Chapter 7

Embodiment abstraction

The architecture presented in Chapter 6 allows us to port services, primitives and tasks to other platforms with a reasonable effort, however it is not possible to execute the same task description on two different embodiments and plans cannot be shared without manually modifying or tuning them.

The aim of the work presented in this chapter is to provide a tool that allows to use the same task descriptions regardless of the robot used. In this way, it is possible that a task learned by a robot could be transferred to another robot or that a manually designed task could be executed by different robots with different embodiments.

In this chapter we present a hardware abstraction mechanism built on top of the manipulation primitives paradigm presented in Chapter 3 that complements the software architecture presented in Chapter 6 and allows the same task description to be used across different hardware platforms. We demonstrate the abstraction mechanism by executing the same task description on two different robots with different arms and grippers.

7.1 Introduction

The approach extends the work presented in Chapter 3 by introducing an embodiment independent abstraction mechanism on top of the manipulation primitives paradigm. The abstraction offers several advantages. Firstly, complex actions can be described in terms of simple abstract primitives. Secondly, plans can be shared over different embodiments because the vocabulary of primitives is shared. Thirdly, manipulation primitives offer to high-level planners a vocabulary of reliable actions onto which build manipulation tasks and plans, thus simplifying and robustifying planning. Finally, these abstract models can be translated to embodiment specific models, constituting of reactive sensor-based controllers, such that the full capabilities of each platform can be utilised.

Task A cyclic, directed, connected and labelled multi-graph where the nodes correspond to *manipulation primitives* or other *tasks* and the edges to *events*.

Plan An instance of a *task* with the parameters set for a specific execution in a determined scenario and context.

Manipulation primitive A reactive controller, designed to perform a specific primitive action on a particular embodiment.

Abstract primitive An embodiment independent primitive action that can be translated to a manipulation primitive. It has required and optional parameters used to adapt the primitive behaviour to the specific action and will be used during the translation process.

Events Represent the detection of a specific perceptual or internal condition.

To flesh out the abstract primitives, in Chapter 3 we presented a complete set of reactive manipulation primitives for an arm manipulator equipped with a wrist force sensor and a three-fingered hand equipped with tactile sensors. We look into the power of the embodiment independent abstract primitives in the scenario where two different platforms with different hardware capabilities are used to complete a manipulation task using the same abstract description. A primitive based vocabulary is an effective way of transferring knowledge and plans between embodiments.

7.1.1 Related work

Few studies have addressed the issue of abstracting hardware from action. [Petersson et al., 1999] presented a somewhat similar framework but to our knowledge that framework has never been demonstrated in practice with multiple embodiments. An earlier version of the framework presented here appeared in [Laaksonen et al., 2010]. [Ellenberg et al., 2010] studied how algorithms for humanoid robot walking can be transferred between embodiments. The RoboEarth project has proposed a web platform for sharing environment models as well as action "recipes" between multiple robots using Ontology Web Language (OWL) [Tenorth et al., 2012].

From another perspective, Programming by demonstration (PbD) instead of focusing on transferring plans between robots, focuses on transferring skills from a human to a robot. However, the problems that need to be solved by PbD (a.k.a. Imitation Learning) include the task generalization problem, where the demonstrated task has to be described using a general representation that can be grounded on a robot. The task representations used by PbD include symbolic representations, sensorimotor representations or machine learning tools such as Artificial Neural Networks (ANNs), Radial Basis Functions (RBFs) or Hidden Markov Models (HMMs) [Billard et al., 2008].

Human action detection and understanding provides tools for abstract action representation. Regarding the symbolic and semantic representations used in this context, [Yang

Abstract primitive	*Parameters*	*Meaning*
move	target, *trajectory*, move type	Move without object.
transport	target, *trajectory*, move type	Move with object.
place	target, *trajectory*, move type	Place down object.
push	target, *trajectory*, move type	Push object.
grasp	*preshape, object size*	Grasp object.
release	*hand opening*	Release object.

