T. Asfour, A. Ude, J. Piater
Contributing Partners:	ALL

Contractual Date of Delivery to the EC:	30.07.2013
Actual Date of Delivery to the EC:	07.04.2014

Contents

Chapter 1

Executive Summary

This deliverable describes the blueprint of the cognitive architecture of Xperience. In particular, it presents the first validated implementation of the three layers of the architecture (the three layers are described already in D1.1.1). The approach and philosophical basis of the architecture are discussed at length in D1.1.1 as derived in part from the RobotCub and Paco+ projects. In this new deliverable we have progressed by proposing an actual implementation of the three layers albeit each layer at the moment is deployed as a separate software library. The remainder of the project will focus in delivery a fully integrated software system.

The blueprint of the architecture derives from the general "paper" architecture of figure 1.1. It is a rather standard three-layer architecture, which can be grossly named: i) behavioral layer, ii) OAC layer, iii) planning layer.

The behavioral layer is modeled after a subsumption-type architecture and the current implementation is based on a modified version of the YARP middleware. In particular, a plug-in system has been developed (reported in Y2) and extended to run Lua scripts - which can be modified on the fly without recompiling the code of the various modules. An example is presented later to show the actual use of the scripts in implementing one of the scenarios (table cleaning) proposed as demonstrations for Y3.

The OAC layer is based on the Armar robot. It implements a state disclosure model which makes the complete robot state always inspectable programmatically on a number of interfaces. More importantly, the Armar architecture includes a statechart model which is shown to be used to implement OACs (the middle layer in figure 1.1).

Finally, the OAC descriptions are used in the upper layer (planning). We designed a STRIP planner (see D3.2.3) called PKS which builds plans at the knowledge level, by representing and reasoning about how the planner's knowledge stte changes during plan generation. Latest additions improve the ability of the planner to model uncertain numerical information (e.g. noisy sensors, effectors, etc.) and a standard interface (API) which supports the modification of planning domains at run time.

The three layers partially overlap since as we shall see (WP5), there are various instances where a given inference can be implemented equally well in two layers. For example, tool use can be either part of an OAC or a set of behaviors. In the former, the knowledge about the use of the tool is explicit, requiring an inference engine to work on the OACs. The latter instead implements an implicit affordance resolution by coordinating various behaviors to achieve the same goal. Clearly, the OACs are more general and powerful to represent complex affordances.

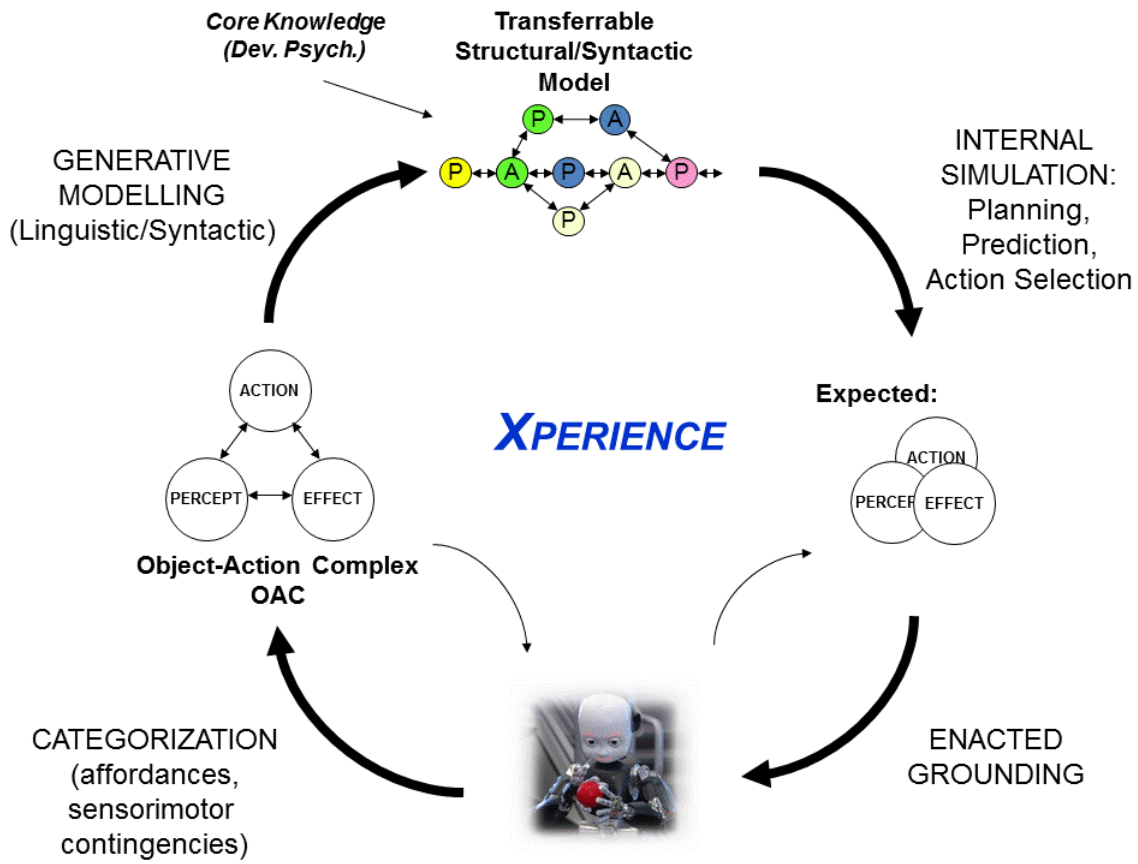


Figure 1.1: The three-layer cognitive architecture presented in Xperience as part of D1.1.1: behavior based layer, middle level and planing level

Similarly, plans can be constructed simply by concatenating OACs or rather by invoking PKS. Also, the world model for PKS can rely to different degrees to the OAC representation or as an alternative it can access the sensors and effectors level. A computational performance criterion may be used to choose beforehand what representation to use. This is left to be explored in the remainder of the Xperience project.

1.1 A comment on biological plausibility

We have not commented so far on the biological plausibility of the current architecture implementation. Several examples exist that link human development, language acquisition, and the very concept of structural bootstrapping to our work.

Previously, we reported for example about the development of grasping in human infants as in the work of Guerin et al [GKK13] (see D2.3.1 and D3.1.1). Another example of how our research is rooted in child development is reported in D4.2.2. In that case [KGZS12a] we model the acquisition of syntax and semantics from a corpus of child-directed utterances paired with possible representations of their meanings. Finally, development is present is our work on motor action acquisition as e.g. in some of our work on incremental learning where we investigate the problem of learning over a developmental timescale, i.e. how can we learn efficiently on ever increasing number of data points [GM12].

In summary, while the overall architecture does not reproduce either child development or a brain-like organization, many elements are of biological inspiration/derivation. The final report on the architecture due at the end of the project will expand on the question of biological plausibility or relevance of this work.

Chapter 2

Behavior Based Layer

The robotics software community is continuing to grow. Within the community, researchers have been developing large number of software components using some of the most common robotic middleware, such as ROS [QGC+09], YARP [GPL07], OROCOS [Bru01], OPROS [JLJ+10] and Open-RTM [ASK08] or based on their customized frameworks using standard communication libraries (e.g., CORBA [OMG08], ICE [Inc], ØMQ [zer]). They try to adopt lessons learned from best practices in robotics [BS09, BS10] and software architecture techniques and standards [Sam97] to build their modules as reusable as possible. Even so, it is quite unlikely that components from different communities fit into a specific off–the–shelf deployment scenario, without any adaptation by third party users. Heterogeneity and lacking of standards are not the only bottlenecks burdening reusability. Even within a community of developers who share the same middleware, software components can be developed with different taste and still hard to reuse. Systematically developing high–quality reusable software component is, indeed, a difficult task. Many developers keep their modules simple. However, simplicity does not necessarily lead to more reusable software. On the other hand, with reusability in mind, there is a risk of over–generalization and increased complexity: to build a more generic and reusable component, the developer tries to foresee all possible future needs and add them as reconfigurable functionalities to the software. Such a commitment leads to complex components, polluted with application–dependent functionalities that are more costly and difficult to maintain and use correctly. Thus, a proper balance must be found between potential reuse and ease of implementation [Sam97].

Software should be extensible enough to be adapted to possibly unanticipated changes [Zen04]. Extensibility is an important property for software which significantly boosts reusability. One direction to extend a module is via its interfaces. In distributed systems interfaces are implemented by exchanging messages through special connection points that are call ports. This plays an important role in nowadays robotic software architectures. This paper concentrates on enhancing robotic software module's reusability by extending its port's functionality using a scripting language. The basic idea is to extend the port's functionalities in order to dynamically load a run–time script and plug it into the port of an existing module without changing the code or recompiling it. In our framework a port extension is called *Port Monitor*: in brief it allows to access the data passing though a connection from/to the port for monitoring, filtering and transforming it (See Section 2.2). Multiple port monitors can interact to allow an input port to select data from multiple sources in an exclusive way. We call this object a *Port Arbitrator*: in other words, a port arbitrator allows a module to arbitrate data coming from other components to its input port and coordinate the corresponding modules (see Section 2.3).

Section 2.1 gives an overview of similar approaches from the literature. The detail of the Port Monitor and an overview of its API is described in Section 2.2. Section 2.3 represents the Port Arbitrator. The applicability of this approach is demonstrated in Section 2.4 through a step–by-step example using the iCub robot's software repository (same of D5.2.3).

2.1 Related Work

Plug-in platforms, in general, extend a core system with new features implemented as components that are plugged into the core at run time and integrate seamlessly with it. When an application supports plug-ins, it enables customization, thus, provides a promising approach for building software systems which are extensible and customizable to the particular needs [WD06]. Probably one of the more prominent example of a platform which broadly supports plug-ins is Eclipse IDE [GB04]. Eclipse offers a framework to develop plug-ins in Java which are delivered as JAR libraries. There are also some generic frameworks for plug-in development and management such as Pluma [plu] which allows loading plug-ins as dynamic linked libraries or FxEngine [fxe] for data flow processing and the design of dynamic systems. plug-ins can also be developed using scripting languages. Scripting languages have been used for decades to extend the functionality offered by software components and they have special interests within the game developer communities. The main advantage of script–based plug-ins is that they are usually easier to be developed and maintained.

Despite plug-in system has been broadly used by software developers over the last decades, to our knowledge, less attention has been devoted to study their potentials in robotics. Our work is an approach to extend a YARP component's port as it can act as a plug-in manager to load plug-ins written in a scripting language [1]. The current implementation offers a plug-in development using Lua but it can be easily extended to support other languages.

2.2 Port Monitor

To better illustrate the concept of the Port Monitor, we can consider, as an example, an application for tracking faces in a humanoid robot as shown in Figure 2.1. The application involves two simple modules. The first one is a Face–Detector module which receives image data from the robot's camera, detects human faces and streams out, through its output port, the 3D position of the detected face together with a confidence level. The second module is called Head–Control; which in its turn receives a 3D position and controls the head of the robot to look at the corresponding point. A simple head tracking application can be achieved by connecting the output of the Face–Detector to the input of the Head–Control. Now suppose we want to extend the application and track the face only if the confidence level is above a certain threshold. This can be achieved by modifying the Head–Control module to take into account the extra information, but it would then be inadvertently polluted with application dependent functionality. A more appropriate choice would be to develop a third module which receives data from the Face–Detector and filters out messages corresponding to detections that do not satisfy the required confidence level. The drawback of this approach is that it introduces extra implementation effort whilst adding further communication and deployment overhead to the system.

[1] The source code and relevant examples can be found at https://github.com/robotology/yarp/tree/master/src/carriers/

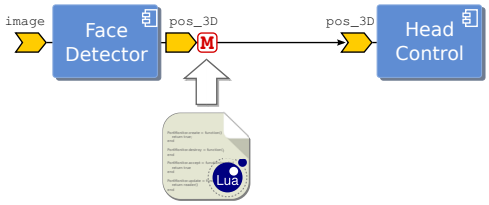


Figure 2.1: Conceptual representation of port monitor. The output port of Face–Detector modules is extended with a plug-in which provides access to the outgoing data through scripting language (e.g., Lua).

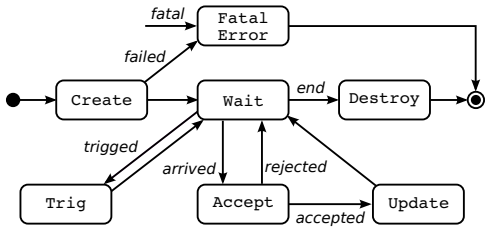


Figure 2.2: The life cycle of port monitor.

One could argue over the immaturity of the involved modules and proposes that, for example, the Face–Detector module could be improved and reimplemented so that it can be reconfigured by specifying the desired confidence level. This solution requires higher development cost and in general it easily leads to complex design. In addition, it makes it even more arduous to connect the Face–Detector to multiple modules with different confidence requirements, thus preventing runtime reusability.

In our approach it is possible to solve this problem by extending the ports of components using run–time scripts. Figure 2.1 represents the concept of the Port Monitor (shown as a box with M) attached to the output of the Face–Detector module. The Port Monitor can load a script file (written using a standard scripting language such as, in our case, Lua [IDC96]) and can access and modify the data traveling through the port using a simple API. Thus, some extra functionalities of a component such as data filtering, transformation, monitoring can be added during the application development time and without the need to modify and rebuild the component itself. Similarly the same script can be loaded by the input port of the Head–Control module.

This mechanisms offers the following advantages. Firstly, it avoids adding to the component application specific functionalities. That is, some application–dependent functionalities can be freely added to the component using the scripting language during the application development stage. Secondly, it allows to simplify the implementation of the components, since the developer does not necessarily need to provide all possible configurations supporting different application scenarios. Finally, by embedding the extra functionalities inside the port, our approach intrinsically reduces communication and deployment overhead that would be introduced if the same functionalities were added as separate modules.