Table 7.1: Abstract primitives and parameters (optional parameters in *italic).*

et al., 2015] propose a framework for learning the semantics of manipulation actions where using a Combinatory Categorical Grammar (CCG) based learning models, the manipulation plans can be obtained from a video sequence. However, after obtaining the task definitions, it is still future work to parametrize each atomic action to address a specific scenario and execute the task on a real robot.

7.2 Embodiment independence through abstraction

For the definition of abstract tasks, the same mechanisms proposed in Chapter 3 can be used. Tasks are composed of manipulation primitives connected by events. However, manipulation primitives and events are embodiment specific. For the implementation of the abstraction mechanism we have defined the concept of abstract primitive: a semantically meaningful primitive action that can be translated to an embodiment specific manipulation primitive. Like the manipulation primitives, abstract primitives are also configurable using parameters. There are *required* parameters that need to be specified for the primitive to work and *optional* parameters that are used to provide additional information to the underlying controllers.

The set of abstract primitives proposed in this work is shown in Table 7.1. This set of abstract actions allows moving objects by grasping or pushing them and has been found to support many common manipulation actions.

When the primitives are translated to an embodiment specific manipulation primitive, the required parameters which describe constraints need to be fulfilled but the optional parameters can be ignored if necessary as their purpose is to serve as hints how to perform the task.

All primitives except *grasp* and *release* are related to arm motions. The required parameters for these primitives define the target pose to move the arm to and the type of the motion. The motion target can be a single waypoint or a trajectory represented as a set of waypoints. However, defining a strict trajectory to be followed should be avoided when not made necessary by the task to allow each embodiment to use its own capabilities in the best possible way. Defining only the end pose is usually sufficient from the task perspective and leaves the freedom for the embodiment to choose a collision free path for that particular embodiment.

Abstract event	*Meaning*
success	Primitive successfully completed.
grasp_stable	Stable grasp detected.
grasp_lost	Grasp loss detected.
timeout	Timeout for specified time.
hardware_failure	Hardware failure detected.
error	Generic error.

Table 7.2: Abstract events.

In addition to the motion target, the type of motion is specified. Supported motion types include free, guarded, and constrained motions. In free motion, the embodiment is free to use any path to reach the target. In a guarded motion, the embodiment is required to use a Cartesian straight line path. In a constrained motion, rotational degrees of freedom can be constrained to remain the same for the duration of the motion. This is useful, for example, to transport containers with liquid. The underlying idea in the required parameters is thus to constrain the effects of a primitive rather than the ways to achieve them.

The grasp primitive allows to use an optional parameter to choose the grasp preshape and the object size. Because the parameters are optional, they can be ignored by platforms which do not support a particular grasp type. In that case, the primitive is likely to be translated to the closest possible grasp available.

To allow the embodiment independent description of tasks, the state transitions need also to be described in an abstract fashion. This is done using abstract events shown in Table 7.2. The events are related to completing a primitive successfully (*success*), grasp stability (*grasp_stable, grasp_lost*), and failure conditions (*timeout, hardware_failure, error*). Each platform is again free to use the available sensor set in any possible way to detect these events.

It should be noted that the primitives and events at the abstract level are not coupled to any particular embodiment. An important note here is that the sets of abstract primitives and events need to be rich enough in order to allow wide use of sensors in the embodiment specific controllers, while at the same time it is important to keep the semantic meanings of the abstract entities clear to allow the mapping between abstract and platform specific sensor events and manipulation primitives.

7.2.1 Abstract Task Description

The abstract task description (ATD) is a hardware independent description of a manipulation task. As in Chapter 3, tasks are defined as cyclic, directed, connected and labelled multi-graphs where the nodes correspond to abstract primitives or abstract tasks and the edges to abstract events. The definitions of task, plan, event and manipulation primitives are detailed in Sec. 3.2. XML (eXtensible Markup Language) is used to describe the relevant information, such as the abstract primitives and the tran-