2.2.1 Port monitor life cycle and API

Figure 2.2 illustrates the states that define the life cycle of a port monitor. A callback function is assigned to each state (except Waiting) which can have a corresponding implementation in the user's script. Using these callbacks, users have full control over the port's data and can access it, modify it and decide whether to accept the data or discard it. Listing 2.1 represents the callback functions corresponding to the port monitor's states in Lua.

The Port Monitor starts in the *Create* state in which `PortMonitor.create` callback is called. The initialization of the user's code can be done at this point. Returning a true value means that the user's initialization was successful and the monitor object is able to start monitoring data from the port. When data arrives to the monitor, `PortMonitor.accept` is called. In this callback, user can access (for reading purposes only) the data, check it and decide whether to accept or discard it. The return value of this function indicates whether the data should be delivered (accepted) or discarded. If the data is accepted, `PortMonitor.update` is called, at which point the user has access to *modify* the data.

```
PortMonitor.create = function() return true end
PortMonitor.accept = function(dt) return true end
PortMonitor.update = function(dt) return dt end
PortMonitor.trig = function() return end
PortMonitor.destroy = function() end
```

Listing 2.1: Port monitor callback functions in Lua

A port monitor will usually act as a passive object [Mur95] where accept and update callbacks are called only upon data reception. However, one may need to periodically monitor a connection (within a specific time interval) and, for example, generate proper events in the case of delay in the communication. For this purpose, a port monitor object can be configured to call `PortMonitor.trig` within desired time intervals. Finally, `PortMonitor.destroy` is called when the port monitor is detached from the port upon disconnection. As an example listing 2.2 illustrates the pseudo–script in Lua that in the hypotetical application that requires filtering out messages from Face–Detector when the confidence level is below a threshold of 80%.

```
1 PortMonitor.accept = function(data)
2    -- read face_pos from 'data'
3    if face_pos.certainty < 0.8 then
4       return false
5    end
6    return true
7 end
```

Listing 2.2: An example of filtering Face–Detector data.

2.3 Port Arbitrator

A port Arbitrator is an extended functionality of an *input* port which can be configured to arbitrate data from multiple source based on some user–defined constraints. Figure 2.3 represents a simple search–and–track application where a humanoid robot looks around in search of a person's face and tracks it. The robot should look around only if it is not tracking a face. The application involves modules described in the face–tracking example from Section 2.2 and an

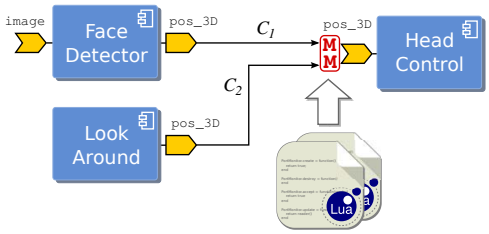


Figure 2.3: Conceptual representation of port arbitrator

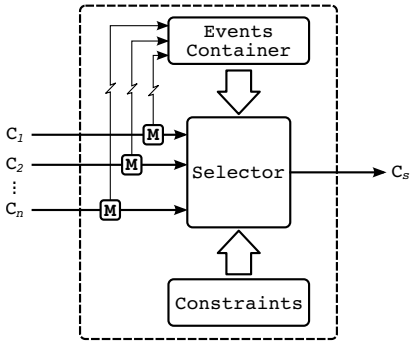


Figure 2.4: The architecture of port arbitrator. Straight lines show the data flow and zigzag lines represent event flows.

extra module to look around. Look–Around generates some random 3D position which makes the robot randomly look around when it is provided to the Head–Control module. To get the desired behavior, the Head–Control module should **not** receive data from the Look–Around module when the Face detector module is sending detected face positions.

This is a common coordination problem which can be solved in different ways (e.g., using a separate coordinator, extending modules to interact with each other). One way to achieve this is to use a selector in the input port of the Head–Control module and constrain it to receive data from each module under specific conditions. The concept is shown in Figure 2.3 where a port arbitrator is used in the input port of the Head–Control (shown as box labeled with two 'M'). The arbitration logic can be written using a scripting language and is loaded by the port arbitrator. Our previous work [PMN] has demonstrated that this type of arbitration mechanism can be effectively used to implement complex tasks without resorting to centralized coordinators.

2.3.1 Architecture of Port Arbitrator

Figure 2.4 represents the internal architecture of the port arbitrator. A port arbitrator consists of multiple port monitors, a set of selection constraints, an event container and a selector block. Port arbitrator extends the port's scripting API for setting constraints and altering events in the container. In fact, when a port monitor is used in an arbitrator, the user's script can access the extended API for arbitration.

A port monitor can be attached to each connection (C_i) going through the port arbitrator. It monitors the data from connection and inserts the corresponding events into a shared container.

A port monitor can also remove an event (if previously inserted by itself) from the container [2]. Normally events have infinite life time. This means that they remain valid in the container until they are explicitly removed by the monitor object. An event can also have a specific life time. A time event will be automatically removed from the container when its life time is over. For each connection C_i, there is a selection constraint written in first order logic as a Boolean combination of the symbolic events. Upon the reception of data from a connection, the selector evaluates the corresponding constraint and, if satisfied, it allows the data to be delivered to the input port; otherwise the data will be discarded. Clearly a consistency check on the boolean rules must be performed to guarantee that only a single connection C_i can deliver data at any given time. Listing 2.3 represents the extended port monitor's API in Lua which can be used with port arbitrator.

```
PortMonitor.setEvent(event, life_time)
PortMonitor.unsetEvent(event)
PortMonitor.setConstraint(rule)
```

Listing 2.3: Port monitor extended API in Lua for arbitration

We refer to the search–and–track example from Figure 2.3 to demonstrate how selection constraints are represented and how they can be evaluated based on events from a container. As we previously mentioned, the Head–Control module should receive data from the Look–Around module if the Face–Detector module is not sending any data. To do this, we first need to inform the port arbitrator about the status of the data from the Face–Detector (i.e., if it is sending any data or not) by setting an event into the container. Listing 2.4 represent a simple script to set the `'e_face_detected'` into the event container. The life time 1.0 indicates that `'e_face_detected'` will be automatically removed after one second if the Face–Detector is not sending any data.

```
1 PortMonitor.accept = function(data)
2     setEvent('e_face_detected', 1.0)
3     return true
4 end
```

Listing 2.4: An example of setting time event into a container.

At this stage, the selection rule that allows the data from the Look–Around module (C_2) to be delivered to the Head–Control module when `'e_face_detected'` does not exist in the event container, can be simply written as follows: `setConstraint('not e_face_detected')`. The data from the Face–Detector module should be freely delivered to the Head–Control. Thus the selection rule for the connection C_1 is written as follows: `setConstraint('true')`. As we previously described, constraints can be expressed as Boolean combinations of symbolic events. To evaluate the expression, every symbolic event is substituted with a Boolean value. If the event is present in the container, it represents a true value in the expression; otherwise it is evaluated as false.

2.4 A step–by-step example

To demonstrate the applicability and advantages of our approach, we present an experiment with the iCub humanoids robot [MSV08]. This experiment is completely built using modules

[2] This is similar to the Event–Mask mechanism used in user interface programming or in operating systems.

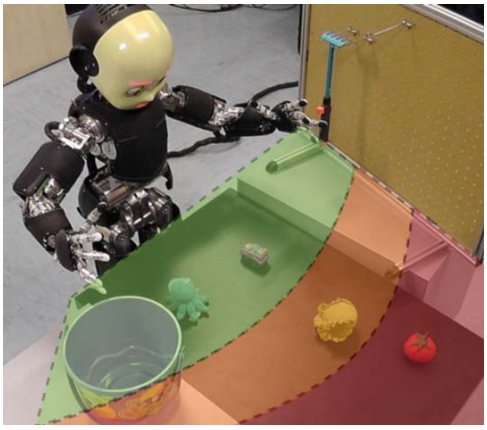


Figure 2.5: The experimental setup of table–cleaning application. The reachable zone is depicted in green, the orange zone represents the zone reachable with the tool and finally the red zone indicates the unreachable space, for which the robot needs human intervention.

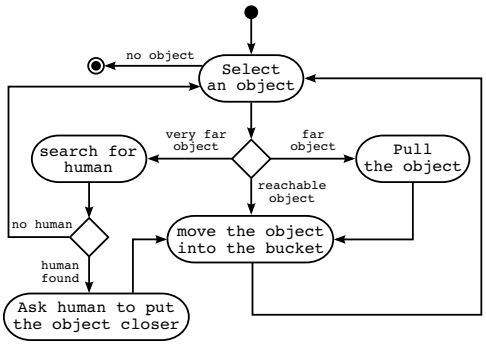


Figure 2.6: The simplified activity diagram that illustrate the table–cleaning application.

from the iCub software repository[3]. The experiment focuses on reusing (with no modifications) existing modules and by extending the required functionalities using port plug-ins. The overall behavior of the experimental task is demonstrated using a simplified activity diagram in Figure 2.6. The goal of the task, as shown in the activity diagram, is to clean the table by removing all the object and place them in a bucket located alongside the table. We allowed the robot to use a tool at his disposal (a rake), located on a rack, to reach objects of interest that are out of his workspace. The modules that allow the robot to grasp and use the tool are implemented as described in [TPNM13]. Furthermore, we consider also the case in which the object is too far to be reached even by the use of the tool. In this case the robot should look for a human and asks his intervention (put the object within reach). Figure 2.5 shows the experimental setup and it illustrates the three areas in which objects can be placed.

The activity diagram depicted in Figure 2.6 may give the impression that the task is only composed of a few simple steps that the robot should follow to accomplish it. But in fact, there are many uncertainties and unexpected conditions which should be taken into consideration to make the task robust. For example, the proper decision should be taken if an object drops from the hand while the robot is placing it into the bucket. Similarly the robot should behave appropriately while it is holding the tool to pull the object closer, the human might intentionally

[3]Modules can be downloaded from: https://github.com/robotology/icub-main.git and https://svn.code.sf.net/p/robotcub/code/trunk/iCub/contrib

intervene and move the object within the iCub's workspace. Considering all possible uncertainties, in fact, reveals the underlying complexity of the task which requires that many modules (e.g, for perception, action and coordination) are properly used and orchestrated (e.g, coordinating robots, gaze, arm, speech) to perform the required task.

Table 2.1: A subset of modules used for the experiment

Module	Input	Output	Type
Face–Detector	image	pos_3D	perception
Object–Detector	image	List<pos_3D>	perception
Bucket–Detector	image	pos_3D	perception
Look–Around	-	pos_3D	implicit action
Head–Control	pos_3D	-	action
Pick–and–Place	msg_cmd	msg_status	action
Pull–Object	msg_cmd	msg_status	action
Speak	msg_text	-	action

The modules used in this experiment are chosen from the iCub software repository and listed here in Table 2.1. To build the desired application, a few modules might simultaneously require to grab the camera image frames from the robot, control the arms and hands in various modes, such as Cartesian or joint space using velocity or position control. However, for the sake of brevity, only a subset of these modules are described in this deliverable. We use the previously mentioned Face–Detector and the Look–Around modules.

Object–Detector gets as an input image from the cameras and produces a list of blobs and extracts 3D positions of all the possible graspable objects as its output. Bucket–Detector is, in fact, an instance of a generic object detector which is configured and trained to recognize this specific object. As we previously mentioned, Look–Around randomly produces positions in 3D space which are used by Head–Control to move the gaze in various positions. The Pick–and–Place module receives a set of commands (e.g., `take <3D_pos>`, `put <3D_pos>`) to take an object and release it on a specific position. The internal status of the module (e.g., `e_taken`, `e_arm_idle`) is continuously sent out using status messages. Pull–Object is a complex set of modules which together get the position of an object on the table and use a tool to bring the object closer [TPNM13]. Similar to Pick–and–Place, the internal status of the Pull–Object module is advertised via its output. The Speak module receives a text message and performs a text–to–speech synthesis. Generally speaking, in order to be able to integrate some modules for building an application, two important points should be considered: *i)* data type on both side of the connections should match and *ii)* a proper coordination mechanism should orchestrate modules to perform the task. We start with the simplest case in which the objects are reachable by the iCub and progressively extend it to build the complete table–cleaning application.

2.4.1 Handling reachable objects

First our application should select the closest object within the reachable area and take it (see Figure 2.7-A). To do that, we connect the output of Object–Detector to the input of Pick–and–Place. Using the port monitor, we implement a simple script that goes through the list of objects, select the one that is closest to the robot and produces the proper 'take' command (i.e., `take <3D_pos>`) for execution. Similarly, to put the object into the bucket we connect the output of Bucket–Detector with the same input of Pick–and–Place and attach to this connection

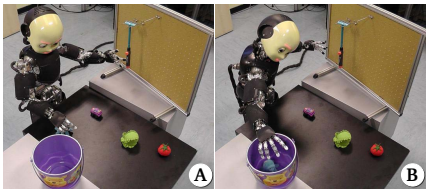


Figure 2.7: The iCub performing table–cleaning on reachable objects. The robot takes the object (A) and places it into to bucket (B).

another port monitor that generates the 'put' command (i.e., `put <3D_pos>`) for execution (see Figure 2.7-B)

Furthermore, an object should be taken only if the hand of the robot is free and the robot is not performing another action using the arm. On the other hand, the 'put' command should be sent to the Pick–and–Place module if the robot is holding an object. To this aim, the status of the Pick–and–Place module should be monitored and the required arbitration rules should be added to the system to properly coordinate taking, placing and releasing actions. Figure 2.8 represents the configuration of the modules that perform this simple task on the reachable objects. As shown in the figure, the status output of the Pick–and–Place module is used to inform the arbitrator about the internal state of the module. Below we illustrate how this is achieved.

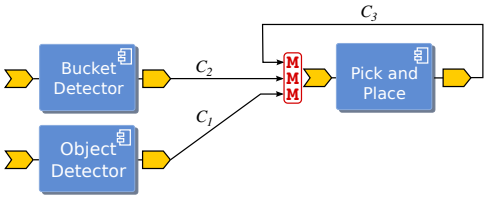


Figure 2.8: Configuration of the modules for handling reachable objects on the table.

As we have previously mentioned, a monitor object is assigned to each connection going through the port arbitrator. Listings 2.5, 2.6 and 2.7 respectively represent pseudo–scripts which will be loaded by each monitor object for connections C_1, C_2, and C_3. Listing 2.7 demonstrates the script which is assigned to the monitor object of connection C_3. This monitor receives status messages from Pick–and–Place (i.e., `e_taken`, `e_arm_idle`) and adds them to the event container of the port arbitrator. These events will be used for the selection of C_1 and C_2. Notice that the connection C_3 and the corresponding script (Listing 2.7) are created to make the status events available for the arbitration. These events will be never delivered to Pick–and–Place. This is achieved by refusing to accept the data from the connection C_3 (`return false`).

```
1  PortMonitor.create = function()
2      setConstraint('not e_taken and e_arm_idle')
3     return true;
4  end
5
6  PortMonitor.accept = function(object_list)
7        -- find closest_obj in the object_list
8        if closest_obj.dist > HAND_REACHABLE then
9           return false
10       end
11    return true
12 end
```

```
13
14 PortMonitor.update = function(object_list)
15    return command('take', closest_obj.pos)
16 end
```

Listing 2.5: Monitoring and arbitrating connection C_1.

```
1 PortMonitor.create = function()
2    setConstraint('e_taken and e_arm_idle')
3    return true;
4 end
5
6 PortMonitor.update = function(bucket_pos)
7    return command('put', bucket_pos)
8 end
```

Listing 2.6: Monitoring and arbitrating connection C_2.

```
1 PortMonitor.accept = function(status_event)
2    setEvent(status_event, 0.5)
3    return false
4 end
```

Listing 2.7: Monitoring connection C_3 for generating events.

Listing 2.5 deserves particular attention: First, within the 'create' callback, the required selection rule for the connection C_1 is set into arbitrator. The rule implies that data from corresponding connection should be delivered if the robot has not already taken (`not e_taken`) an object and if it is not performing an action (`e_arm_idle`). In the 'accept' callback, first the closest object to the robot is selected from the list of detected objects. If the object is reachable (the data is accepted), the 'update' method will be called to generate the 'take' message to be delivered to Pick–and–Place. If the object is out of reach, it will be discarded (`return false`). Similar Listing 2.6 represents the script that generates the 'put' command and that specifies the condition which enable action generation.

2.4.2 Handling objects using tool

We now extend the previous application to allow the iCub to use a tool to bring unreachable object within its workspace (see Figure 2.9). Figure 2.10 represents how Pull–Object is integrated in the application. The output of Object–Detector module provides a list of objects; this list should be filtered to select one object that is within the tool–reach area and out of the robot's workspace. The position of this object should be given to the Pull-Object to trigger a sequence of actions to take the tool from the rack, reach for the object with the tool, pull the object and finally putting back the tool on the rack (see Figure 2.9-B, C, D).

```
1 PortMonitor.create = function()
2    setConstraint('not e_taken and e_arm_idle')
3    return true;
4 end
5
6 PortMonitor.accept = function(object_list)
7    -- find closest_obj in the object_list
8    if closest_obj.dist > TOOL_REACHABLE then
9       return false
```

```
10      end
11      return true
12  end
13
14  PortMonitor.update = function(object_list)
15      if closest_obj.dist < HAND_REACHABLE then
16          return command('cancel', nill)
17      end
18      return command('pull', closest_obj.pos)
19  end
```

Listing 2.8: Monitoring and arbitrating connection C_4.

Once the object is located within the reachable area of the robot, the previous picking–and–placing application is activated. Appropriate selection rules should be added to the system to properly arbitrate pulling and pick–and–placing.

Listing 2.8 represents the pseudo code of the script which is used in the port monitor of connection C_4. The selection constraint (`not e_taken and e_arm_idle`) filters messages to Pull–Object when the robot is already involved in other actions (i.e. picking and placing an object). Similar to Listing 2.5 from the previous application, first the closest object is extracted from the list of detected objects. This object is accepted and generates a 'pull' command if it is within the tool–reach area. Otherwise it is discarded. An interesting behavior is the fact that the pulling action is composed of several sub–actions that should be aborted if the tool becomes unnecessary (e.g. if a human moves the target objects in the workspace of the robot). This is achieved by continuously monitoring the target object in the 'update' function and generating the 'cancel' command when necessary. Notice that as opposed to Pick–and–Place, Pull–Object ignores redundant 'pull' commands until all ongoing sub-actions are accomplished or aborted (with the 'cancel' command). Therefore, unlike Pick–and–Place, we do not need to monitor the internal status of Pull–Object and filter conflicting 'pull' commands.

Clearly Pick–and–Place and Pull–Object are conflicting behaviors. To avoid conflicts the selection rule for connection C_1 must be updated to prevent generation of 'take' commands while Pull–Object is active (i.e. not idle). This is achieved by making the internal state of the Pull–Object available in the arbitrator of Pick–and–Place via connection C_5 and by modifying the

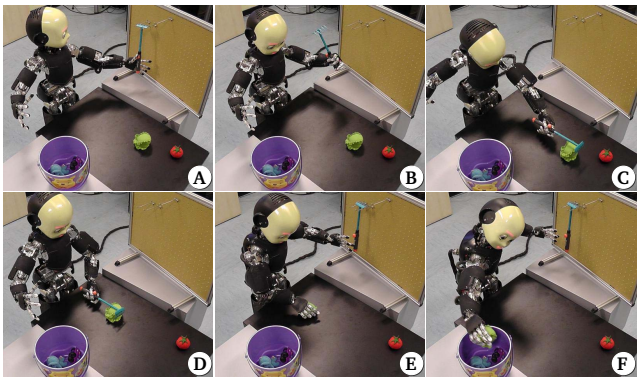


Figure 2.9: The iCub performing table–cleaning using a tool (rake). The robot take the tool (A), reaches for the object (B,C), pulls the object (D), grasps the object (E) and finally places it into the bucket (F).

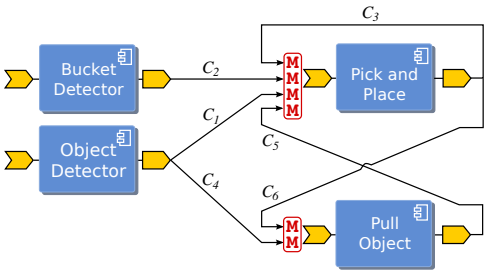


Figure 2.10: Configuration of the modules for handling objects within tool–reach space.

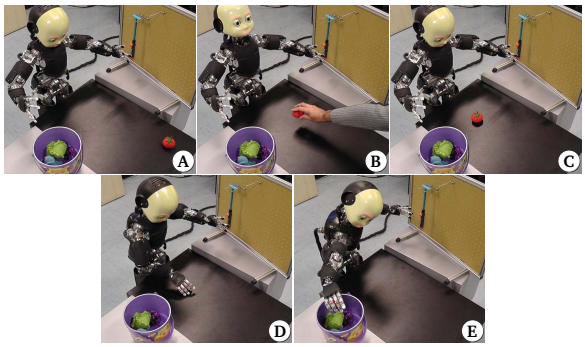


Figure 2.11: The iCub performing table–cleaning with human assistance. The robot detects an unreachable object (A), detects the presence of a human and asks assistance (B,C), grasp the object (D) and finally places it into the bucket (E).

selection constraint of Listing 2.5 as follows: `'not e_taken and e_arm_idle and e_pull_idle'`

As for the connection C_3, Listing 2.7 is used for the port monitor of connections C_5 and C_6 to inserts the status events into the corresponding event containers.

2.4.3 Handling objects with human assistance

In Section 2.3 we explained how the Face–Detector and Look–Around modules can be properly used with the Head–Control module to implement a basic face tracking application. In this section, we use these modules to complete our table–cleaning application. When an object is completely unreachable, the robot should look for a person and asks assistance (see Figure 2.11). Figure 2.12 depicts the complete system. The output of Object–Detector arbitrates the connections from Face-Detector and Look-Around via C_7 and C_{11} so that when required, the robot will look around searching and tracking human faces. This is achieved in Listing 2.9 by monitoring the closest object and generating an event `'e_unreachable'` when the latter is out of the tool–reach area. Notice that this event is cleared (removed from the container) only when the object becomes reachable again.

```
1 PortMonitor.accept = function(object_list)
2     -- find closest_obj in the object_list
3     if closest_obj.dist > TOOL_REACHABLE then
4         setEvent('e_unreachable')
5     else
6         unsetEvent('e_unreachable')
```

```
7     end
8     return false
9  end
```

Listing 2.9: Monitoring connection C_7 and C_{11}.

```
1  PortMonitor.create = function()
2     setConstraint('e_unreachable')
3     return true;
4  end
5
6  PortMonitor.accept = function(data)
7     if time() - time_prev < DESIRED_TIME then
8        return false
9     end
10    time_prev = time()
11    return true
12 end
13
14 PortMonitor.update = function(data)
15    return msg('Please put the object closer!')
16 end
```

Listing 2.10: Monitoring and arbitrating connection C_{10}.

Messages from Look–Around and Face-Detector are discarded depending on the internal state of Pick–and–Place and Pull–Object via connections C_{12} and C_{13} and the event generator script (i.e., Listings 2.7). This prevents moving the head when the robot is picking, placing or attempting to pull an object. Finally the output of Face–Detector generates a voice message synthesized by the Speak module. This is achieved by connecting the two modules (C_{10}) and adding a script to the corresponding port monitor. This script generates a text message (a valid command for the Speak module) if a human face is detected, but only if a certain amount of time has passed from the last command, to reduce verbosity (Listing 2.10). Notice that these commands are arbitrated by C_{11} so that the speech is activated only when necessary.

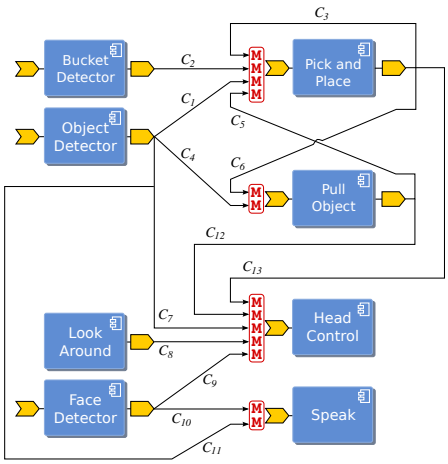


Figure 2.12: Configuration of the modules for table–cleaning application.

Chapter 3

Middle Level

The robot programming environment ArmarX [WVW+13] has been developed in order to ease the development of higher level capabilities needed by complex robotic systems such as humanoid robots. ArmarX is built upon the idea that consistent disclosure of the system state strongly facilitates the development process of distributed robot applications.

Beside the development aspects that are addressed by the framework, ArmarX serves as a software backbone for distributed robot programs as they are needed in complex systems in order to incorporate all aspects of distributed sensor processing, robot control, planning and task scheduling. The advanced inspection capabilities of ArmarX (graphical user interfaces, state chart editors, etc.) provide possibilities to monitor and control the data and control flow within a distributed robot application in order to allow both, the developer and the robot program itself, to gain full knowledge about the current system state. The information can be used to predict future resource usage profiles and to monitor execution footprints for bootstrapping novel execution sequences.

There are several essential prerequisites for any RDE that have to be met in order to fulfill the requirements for robotics research and development. In the following, we will briefly discuss design principles, which are considered by ArmarX to provide solutions for the demands of current robot platforms and development processes.

- **Distributed processing**
  Due to the complexity of robotics a typical robot hardware architecture consists of several embedded PCs. Software components are spread over several, sometimes specialized, computation units to fulfill hardware requirements or real-time conditions. Easy interfacing and connecting of these components is a key feature of modern RDEs. ArmarX addresses these issues with distributed applications that are coordinated by a central managing system. This allows for load balancing and robustness in case of failing hardware components.

- **Disclosure of the system state**
  Another key design principle of ArmarX is the disclosure of the internal system state on

all levels - from the from the meta level to the sensorimotor level to the task level. Three mechanisms support this idea on the technical level: inspectable hierarchical statecharts with well-defined interfaces, well-defined application interfaces and flexible communication mechanisms. Distributed applications, the robot program, the robot sensors, and the internal model of the world establish the disclosure of the internal system state.

- **Interoperability**
  An essential feature of an RDE is to support an assortment of different hardware platforms and operating systems for an easy integration of new components without the necessity to adapt the new component to the given circumstances. Therefore, ArmarX can be compiled under Linux, Windows and Mac OS X. Additionally, ArmarX uses an interface definition language (IDL) which supports a variety of platforms and programming languages. Thus, the supported operating systems can easily be extended by implementing the ArmarX interfaces on the target platform using one of the supported programming languages. By the IDL supported programming languages are C++, .NET, Java, C#, Objective-C, Python, Ruby, PHP, and ActionScript.

- **Open Source**
  To achieve the most impact on robotics an RDE should be open source since the budget is spent usually on hardware than on software. Additionally, open source software allows developers to achieve a deep insight in the underlying mechanisms of the RDE. Consequently, ArmarX is available open source under the GPL license.

The core layer of ArmarX comprises four main building blocks: inter-object communication, Sensor-Actor Units, Observers, and Operations as shown in Figure 3.1. The Inter-object communication block provides convenient mechanisms for communication between objects of the systems, while these objects could be any of the other buildings blocks. The Sensor-Actor Units present a generic interface as the lowest level of abstraction for robot components like motor control or sensors. These units are monitored by the Observers for changes to send application specific events to appropriate operations. These operations are designed as statecharts. They process the events and send resulting control commands to the Sensor-Actor Units, which control the robot or the dynamic simulation.

3.1 Statecharts - High-Level Programming in ArmarX

Complex robot operations are usually compositions of a number of individual smaller operations which result into certain states of the robot and the environment. Based on this principle, robot operations in ArmarX are arranged as hierarchical, distributed, and orthogonal statecharts based on Harel's approach [Har87]. Statecharts are widely used in robotics to control the behavior on a high level [KB12, MRW06, BECHR10, BC10]. In the well-known RDE ROS [QCG+09] an approach [BC10] is employed that focuses on the data-flow similar to the

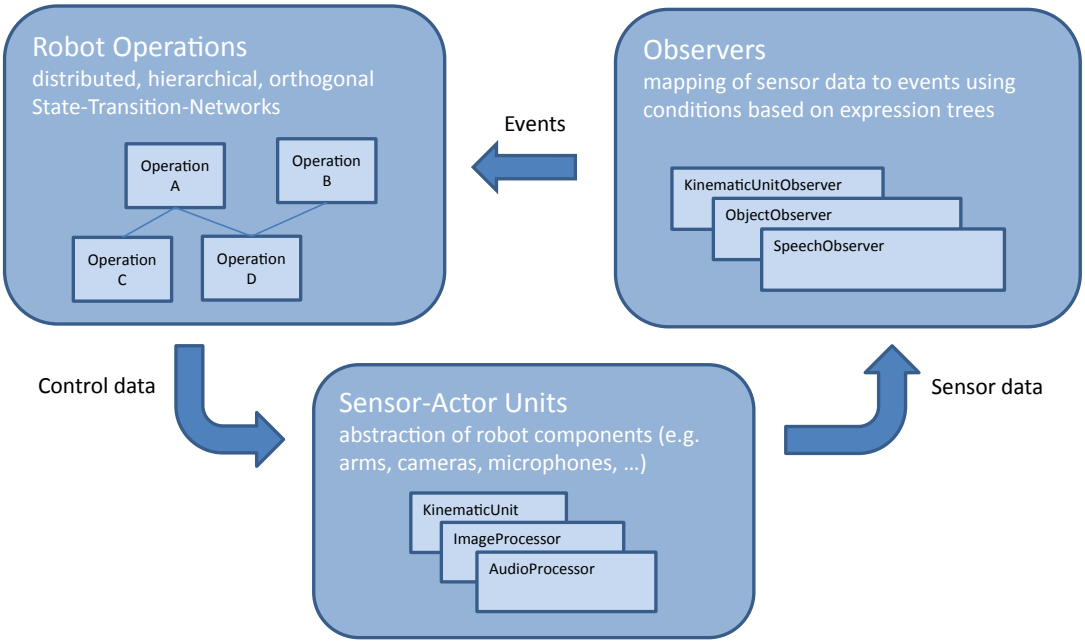


Figure 3.1: The application programming interface comprises four different elements. The Sensor-Actor Units serve as abstraction of robot components, the Observers generate events from the continuous sensory data stream which result in transitions between Operations. The operations are organized as hierarchical state-transitions networks. These elements are connected by the communication mechanisms (arrows in the figure).

dataflow in our approach, but differs on the triggering of transitions. In many points our statecharts are similar to the rFSM [KB12] from Orocos [BSK03] which focus on the coordination of components, while our approach concentrates more on the disclosure of the internal state.

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

- *Modularity* is supported in our statecharts naturally through the individual states and a specified input and output.
- *Re-usability* is ensured since every state can be used as a substate in any other state and has a specific interface for interaction.
- *Runtime-reconfigurability* means that a statechart can be defined in configuration files and that the structure of a statechart 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 components. This enables load balancing and robustness. A crashed distributed state component would not crash the whole statechart, i.e. the complete high level robot behavior, but would just create an event for higher layers that this specific component has failed.

- *State-disclosure* means that the statechart can be inspected at run-time and logged for future behavior adaptation. The state transition chain and the dataflow between states could be used to predict the hardware workload or optimize the robot behavior.

3.2 Memory Structures

MemoryX, the memory layer of ArmarX, includes basic building blocks for memory structures which can be either held in the system's memory or made persistent in a non-relational database. Based on these building blocks, the memory architecture illustrated in Figure 3.2 is realized. A key element of MemoryX is the network transparent access facilities which allow consistently updating or querying the memory within the distributed application. The architecture consists of different memory types such as working memory (WM), long term memory (LTM), and prior knowledge. Each memory type is organized in segments which can be individually addressed and used to store arbitrary types or classes.

- **Working Memory (WM)**
  The WM represents the current context of the robot's internal state. It can be updated by perceptual processes or prior knowledge via an updater interface. The memory consists of logical segments which hold information about perceived object instances, object classes, agents (including the robot itself) and world entities.

- **Longterm Memory (LTM)**
  Compared to the short term memory structures, the LTM provides long term storage capabilities (e.g. database access) and offers an inference interface which allows attaching learning and inference processes. The LTM can also be used to save snapshots of the working memory in order to load them later on for extended processing like learning, comparison, statistical analysis.

- **Prior Knowledge**
  Prior knowledge contains information which is already known to the robot. Entities in prior knowledge can be enriched with known data such as 3D models or features for object detection which usually refer to user generated data that form the initial world knowledge of the robot.

Powerful update mechanisms can be realized with this setup. For example, a perception module localizes a known object and updates the WM. This update creates a new entity of the object with the current location and enriches it with the 3D model from the prior knowledge database. Besides the possibility of directly addressing the WM, an observer exists which allows installing conditions based on the memory content. If the associated content changes the matching events will get generated.

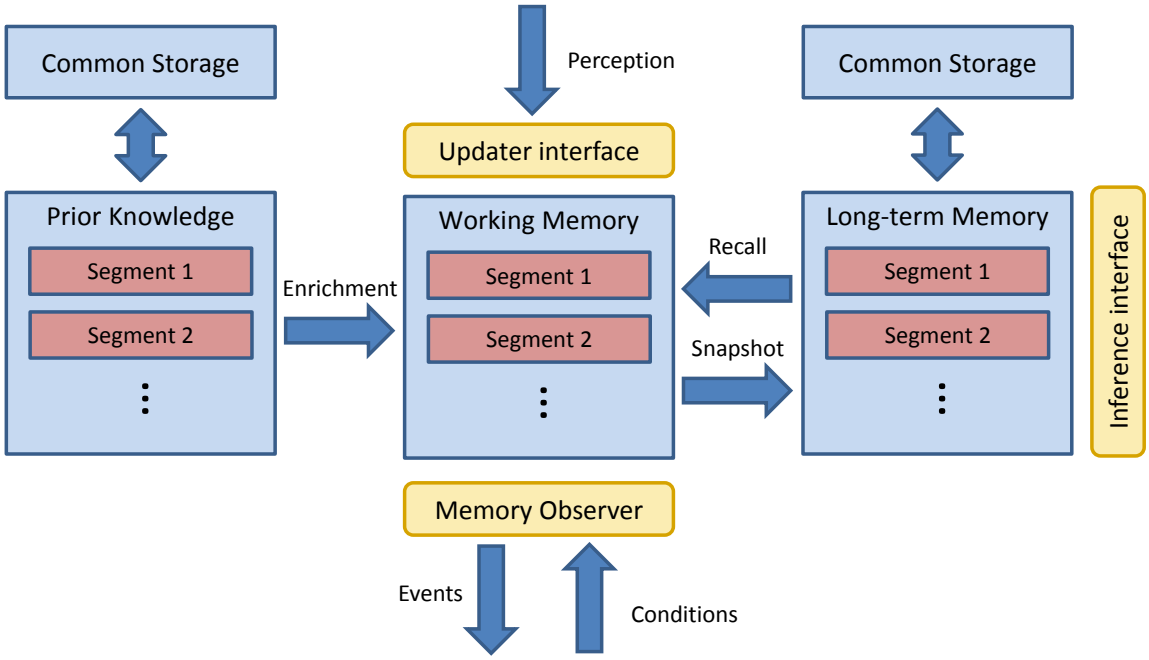


Figure 3.2: ArmarX offers the MemoryX architecture consisting of a working memory and a long-term memory. Both memories can be accessed within the distributed application. Appropriate interfaces allow attaching processes to the memory for updating and inference.

3.3 OAC representation with statecharts

In this task, we worked on a hierarchical representation for Object-Action Complexes (OACs) [KPP+11]. An OAC is a formalism for representing interactions with objects. They are compositional and can be arbitrary complex, e.g. the OAC 'Open door' contains the OAC 'Grasp handle'. They require a defined world state before the action and predict the effects of the OACs on the world state (preconditions and effects). Additionally, they may provide some new information, which was gained during the execution of the OAC. To be able to represent all kinds of OACs with one representation a powerful internal structure is needed.

We developed a new structure to represent OACs as part of our new Robot Development Environment ArmarX, which satisfies the demands of OACs and is based on the ArmarX design principle 'Disclosure of the internal state'. We modified the statecharts of Harel [Har87] to suit our requirements of a controlled and inspectable dataflow and state reusability, and developed a new statechart specification.

The ArmarX statecharts support the OAC requirements as follows:

- **Compositional OACs**
  Statecharts are hierarchical and divide an action into smaller sub-actions. In statechart

notation an OAC corresponds to a state with sub states. Furthermore, the states ensure reusability of OACs by a specified, but flexible interface. Therefore, states, i.e. OACs, can be reused in a higher level state.

- **Parameterization**
  The states in a statechart specify input parameters of various types, which are provided by previous states and managed by the higher level states before the state is entered. The state itself is unaware of the source of the input parameters, which ensures the reusability of states. A higher level state has to provide the parameterization of this state by utilizing the context knowledge that is available at the higher level but unimportant for the lower level.

- **Preconditions**
  The OAC preconditions are crucial for a correct execution of an OAC. To this end, ArmarX provides a powerful condition management system that allows generating complex conditions over all robot components. With this system the fulfillment of the OAC preconditions can be checked and a transition to the next state, i.e. the next OAC, can be triggered.

- **Effects**
  The effects of OACs are propagated in the exiting phase of a state, which triggers an evaluation of the conditions for the next OAC.

- **Gained information**
  By executing the OAC new information might be gained. This information is made available for subsequent states via the output parameters of a state.

With this OAC representation the internal state can be easily visualized for a convenient overview over a complex task and for run-time inspection. Furthermore, it is possible to generate new OACs automatically as a composition from basic OACs (see figure 3.3).

As shown in [WSA+13] it is possible to segment a observed human demonstration of a complex task automatically into smaller basic OACs. With this segmentation and the new OAC representation as statecharts it is possible to generate a new OAC, i.e. a new statechart, automatically by creating a chain of states (consisting of existing OACs) connected by transitions. The parametrization is extracted from the world state of the key frames and the change of the world state at the key frames of the segmentation is converted into ArmarX conditions that trigger state transitions.

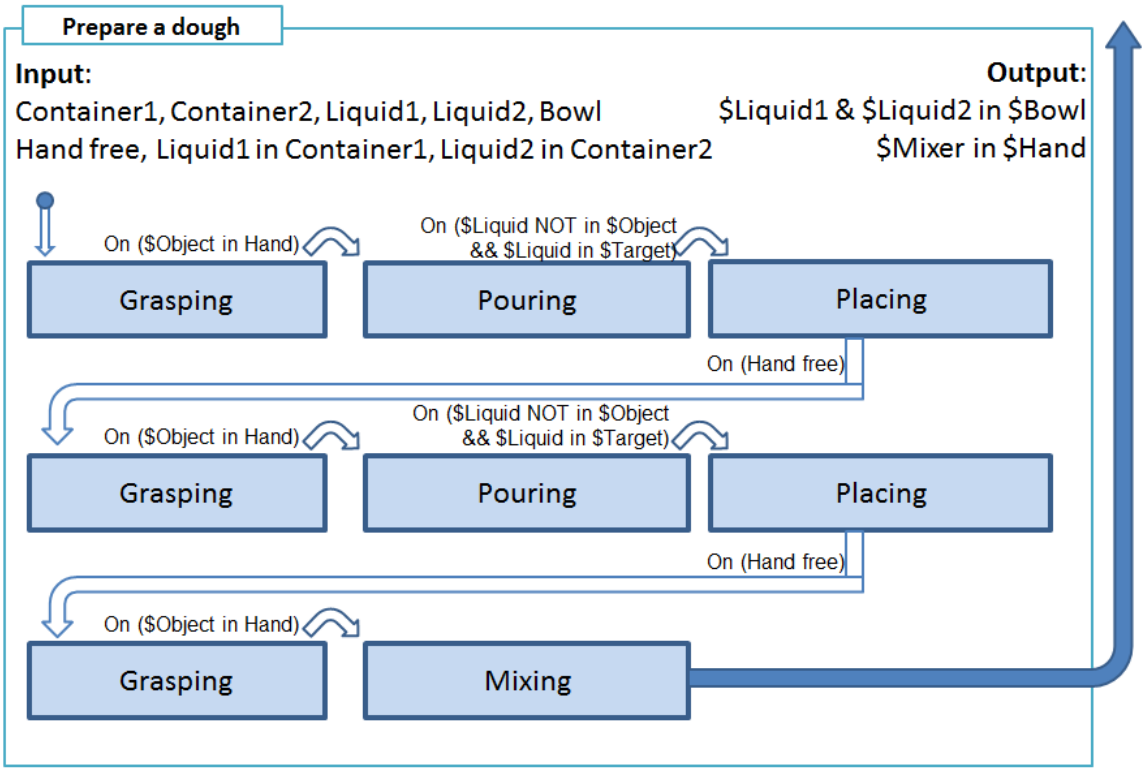


Figure 3.3: Example of a composition of basic OACs into a complex task with the statechart notation.

Chapter 4

Planning Level

The following description is in good overlap with D3.2.3. Here we summarize the important aspects related to integration into the architecture. The main goal here is to extend the capabilities of current high-level planning models by applying structural bootstrapping to the knowledge-rich representation of actions and plans, to provide the apparatus needed to support plan generation and execution in low-level robotics domains and higher-level domains requiring language and communication.

To this end, we extended the PKS planner [PB02, PB04]. PKS is a state-of-the-art conditional planner that constructs plans in the presence of incomplete information. Unlike traditional planners, PKS builds plans at the knowledge level, by representing and reasoning about how the planner's knowledge state changes during plan generation. Actions are specified in a STRIPS-like [FN71] manner in terms of action preconditions (state properties that must be true before an action can be executed) and action effects (the changes the action makes to properties of the state). PKS can build contingent plans with sensing actions, and supports numerical reasoning, run-time variables [EHW$^+$92], and features like functions that arise in real-world planning scenarios.

Like most AI planners, PKS operates best in discrete, symbolic state spaces described using logical languages. As a result, research that addresses the problem of integrating planning on real-world robot platforms often centres around the problem of representation, and how to abstract the capabilities of a robot and its working environment so that it can be put in a suitable form for use with a goal-directed planner. A key problem when constructing such a representation is the question of how to encode the planner's knowledge about objects that are not completely described in the domain model (but which are known or believed to exist), and how to make assertions about domain properties that reference such objects. Similarly, as new information becomes available to the planner, for instance as a result of sensorimotor processes external to the planner itself, a facility should exist for updating the planner's underlying domain model to make use of this more certain information.

To address these problems in complex planning domains, we have reported (D3.2.1 and D3.2.3) two main threads of research related to the PKS planner. These are concerned with the use

of interval-valued fluents for modeling uncertain numerical information, in order to capture the effects of noisy sensors and noisy effectors at the planning level. Since noisy information is common in real-world robot domains, we believe these extensions will be particularly useful in integrating high-level planning with the low-level sensorimotor systems on Xperience. Moreover, this representation gives rise to a form of indexical (or relative) referencing during plan generation, which addresses the problem of making assertions about domain properties for partially-defined objects. This work is described in [Pet14] which is available in D3.2.3 (attached paper).

The second extension is central to the integration effort of this deliverable and it is in fact an application programming interface (API) to the planning component, which supports the modification of existing planning domains at run time using the Internet Communications Engine (ICE). In particular, the API abstracts common planning activities already supported by PKS, including functions for adding new actions, properties, and objects to a planning domain, and offers a network-based solution for communicating with the planner component. One of the main contributions of this interface is that it is designed to be generic, which offers the possibility that alternative planners could be used in place of PKS, facilitating future integration tasks, provided they support the same interface. Additional details are described in D3.2.3 ([Pet13]).

4.1 Interface design philosophy

This document is not meant to provide precise details concerning the implementation of the PKS API, however, we note the following design decisions which affect our current implementation:

- Internet Communications Engine (ICE): The API detailed in D3.2.3 is implemented using the Internet Communications Engine (http://www.zeroc.com/ice.html), which provides an object-oriented middleware for building distributed applications. The default implementation using the PKS planner provides a planning server which allows clients to to access the services provided by the API.

- Support for multiple backends: The current implementation of the planning API was adapted from the interface to the PKS planner, but has been abstracted to avoid PKS-specific representations and syntax. API functions connect the ICE layer to a version of PKS implemented as a C++ library, which is linked to form the plan server. However, there is no strict requirement that PKS must be used as the planning backend, and any planner which is able to implement the API can be used in its place as an alternative backend.

- Backend-dependent syntax: Many of the functions in our API require specifying a file or a definition for a particular aspect of the planning domain (e.g., actions, problems, symbols, etc.). The precise syntactic form of these definitions is left to the backend planner. As a result, this means that the content (parameters) of certain function calls may change from backend to backend. However, this also means that all planning domain entities do not

need to be standardized in order to support the API. This functionality may change in the future as we reconsider certain aspects of the interface.

4.2 Interface elements

The elements of the interface (which represent the main classes of functions in PKS) are:

- Properties and states;
- Plan steps and plan sequences;
- Planner configuration and debugging;
- Domain configuration;
- Plan generation and plan iteration.

The following subsections describes these elements with some additional details.

4.2.1 Properties and states

A structure called `StateProperty` defines the abstract notion of a domain property (or feature, fluent, relation, function, etc.) as an entity with a `name`, a list of arguments `args`, a `sign`, and a `value`. This definition is meant to accommodate both relational and functional entitles which commonly arise in planning states. For instance, a relation like $\neg F(a)$ could be encoded as:

```
name : F
args[0] : a
sign : false
value : (unused),
```

while a function mapping like $f(a) = c$ could be encoded as:

```
name : f
args[0] : a
sign : true
value : c.
```

A state can be thought of as simply a list of `StateProperty` definitions, denoted in the API as `StatePropertyList`. Note that this definition supports the standard STRIPS-style view of states as collections of instantiated properties, and is consistent with many types of planning approaches. It can also be used to encode the notion of an observed state, as a collection of properties as returned from a set of sensors.

4.2.2 Plan steps and plan sequences

A `PlanStep` can be thought of as a particular instantiated action in a plan, which is defined by a structure specifying its `name`, its `type`, and a list of parameters, `args`. For instance, an instantiated action `pickup(blockA, table, lefthand)` (pick up `blockA` from the `table` using the `lefthand`), which denotes a type of manipulation action, could be encoded in this structure as:

```
name : pickup
type : manipulation
args[0] : blockA
args[1] : table
args[2] : lefthand.
```

The `type` field is not often used by many traditional "linear" (i.e., non-hierarchical) planners, but may provide useful heuristic information to an execution system at run time. A `PlanStepList` is simply a sequence of `PlanSteps` (i.e., instantiated actions) which can also be thought of as a simple linear plan, of the form that most classical planners are able to generate. This allows the possibility of providing a generic container for returning plans to external modules that does not rely on the particular plan encoding used by the underlying planning system. More complex plans (e.g., contingent plans involving branches, or programs involving loops) could be encoded using standard containers (e.g., trees, maps, etc.) found in most modern programming languages (e.g., STL containers in C++).

4.2.3 Planner configuration and debugging

The first set of functions in the planning API provide a planner-independent way of configuring the underlying planning system, and providing access to certain features needed for debugging:

- `reset()`: This function resets the planner to its initial state.
- `getPlannerProperty(string s)`: This function returns the state of the planner property variable `s`. The precise set of accessible properties is defined by the underlying planner.
- `setPlannerProperty(string s, string t)`: This function sets the state of planner property `s` to value `t`. The precise set of accessible properties and associated values is defined by the underlying planner.
- `getInternalStructure(string s)`: This function is a hook to allow internal planning structures to be queried by external modules. This is primarily included to provide access to internal debugging information.

In general, the implementation of these functions relies on such features being supported by the underlying planner. Since many planners offer such functionality already, these functions simply standardise the interface.

4.2.4 Domain configuration

The next set of functions provide the main methods for defining planning domain models to the planning system. These functions provide support for loading predefined models, or incrementally augmenting existing models at runtime:

- `clearDomain(), clearActions(), clearProblems(), clearStates()`: These functions direct the planner to delete any domain (similarly, actions, problems, or states) that are currently defined.
- `loadDomain(string s)`: This function directs the planner to load a domain from the specified file/URL `s`. The actual format of the domain is specified by the backend planner.
- `loadSymbols(string s)`: This function directs the planner to load a set of symbol definitions from the specified file/URL `s`. Symbol definitions typically involve a specification of the allowable objects, types, and properties in a planning domain.
- `loadActions(string s)`: This function directs the planner to load a set of action definitions from the specified file/URL `s`. Actions are defined in a language supported by the backend planner.
- `loadProblems(string s)`: This function directs the planner to load a set of problem definitions from the specified file/URL `s`. A problem definition typically consists of initial state and goal specifications, but may also contain additional problem constraints or control information. Again, the precise form of a planning problem is specified by the backend planner.
- `loadPlanState(string s)`: This function allows the planner to load a cache a state definition from the specified file/URL `s`. Such a state can be used by the planner as a starting state, or a possible recovery state for replanning purposes. The only hard requirement this function imposes on the backend planner is that this state be cached for future use.
- `loadObservedState(string s)`: This function is similar to `loadPlanState` except the loaded state is additionally tagged as being an observed state. The only hard requirement this function imposes on the backend planner is that this state be cached for future use.

- `defineDomain(string s), ..., defineObservedState(string s)`: These functions are analogous to the functions `loadDomain(string s), ..., loadObservedState(string s)`, as defined above, except rather than loading definitions from a specified file or URL, the definitions are directly included in the parameter string `s`. These functions allow all domain definitions to be performed directly through function calls, without requiring access to external files.

- `definePlanStateFromList(StatePropertyList s), defineObservedStateFromList(StatePropertyList s)`: These functions are similar to their counterparts `definePlanState` and `defineObservedState`, except rather than specifying a state definition in a string `s`, it is defined using the state structure `StatepropertyList`, as described above.

One of the important ideas behind these functions is that they offer the possibility of specifying domains to the planner incrementally, using function calls alone, rather than specifying a single monolithic domain file to the planner as a single entity, as is usual for many off-the-shelf planners from the planning community. This means that an initial domain could be specified and then later revised, for instance due to additional information discovered by an external learning process (e.g., new domain objects, revised action descriptions, additional properties corresponding to new capabilities of the robot, etc.). This is a potentially powerful mechanism, however, it pushes the problem of how a planner should react to a change in the planning domain onto the planner itself. Conceptually, this may present problems for the underlying planner, especially in the presence of partially built plans, and this API offers no solution to this problem.

4.2.5 Plan generation and plan iteration

The final set of functions defined in the API specify methods for controlling various aspects of the plan generation process, and for iterating through generated plans:

- `buildPlan()`: This function directs the planner to generate a plan using the current settings, domain, and default planning problem.

- `clearPlan()`: This function directs the planner to clear the current plan in its memory, if one exists.

- `getCurrentPlan()`: This function directs the planner to return the current plan as a string. This function is normally used to direct the planner to return a plan in its native format.

- `getCurrentPlanAsList()`: This function directs the planner to return the current plan as a PlanStepList structure. As a result, this function currently only supports plans that can be returned as a linear sequence of actions.

- `getNextAction()`: This function directs the planner to return the next action in a plan, as a PlanStep structure.
- `getNextActionUsingControlInfo()`: This function is similar to `getNextAction` except it allows for the specification of additional control information in the parameter string `s`. This information is intended to help resolve plan ambiguities concerning execution decisions (e.g., which branch of a plan should be followed, whether a loop termination decision has been achieved). The precise form of the control information is planner dependent.
- `isNextActionEndOfPlan()`: This function determines whether we have reached the end of the plan during plan iteration.
- `isPlanDefined()`: This function returns a status update on whether or not a valid plan currently exists.
- `setProblem(string s)`: This function informs the planner that it should work with the planning problem specified by the string `s`. The string may specify a label to a previously defined problem, or contain the definition of a new problem.
- `setProblemGoal(string s)`: This function informs the planner that the current goal condition should be replaced by the goal specified by the string `s`. The problem is otherwise unchanged.

The idea behind many of these functions is to extend a degree of control over the plan generation and execution processes, as necessary, to components outside the planner itself, to the extent that simple plan execution monitoring activities can be supported without reliance on the planner. As a result, a client using these services can determine whether to generate a plan, and can iteratively ask for individual plan steps, advancing the plan one step at a time. Entire plans can also be processed by external processes in their entirety. The functions also support run-time updates to certain aspects of the planning problem, such as goal change.

4.2.6 Examples

Examples of the use of the PKS in the robotic domain are described in D3.2.3 and relative paper attachments.

Chapter 5

Benchmarking

During the first three years of the project the consortium has developed several bootstrapping methods, and has gained a much better understanding of the concept. This motivates a revision of the bootstrapping benchmarks outlined at the start of the project in D1.1.1. The following benchmarking specification replaces Chapter 3 of D1.1.1.

We benchmark bootstrapping mechanisms at three levels, sensorimotor, objects and actions, and symbolic. The following sections define benchmarks for each level. While in D1.1.1 we defined a generic multi-level benchmarks, here we are more specific analysing progress in various experiments. Each experiment is paradigmatic of the Xperience approach.

5.1 Sensorimotor Level

5.1.1 Learning to grasp unknown objects from partial point clouds information

This benchmark activity deals with the test of grasping methods on unknown objects. We analysed the quality of grasping on the iCub as described in [GPTM13b, GPTM13a]. In this section we describe the grasping evaluation for this method. We decouple two object properties, i.e.:

- shape: detected through stereo vision and image postprocessing;
- orientation: detected through stereo vision (see [GPTM13b]);
- weight: non measurable at the moment.

Following this subdivision, we devised the following tests:

- synthetic generic shape tests (full point cloud);

- synthetic real objects of various shapes (but similar weight);
- real objects with varying orientation (detected through 3D vision and using a partial point cloud).

It is to be noted that the weight (as well as other non detectable object properties) can be analysed by far and large even if the robot cannot really measure and/or take them into account for planning grasping. They nonetheless provide useful information to the user with respect to the applicability of a given robot to a given experimental scenario.

The following figure 5.1 shows an example of grasping using the iCub hand (model) on a number of objects with various orientations and shape. These are all regular shapes with known (perfect) orientation and size.

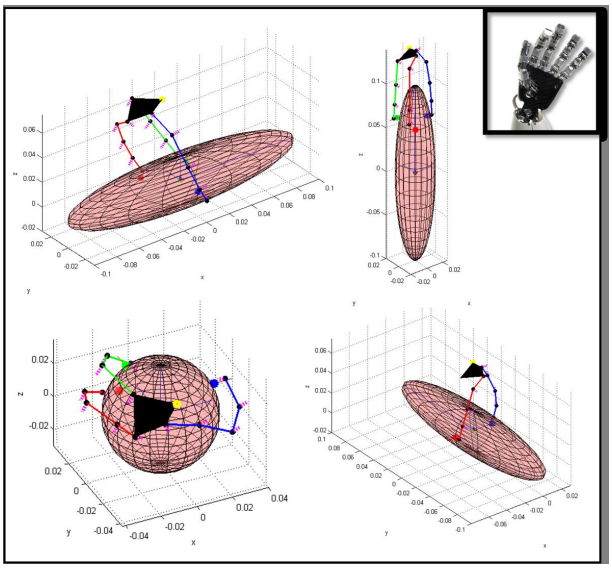


Figure 5.1: Synthetic objects and grasping simulations (benchmark 1 above).

A more realistic test is typically performed to quantify the grasping method and its failures. Here a set of objects of varying size and shapes is designed (styrofoam) and employed to evaluate the actual grasping performance 5.2 (the robot is unaided in these experiments). The results for the iCub hand are shown in the table 5.1. These show that the iCub using the proposed method (power grasp) work well in a range and elongations of the objects. This is expected and thus this benchmark constitutes the baseline to compare further development in the coming years.

Finally, a real grasping experiment testing orientation of an elongated object with varying angles (with respect to the robot) has been performed 5.3. The goal was to show that the perception of the orientation of the object works well and for objects whose size is compatible with the results of the previous paragraph, the proposed method does a good job.

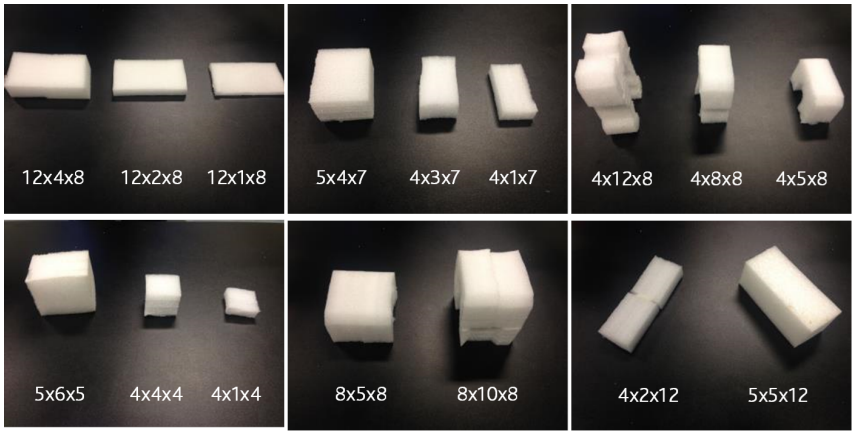


Figure 5.2: Synthetic objects but actual grasping is performed to measure the overall quality and range of the given grasping method (on the iCub in this case) - benchmark 2.

class	small	medium	large
1	0%	34%	82%
2	42%	67%	94%
3	87%	90%	85%
4	0%	78%	89%
5	n/a%	77%	72%
6	n/a%	62%	89%

Table 5.1: Power grasp evaluation on the iCub robot.

We showed these three tests for the iCub grasping (power grasp) in order to illustrate how they can be readily applied to various robot hands and employing more sophisticated grasping strategies and visual routines.

5.1.2 Learning to Grasp Objects Based On Feature Relations

SDU and UIBK are collaborating on methods to identify visual object features that predict suitability for specific grasp types. A visual feature is represented as a six-dimensional vector specifying normalized distances and angles between each pair of three ECV surface patches. Features are sampled from the region where the gripper touches the object during grasp attempts. Grasps are represented as binary outcome values (success or failure) in $SE(3)$ pose space relative to the visual features.

Bootstrapping consists in extracting features that predict success or failure of grasps attempted using the from $SE(3)$ pose parameters in question. We are experimenting with different methods for extracting structure from high-dimensional data, including maximum-margin structured learning methods and homogeneity analysis (see Deliverable D3.1.2).

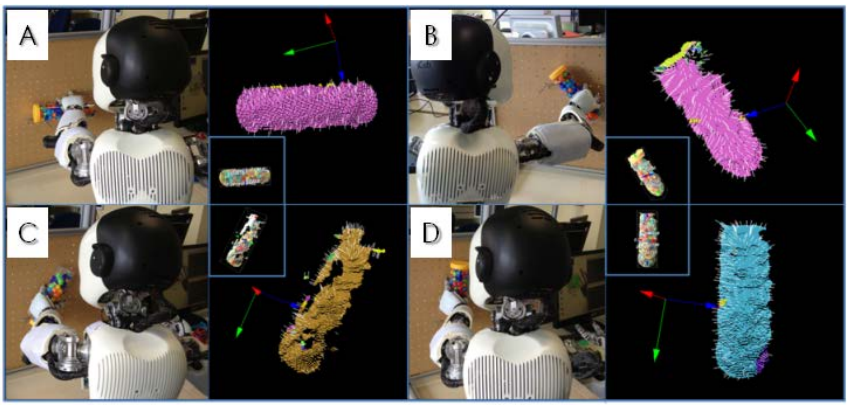


Figure 5.3: Real object grasping with various orientations - benchmark 3.

The baseline comparison without bootstrapping involves learning the graspability of individual objects without transferring learned representations between distinct objects:

Without Bootstrapping: Learn to grasp each object independently from scratch by incrementally refining its object-specific feature/success association.

With Bootstrapping: Learn to grasp one object after the other by incrementally refining a single, generic feature/success association.

We will quantify the concept in simulation by:

- comparing the number of grasp trials required to learn to grasp N different objects with a success rate of r in these two cases, and
- comparing the success rates of grasps suggested by these two learned representations applied to novel objects.

5.1.3 Transfer of manipulation predictions between actions

We will use distributions of learned particles predicting success of actions based on observed features and apply it to new actions. For a very straightforward and admittedly rather artificial example this is shown in figure 5.4.

The main idea is that we by transferring earlier acquired particles from a different yet similar action are able to improve the rate of learning on a novel action. The example shows how we transfer knowledge from an inside grasping action to a dropping into action and vice versa. Furthermore is depicts how the performance of the transferred learning compares to learning from scratch. The cost of the learning in the example is described by a percentage of data

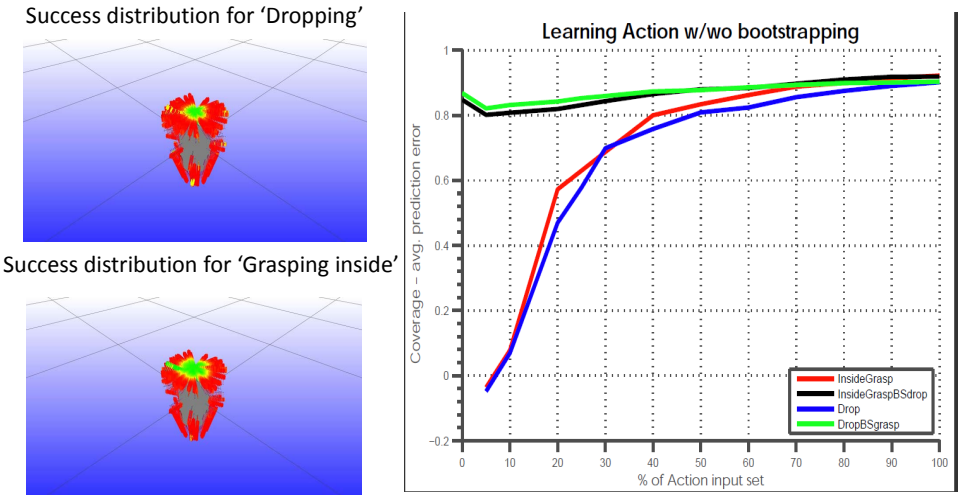


Figure 5.4: Speed up of learning actions by biasing them with already known action-success distributions

we utilize and the performance is measured as a combination of object/action coverage and success-prediction.

The bootstrapping process is introduced when we use the earlier learned action prediction as a starting point for learning a new type of action, hence we already have a prediction basis which we then can improve/correct with knowledge from the novel action. The action predictions is based on visual feature combinations and the experimental work are performed in a simulated context of dynamic action simulation as well as simulated sensors.

Without bootstrapping We learn the action prediction from scratch for every novel manipulation action.

With bootstrapping We use the earlier learned action prediction as a basis on a novel action. We then introduce particles from the new action to update the already known action prediction.

We will quantify the bootstrapping

- Comparing the success prediction of a learned action as well as the coverage.
- Comparing the amount of input data needed.

5.1.4 Accelerated Sensorimotor Learning In Constrained Domains

Through imitation a robot can obtain a first approximation of the desired movement, which can later be improved by autonomous exploration. It has been demonstrated that this way

difficult to learn motor behaviors can be acquired [MSG+96], which cannot be learned by direct mimicking. Based on such early works, reinforcement learning (RL) of robot movements has started to be seen as a viable approach to learning motor behaviors in robotics [MD01, PS08, TBS10, TNUW11, STS12]. Standard reinforcement learning focuses on how to obtain optimal task performance in a specific configuration of the robot's environment.

Thus with the application of reinforcement learning, a robot can generate new variants of the desired behavior. In Xperience project we focus on how the structure provided by previously acquired example movements can be exploited for further learning. By taking into account the structure of trajectory space defined by previously acquired example movements, the robot can bootstrap its learning process [NFV+12]. The newly acquired example trajectories can be added to the database of example movements, thereby increasing the performance of statistical generalization techniques [UGAM10, MHM11, FGMU12, MKKP13].

In the context of reinforcement learning, we are going to benchmark the speed-up in learning gained by exploiting the structure of previously acquired example movements. The following approaches will be compared:

RL without Bootstrapping: Learn a new motor task, e. g. pouring, with standard reinforcement learning techniques such as Policy learning by Weighting Exploration with the Returns (PoWER) [KP10] or Policy Improvement with Path Integrals (PI^2) [TBS10]. The reinforcement learning process will be initialized by the knowledge from one example behaviour, which can be obtained by imitation.

RL with Bootstrapping: Use reinforcement learning that exploits the structure provided by multiple example trajectories to learn the same motor task.

We expect that reinforcement learning based on structural bootstrapping will be more efficient due to the reduced dimensionality of the trajectory space spanned by previously acquired example trajectories. The following **key performance indicators** will be tested:

- Number of rollouts needed to add another trajectory to the trajectory database
- Increase in the precision of statistical generalization with respect to the original database and the database augmented by new motor patterns.

5.1.5 Action replacement versus unconstrained reinforcement learning

By exploiting higher-level information provided by semantic event chains [AAD+11], it can be suggested to replace one action with another [WGT+14]. For example, it turns out that at a semantic event chain level, stirring is similar to wiping. KIT and JSI are collaborating on how to exploit such action replacement suggestions to learn new motor behaviors. Our goal is to show that by initializing motor learning by exploration with the knowledge of one action, e. g. wiping, we can learn the appropriate motor representation of another action, e. g. stirring,

quicker than when learning from scratch. In other words, the learning of stirring motor primitive can be bootstrapped by the knowledge originating from the wiping behavior. We are going to benchmark the following types of learning

Reinforcement learning without action replacement bootstrapping: Learn a new motor task, e. g. stirring, with standard reinforcement learning techniques (see Section 5.1.4). No initial information about the task will be provided.

Reinforcement learning with action replacement bootstrapping: Exploit the motor knowledge suggested by action replacement based on semantic event chains to initialize reinforcement learning and learn the same motor task.

We are going to evaluate the speed-up that can be achieved by the proposed action replacement process. The **key performance indicator** will be the number of needed to learn a new motor task with or without bootstrapping by action replacement.

5.1.6 Learning Between Object Softness And Action Parameters

In [MDA14] we addressed the question of generative knowledge construction from sensorimotor experience acquired by exploration. We show how actions and their effects on objects, together with perceptual representations of the objects, are used to build generative models, which then can be used in internal simulation to predict the outcome of actions. Specifically,

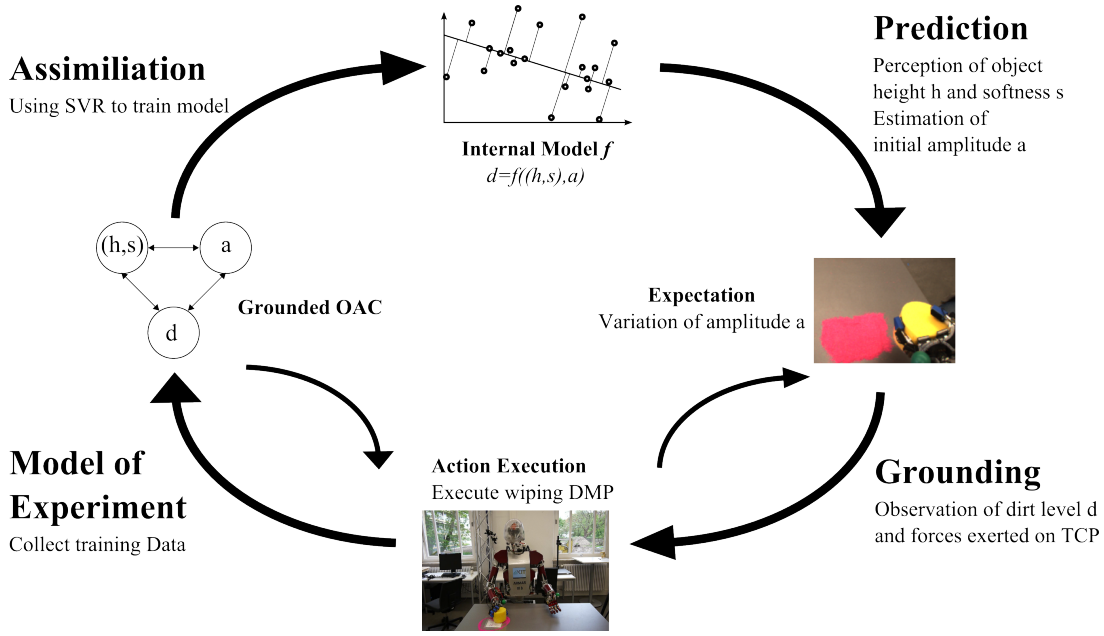


Figure 5.5: Learning correlations between object softness and action parameters for wiping in the Xperience cycle.

we addressed the learning of association between object properties (softness and height) and action parameters (DMPs parameters) for the wiping task to build generative models from sensorimotor experience resulting from wiping experiments (see Figure 5.5). Object and action are linked to the observed effect to generate training data for learning a nonparametric continuous model using Support Vector Regression. In subsequent iterations, the learned model is grounded and used to make predictions on the expected effects for novel objects and thus to constrain the action parameter exploration space. The approach has been implemented on the humanoid platform ARMAR-IIIb. Experiments with set of wiping objects differing in softness and height demonstrate efficient learning and adaptation behavior of wiping objects with different softness. We showed, that

- Increasing the number of wiping trails with different objects lead to decreasing time for action parameter search.
- **Without bootstrapping**, the robot has to test each novel object and select suitable action parameters.
- **With bootstrapping**, the robot is able to select suitable action parameters for novel objects without the exploration step.

In the future, we will extend this work in two ways: First we will consider additional object properties such as geometry, weight which allow the estimation of additional action parameters such as force, hand orientation and wiping pattern. Second, we will provide quantitative measure for the speed-up achieved by learning such generative models in the cross space of object properties and action parameters.

5.1.7 Accelerating Blind Grasping

Approaches for accelerated grasping of unknown objects can be seen as bootstrapping mechanisms. In our previous work ([SMAU13, SUA14]), we have shown how humanoid robots can leverage their capability to physically interact with the world in order to support the autonomous visual segmentation, learning and grasping of unknown objects in a cluttered environment. In [SSA12] we demonstrated a reactive grasping approach, which integrate the acquired visual object segmentation after pushing to accelerate pure haptic based object grasping (blind grasping). We showed that

- **Without bootstrapping**, the time and the number of exploration behaviors for blind grasping of unknown objects is high.
- **With bootstrapping**, the interactive visual segmentation significantly reduce the time required for grasping unknown objects.

In the future, we are going to benchmark the speed-up in blind grasping of unknown objects by integrating the interactive segmentation approaches. We will provide qualitative measures on the speed-up and relation between number of pushing actions and reactive grasping attempts.

5.2 Objects and Actions

5.2.1 Robot object manipulation database for recognition

One of the benchmarks we proposed in Y2 consists on a "classical" image database for recognition acquired in an embodied setting: i.e. considering a human-robot interaction scenario as well as considering a number of conditions (e.g. object in the hand, etc.). The database has been made Open Source and distributed at: *http://www.iit.it/en/projects/data-sets.html*.

Within the collaboration with the Laboratory for Computational and Statistical Learning (IIT@MIT) and the SLIP-GURU group at the University of Genoa (occasional collaboration), we developed a Human-Robot-Interaction (HRI) schema whereby the iCub can label images automatically exploiting self-supervision by either using knowledge of its own kinematics or visual motion cues. This knowledge provides a prior on the object location in the images which allows selecting a simple but reliable region of interest for further processing. We collect data in two modalities: human mode and robot mode (e.g. see video: http://www.youtube.com/watch?v=vhPLUNg9r5k).

There are two modalities illustrated in Figure 5.6:

- Human Mode – A human demonstrator holds the object of interest in her/his hand and moves it in front of the robot. The head of the robot tracks the object during acquisition allowing a certain degree of background variability. The object is also presented from various points of view. We determine the bounding box of the object by using an "independent motion detection algorithm" developed previously for the iCub;
- Robot Mode – The object of interest is held by the iCub and the robot moves the arm pseudo-randomly to generate a variation of points of view and backgrounds. The forward kinematics is known and it is used to determine the object bounding box.

Figure 5.6: Illustration of the two modes of acquisition for the dataset of *iCubWorld*.

We collected two different datasets:

- 10 categories, 40 objects acquired in human mode (see definition above) for the training phase. Data for the testing phase have been collected both in human and robot mode. The acquisition size is 640×480 and subsequently cropped to the bounding box of the object according to the kinematics or motion cue. The bounding box is 160×160 in humand mode and 320×320 in robot mode. For each object we provide 200 training samples. Each category is trained with three objects (600 examples per category);
- This is the first release of the *iCubWorld* dataset. It consists of seven instances of objects acquired in the two different modalities: human and robot as defined earlier. The size of the images is 320×240 subsequently cropped to the bounding box size according to the following:
  - Human mode: the bounding box is set to 80×80;
  - Robot mode: the bounding box is set to 160×160.

  The kinematics of the robot is known and used to position the bounding box. The independent motion detector method is used to position the bounding box in the human mode. We provide 500 images per class during the training phase and 500 images per class for the testing phase.

An example of the images contained in *iCubWorld1.0* is shown in figure 5.7.

Figure 5.7: Sample images of the *iCubWorld1.0* dataset.

These standard datasets are relatively different from other image centered databases since we strove to collect data in realistic HRI experiments which are different from simple collections of images taken by a human photographer (e.g. centered, in focus, etc.). In *iCubWorld* we have to deal with blur, with considerable clutter, occlusions, etc. thus providing a better evaluation of actual performance in the robotic/HRI setting.

5.2.2 Inferring Object Affordances By Generalizing From Experience

The methods UIBK is developing for grasp generalization (cf. Sec. 5.1.2) apply more generally to inference between attributes of objects, actions, etc. In Xperience Scenario [**?**], also described in a draft journal publication [WGT+14], we will put it to use to provide a *Repository of Objects and Attributes with Roles* (ROAR).

The relevant part of the scenario proceeds as follows: The robot has encountered, in different contexts, the activities of wiping (with a sponge) and of mixing (with a spoon). These two activities are described by the same SECs. Moreover, both share approximately the same circular hand motion. Thus, the robot (wrongly) infers that sponges and spoons are equivalent.

Now, the robot is tasked with mixing batter, but does not find a spoon in its workspace.

With bootstrapping (ROAR): The robot checks the ROAR to verify the hypothesis that a sponge can serve as a substitute for a spoon in mixing, and finds that the sponge, being soft, is clearly outside the ROAR cluster of objects known to be useful for mixing.

It then checks the ROAR for other objects it finds in its workspace, and finds that the fork is located near objects known by the ROAR to be useful for mixing.

The ROAR's performance can be quantified without an end-to-end robotic implementation in the following manner:

1. Create a fully annotated, realistic, ground-truth data set G of objects with attributes.
2. Do K times:
   (a) Create ROAR R_k by populating it with G, erasing a random subset of attributes.
   (b) For each object in the ROAR, check whether its utility for mixing is correctly predicted.
3. The ROAR's performance is the average of the correct-prediction rates over all objects in the ROAR.

To test the stability of the ROAR with respect to the population of objects, the above procedure can be repeated for various subsets of G.

Without bootstrapping (no ROAR): The robot attempts to mix the batter with the sponge, but fails to achieve the desired result. All it can do is proceed trying the other objects in its workspace, one after the other, until it finds an object useful for mixing.

The system performance without ROAR is then simply given by the proportion of objects in the robot's workspace that are in fact useful for mixing.

5.2.3 Learning Pairwise Affordances

Many actions involve pairs of objects. Understanding the effect of actions involving pairs of objects constitutes the next step beyond acting on a single object. We hypothesize that knowl-

edge about how these objects behave under single-object actions help us predict how they will behave under dual-object actions.

To quantify this, UIBK's robot is performing exploratory actions on single objects (poking from the top, the front, or the side; releasing them above the table) while observing their effects (rolled, pushed, toppled, no effect). Then, it releases one object above the other and observes the effect (inserted, stacked, toppled, covered).

Learning to predict the effect of the latter, two-object action by exhaustive experimentation is expensive, involving on the order of N^2 experiments. This is exacerbated if the goal of learning is not simply to predict the outcome for pairs of specific, known objects, but for new objects based on features known about them (e.g., features of shape). However, if these features include their single-object affordances such as the above, learning the dual-object affordances should require much less training.

We are currently performing experiments to quantify the difference in training complexity between learning these dual-object affordances with and without including the single-object affordances in the feature vector describing each object.

5.3 Planning and Language

The work at UEDIN falls roughly into three areas:

1. learning actions,
2. recognizing and learning plans, and
3. learning language.

We discuss benchmarking in each of these areas individually below.

5.3.1 Learning Actions

There is no standard benchmark for comparing different planning domain models. We have used domains from the International Planning Competition (http://ipc.icaps-conference.org/), which are commonly used when benchmarking planners and are widely known and available. To compare the performance of different domain models we have used F-score, error rate [ZYHL10] and variational distance [PZK07]:

- The F-score is the harmonic mean of precision and recall (true positives/predicted changes and true positives/actual changes, respectively). It was calculated by designating the number of correctly predicted changes to the fluents as the true positives, the number of changes which were not predicted as the false negatives, and the number of incorrectly predicted changes as false positives.

- The error rate for a single action is defined as the number of extra or missing fluents in the preconditions and effects (E_{pre} and E_{eff} respectively) divided by the number of possible fluents in the preconditions and effects (T): $Error(a) = \frac{1}{2T}(E_{pre} + E_{eff})$. The error rate of a domain model with a set of actions A is: $Error(A) = \frac{1}{|A|}\sum_{a \in A} Error(a)$.
- Variational distance is more useful than the above measures when comparing probabilistic domain models. It compares the true model P with the learnt model $\hat{P}$ over a set of examples E. The variational distance between two domain models is the average difference between the probability of an example occurring assigned by the true model and the probability assigned by the learnt model: $VD(P,\hat{P}) = \frac{1}{|E|}\sum_{e \in E} |P(e) - \hat{P}(e)|$.

5.3.2 Recognizing and Learning Plans

Our research on recognizing plans in the Xperience area is very much related to work on parsing in natural language and work on plan recognition. As such, benchmarking our systems will use well known metrics from the plan recognition literature including accuracy, precision, and mean time to recognition[BA06]. To provide a baseline values for these metrics we anticipate using either other well known domains from the plan recognition literature or synthetic domains that have known properties. Note that in some instances synthetically generated plan libraries are able to control form various factors and provide a level of systematic study that using real world domains cannot.

To benchmark the learning of plans will require looking at two different aspects of the learned plans, recognition of further instances of the plan and generation of plans for future use. We anticipate using different metrics and benchmarking for these tasks.

For benchmarking recognition of newly learned plans we anticipate using the same metrics as described for recognition in the domain in general namely: accuracy, precision, recall, and mean time to recognition of new instances of the newly learned plans. As new plans are learned the system's performance on these critical metrics should not be significantly decreased. We would anticipate some change, but a significant decrease would be cause for concern.

To benchmark the effectiveness of newly learned plans for plan generation requires considering the speed of planning, that is average time to construct a successful plan. For cases where the system was previously unable to construct a plan (like the current Xperience wiping vs mixing example) the value of the learned plan is obvious and can be benchmarked as the value of executing the plan at all.

However, we can also imagine more general cases where the system is able to find a plan but the newly learned plan should be an improvement over the known plan. In order for such a benefit to be quantifiable it must be the case that either executing the learned plan results in a preferred outcome (higher utility) or the system is able to generate a plan of the same quality faster. In this case we might view what is going on a "speedup learning"[Fer10].

5.3.3 Learning Language

In terms of UEDIN's work on language learning there are a number of different benchmarks that we evaluated our work against. [TS11] Semi-supervised CCG Lexicon Extension was evaluated against the baseline unextended parser and against other learning techniques (self training) on a standard benchmark dataset (the CCG version of) the Penn Wall Street Journal Treebank[MSM93].

[KGZS12b] A Probabilistic Model of Syntactic and Semantic Acquisition from Child-Directed Utterances was evaluated on a new publicly available benchmark training set derived from the standard CHILDES Eve corpus. Ours is the only work we know of using this dataset in any form, but we evaluated our model in comparison to [KZGS10] on the same benchmark, The latter is a state of the art semantic parser learner which was in turn evaluated on the standard Geoqueries benchmark datasets[TM03] in comparison with a number of other learners.

[KLP$^+$14] Extracting Common Sense Knowledge from Text for Robot Planning is evaluated against human evaluations and against the baseline planner, but we are not aware of any available benchmark dataset.

Bibliography

[AAD+11] E. E. Aksoy, A. Abramov, J. Dorr, K. Ning, B. Dellen, and Florentin Wörgötter. Learning the semantics of object-action relations by observation. *The International Journal of Robotics Research*, 30(10):1229–1249, 2011.

[ASK08] N Ando, T Suehiro, and T Kotoku. A software platform for component based rt-system development: OpenRTM-Aist. *Simulation, Modeling, and Programming for Autonomous Robots*, pages 87–98, 2008.

[BA06] Nate Blaylock and James Allen. Fast hierarchical goal schema recognition. In *Proceedings of AAAI-2006*, pages 796–801. AAAI Press, 2006.

[BC10] J. Bohren and S. Cousins. The SMACH High-Level Executive [ROS News]. *Robotics Automation Magazine, IEEE*, 17:18–20, 2010.

[BECHR10] David Billington, Vladimir Estivill-Castro, RenÃl' Hexel, and Andrew Rock. Modelling behaviour requirements for automatic interpretation, simulation and deployment. In *SIMPAR*, volume 6472 of *Lecture Notes in Computer Science*, pages 204–216. Springer, 2010.

[Bru01] H Bruyninckx. Open robot control software: the OROCOS project. In *Robotics and Automation, 2001. Proceedings 2001 ICRA. IEEE International Conference on*, volume 3, pages 2523–2528. IEEE, 2001.

[BS09] Davide Brugali and Patrizia Scandurra. Component-based Robotic Engineering Part I : Reusable building blocks. XX(4):1–12, 2009.

[BS10] Davide Brugali and Azamat Shakhimardanov. Component-based Robotic Engineering Part II : Systems and Models. XX(1):1–12, 2010.

[BSK03] Herman Bruyninckx, Peter Soetens, and Bob Koninckx. The real-time motion control core of the Orocos project. In *IEEE International Conference on Robotics and Automation (ICRA)*, pages 2766–2771, 2003.

[DSEA14] 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)*, pages 0–0, 2014.

[EHW+92] Oren Etzioni, Steve Hanks, Daniel Weld, Denise Draper, Neal Lesh, and Mike Williamson. An approach to planning with incomplete information. In William Swartout, Bernhard Nebel, and Charles Rich, editors, *Proceedings of the Third International Conference on Principles of Knowledge Representation and Reasoning (KR-92)*, pages 115–125, Cambridge, MA, October 1992. Morgan Kaufmann Publishers.

[Fer10] Alan Fern. Speedup learning. In Claude Sammut and Geoffrey I. Webb, editors, *Encyclopedia of Machine Learning*, pages 907–911. Springer, 2010.

[FGMU12] D. Forte, A. Gams, J. Morimoto, and A. Ude. On-line motion synthesis and adaptation using a trajectory database. *Robotics and Autonomous Systems*, 60:1327–1339, 2012.

[FN71] Richard E. Fikes and Nils J. Nilsson. STRIPS: A new approach to the application of theorem proving to problem solving. *Artificial Intelligence*, 2:189–208, 1971.

[fxe] FxEngine: A Framework for data flow processing and the design of dynamic systems using plugins.

[GB04] Erich Gamma and Kent Beck. *Contributing to Eclipse: principles, patterns, and plug-ins*. Addison-Wesley Professional, 2004.

[GKK13] F. Guerin, N. Kruger, and D. Kraft. A survey of the ontogeny of tool use: From sensorimotor experience to planning. *Autonomous Mental Development, IEEE Transactions on*, 5(1):18–45, March 2013.

[GM12] A. Gijsberts and G. Metta. Real-time model learning using incremental sparse spectrum gaussian process regression. *Neural Networks*, pages 59–69, 2012.

[GPL07] M Giorgio, F Paul, and N Lorenzo. Towards Long-Lived Robot Genes. *Elsevier*, 2007.

[GPTM13a] I. Gori, U. Pattacini, V. Tikhanoff, and G. Metta. A comprehensive method for power grasp on incomplete point clouds. In *Robotics Science and Systems Workshop: Manipulation with Uncertain Models*, Berlin, Germany, 2013.

[GPTM13b] I. Gori, U. Pattacini, V. Tikhanoff, and G. Metta. Ranking the good points: A comprehensive method for humanoid robots to grasp unknown objects. In *International Conference on Advanced Robotics*, Montevideo, Uruguay, November 25–29, 2013.

[Har87] David Harel. Statecharts: A visual formalism for complex systems. *Sci. Comput. Program.*, 8(3):231–274, June 1987.

[IDC96] Roberto Ierusalimschy, Luiz Henrique De Figueiredo, and Waldemar Celes Filho. Lua-an extensible extension language. *Softw., Pract. Exper.*, 26(6):635–652, 1996.

[Inc] ZeroC Inc. Internet Communications Engine.

[JLJ+10] C Jang, S I Lee, S W Jung, B Song, R Kim, S Kim, and C H Lee. Opros: A new component-based robot software platform. *ETRI journal*, 32(5):646–656, 2010.

[KB12] Markus Klotzbücher and Herman Bruyninckx. Coordinating Robotic Tasks and Systems with rFSM Statecharts. *JOSER: Journal of Software Engineering for Robotics*, pages 28–56, 2012.

[KGZS12a] T. Kwiatkowski, S. Goldwater, L. Zettlemoyer, and M. Steedman. A probabilistic model of syntactic and semantic acquisition from child-directed utterances and their meanings. In *Conference of the European Chapter of the Association for Computational Linguistics (EACL)*, 2012.

[KGZS12b] Tom Kwiatkowski, Sharon Goldwater, Luke Zettlemoyer, and Mark Steedman. A probabilistic model of syntactic and semantic acquisition from child-directed utterances and their meanings. In *Proceedings of the 13th Conference of the European Chapter of the ACL (EACL 2012)*, pages 234–244, Avignon, 2012. ACL.

[KLP+14] Peter Kaiser, Mike Lewis, Ronald Petrick, Tamim Asfour, and Mark Steedman. Extracting common sense knowledge from text for robot planning. In *International Conference on Robotics and Automation (ICRA)*, pages XXX–XXX. IEEE, 2014. to appear.

[KP10] J. Kober and J. Peters. Policy search for motor primitives in robotics. *Machine Learning*, 84(1-2):171–203, 2010.

[KPP+11] N. Krüger, C. Geib J. Piater, R. Petrick, M. Steedman, F. Wörgötter, A. Ude, T. Asfour, D. Kraft, D. Omrcen, A. Agostini, and R. Dillmann. Object-action complexes: Grounded abstractions of sensorimotor processes. *Robotics and Autonomous Systems*, 59:740–757, 2011.

[KZGS10] Tom Kwiatkowski, Luke Zettlemoyer, Sharon Goldwater, and Mark Steedman. Inducing probabilistic CCG grammars from logical form with higher-order unification. In *Proceedings of the Conference on Empirical Methods in Natural Language Processing*, pages 1223–1233, Cambridge, MA, 2010. ACL.

[MD01] J. Morimoto and K. Doya. Acquisition of stand-up behavior by a real robot using hierarchical reinforcement learning. *Robotics and Autonomous Systems*, 36:37–53, 2001.

[MHM11] T. Matsubara, S.-H. Hyon, and J. Morimoto. Learning parametric dynamic movement primitives from multiple demonstrations. *Neural Networks*, 24(5):493–500, 2011.

[MKKP13] K. Mülling, J. Kober, O. Kroemer, and J. Peters. Learning to select and generalize striking movements in robot table tennis. *International Journal of Robotics Research*, 32(3):263–279, 2013.

[MRW06] Torsten Merz, Piotr Rudol, and Mariusz Wzorek. Control system framework for autonomous robots based on extended state machines. In *ICAS*, page 14. IEEE Computer Society, 2006.

[MSG+96] H. Miyamoto, S. Schaal, F. Gandolfo, H. Gomi, Y. Koike, R. Osu, E. Nakano, Y. Wada, and M. Kawato. A kendama learning robot based on bi-directional theory. *Neural Networks*, 9(8):1281–1302, 1996.

[MSM93] Mitch Marcus, Beatrice Santorini, and M. Marcinkiewicz. Building a large annotated corpus of English: The Penn Treebank. *Computational Linguistics*, 19:313–330, 1993.

[MSV08] Giorgio Metta, G Sandini, and David Vernon. The iCub humanoid robot: an open platform for research in embodied cognition. *Proceedings of the 8th workshop on performance metrics for intelligent systems*, pages 50—-56, 2008.

[Mur95] Mario Murata, Kenichi and Horspool, R Nigel and Manning, Eric G and Yokote, Yasuhiko and Tokoro. Unification of active and passive objects in an object-oriented operating system. *Object-Orientation in Operating Systems, 1995., Fourth International Workshop on*, pages 68–71, 1995.

[NFV+12] B. Nemec, D. Forte, R. Vuga, M. Tamošiunaite, F. Wörgötter, and A. Ude. Applying statistical generalization to determine search direction for reinforcement learning of movement primitives. In *2012 12th IEEE-RAS Int. Conf. Humanoid Robots*, pages 65–70, Osaka, Japan, 2012.

[OMG08] OMG. Common Object Request Broker Architecture (CORBA/IIOP).v3.1. Technical report, OMG, January 2008.

[PB02] Ronald P. A. Petrick and Fahiem Bacchus. A knowledge-based approach to planning with incomplete information and sensing. In Malik Ghallab, Joachim Hertzberg, and Paolo Traverso, editors, *Proceedings of the Sixth International Conference on Artificial Intelligence Planning and Scheduling (AIPS-2002)*, pages 212–221, Menlo Park, CA, April 2002. AAAI Press.

[PB04] Ronald P. A. Petrick and Fahiem Bacchus. Extending the knowledge-based approach to planning with incomplete information and sensing. In Shlomo Zilberstein, Jana Koehler, and Sven Koenig, editors, *Proceedings of the International Conference on Automated Planning and Scheduling (ICAPS-04)*, pages 2–11, Menlo Park, CA, June 2004. AAAI Press.

[Pet13] Ronald P. A. Petrick. An application programming interface to high-level planning. Technical Report Internal Technical Report, University of Edinburgh, 2013.

[Pet14] Ronald P. A. Petrick. Approximate reasoning with numeric fluents for contingent planning. In *AAAI Conference on Artificial Intelligence.*, 2014. submitted.

[plu] Pluma: an open source C++ framework for plug-in management.

[PMN] Ali Paikan, Giorgio Metta, and Lorenzo Natale. A port-arbitrated mechanism for behavior selection in humanoid robotics. *The 16th International Conference on Advanced Robotics*.

[PS08] J. Peters and S. Schaal. Reinforcement learning of motor skills with policy gradients. *Neural Networks*, 21:682–697, 2008.

[PZK07] Hanna M. Pasula, Luke S. Zettlemoyer, and Leslie P. Kaelbling. Learning symbolic models of stochastic domains. *JAIR*, 29:309–352, 2007.

[QCG+09] Morgan Quigley, Ken Conley, Brian P. Gerkey, Josh Faust, Tully Foote, Jeremy Leibs, Rob Wheeler, and Andrew Y. Ng. Ros: an open-source robot operating system. In *ICRA Workshop on Open Source Software*, 2009.

[QGC+09] M Quigley, B Gerkey, K Conley, J Faust, T Foote, J Leibs, E Berger, R Wheeler, and A Ng. ROS: an open-source Robot Operating System. *Science*, 2009.

[Sam97] Johannes Sametinger. *Software Engineering with Reusable Components*. Springer Berlin Heidelberg, Berlin, Heidelberg, 1997.

[SMAU13] D. Schiebener, J. Morimoto, T. Asfour, and A. Ude. Integrating visual perception and manipulation for autonomous learning of object representations. *Adaptive Behavior*, 21(5):328–345, 2013.

[SSA12] David Schiebener, Julian Schill, and Tamim Asfour. Discovery, segmentation and reactive grasping of unknown objects. In *Proc. 12th IEEE-RAS International Conference on Humanoid Robots*, Osaka, Japan, 2012.

[STS12] F. Stulp, E. Theodorou, and S. Schaal. Reinforcement learning with sequences of motion primitives for robust manipulation. *IEEE Transactions on Robotics*, 28(6):1360–1370, 2012.

[SUA14] D. Schiebener, A. Ude, and T. Asfour. Physical interaction for segmentation of unknown textured and non-textured rigid objects. In *IEEE International Conference on Robotics and Automation (ICRA)*, Hong Kong, 2014.

[TBS10] E. A. Theodorou, J. Buchli, and S. Schaal. A generalized path integral control approach to reinforcement learning. *Journal of Machine Learning Research*, 11:3137–3181, 2010.

[TM03] Cynthia Thompson and Raymond Mooney. Acquiring word-meaning mappings for natural language interfaces. *Journal of Artificial Intelligence Research*, 18:1–44, 2003.

[TNUW11] M. Tamosiunaite, B. Nemec, A. Ude, and F. Wörgötter. Learning to pour with a robot arm combining goal and shape learning for dynamic movement primitives. *Robotics and Autonomous Systems*, 59(11):910–922, 2011.

[TPNM13] V Tikhanoff, U Pattacini, L Natale, and G Metta. Exploring affordances and tool use on the iCub. *IEEE-RAS International Conference on Humanoid Robots*, 2013.

[TS11] Emily Thomforde and Mark Steedman. Semi-supervised CCG lexicon extension. In *Proceedings of the Conference on Empirical Methods in Natural Language Processing*, pages 1246–1256. ACL, 2011.

[UGAM10] A. Ude, A. Gams, T. Asfour, and J. Morimoto. Task-specific generalization of discrete and periodic dynamic movement primitives. *IEEE Transactions on Robotics*, 26(5):800–815, 2010.

[WD06] Reinhard Wolfinger and Deepak Dhungana. A component plug-in architecture for the. NET platform. *Modular Programming ...*, pages 1–20, 2006.

[WGT+14] Florentin Wörgötter, Chris Geib, Minija Tamosiunaite, Eren Erdal Aksoy, Justus Piater, Hanchen Xiong, Ales Ude, Bojan Nemec, Dirk Kraft, Norbert Krüger, Mirko Wächter, and Tamim Asfour. Structural bootstrapping – a novel concept for the fast acquisition of action-knowledge. *Transaction of Autonomous Mental Development*, 2014. Submitted.

[WSA+13] M. Wächter, S. Schulz, T. Asfour, E. Aksoy, F. Wörgötter, and R. Dillmann. Action sequence reproduction based on automatic segmentation and object-action complexes. In *IEEE/RAS International Conference on Humanoid Robots (Humanoids)*, pages 0–0, 2013.

[WVW+13] K. Welke, N. Vahrenkamp, M. Wächter, M. Kroehnert, and T. Asfour. The armarx framework - supporting high level robot programming through state disclosure. In *INFORMATIK Workshop*, pages 0–0, 2013.

[Zen04] Matthias Zenger. *Programming language abstractions for extensible software components*. PhD thesis, ÉCOLE POLYTECHNIQUE FÉDÉRALE DE LAUSANNE, 2004.

[zer] ZeroMQ: The intelligent transport layer.

[ZYHL10] Hankz H. Zhuo, Qiang Yang, Derek H. Hu, and Lei Li. Learning complex action models with quantifiers and logical implications. *Artificial Intelligence*, 174(18):1540–1569, December 2010.