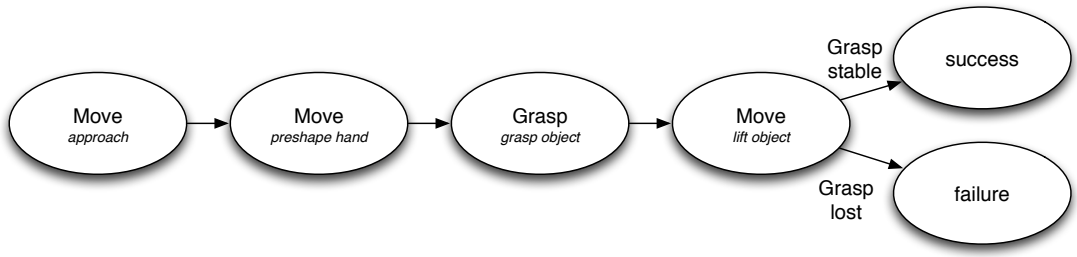


Figure 7.1: An example abstract task definition, describing a simple grasp and lift manipulation task. Some of the elements have been left out for brevity, e.g. properties of the object and some of the common edges, e.g. timeout to the failure state.

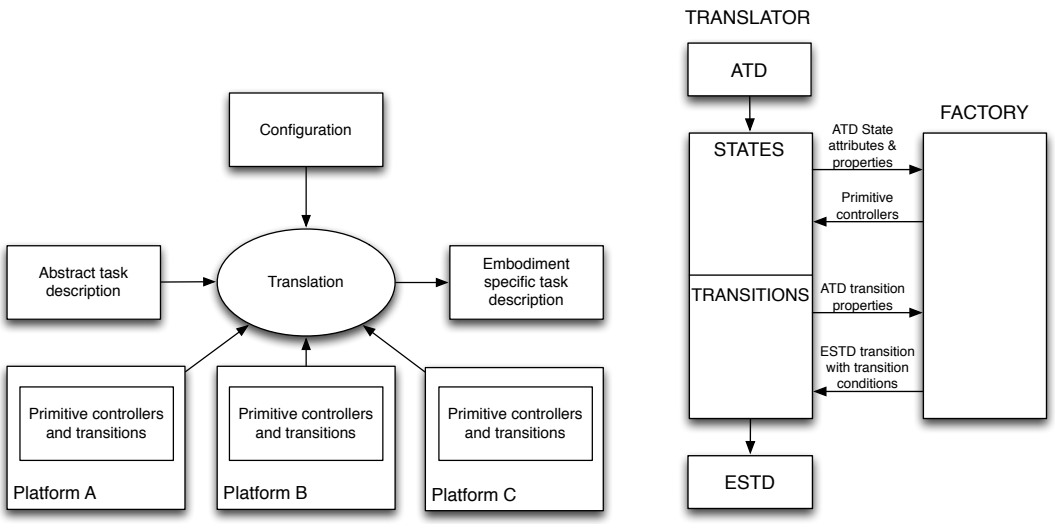
```
<statemachine>
  <state name="approach" type="move">
    <movement>free</movement>
    <hand_shape>open</hand_shape>
   </state>
  <state name="preshape_hand" type="move">
    <movement>guarded</movement>
    <hand_shape>pinch_grasp_preshape</hand_shape>
  </state>
  <state name="grasp_object" type="grasp">
    <movement>guarded</movement>
    <hand_shape>pinch_grasp</hand_shape>
  </state>
  <state name="lift_object" type="transport">
    <movement>guarded</movement>
    <hand_shape>pinch_grasp</hand_shape>
    <path>
      <position>0.2 0.6 0.25</position>
    </path>
  </state>
  <state name="success_end" type="success">
  </state>
  <state name="fail_end" type="failure">
  </state>
```

```
  <transition origin="approach"
   destination="preshape_hand">
    <success/>
  </transition>
  <transition origin="preshape_hand"
   destination="grasp_object">
    <success/>
  </transition>
  <transition origin="grasp_object"
   destination="lift_object">
    <success/>
    <grasp_stable/>
  </transition>
  <transition origin="lift_object"
   destination="fail_end">
  <grasp_lost/>
  </transition>
  <transition origin="lift_object"
   destination="success_end">
    <success/>
    <grasp_stable/>
  </transition>
</statemachine>
```

Figure 7.2: XML definition of the Abstract Task Description shown in Fig. 7.1.

sitions triggered by abstract events. In addition to nodes and edges, information about the environment such as obstacles and the location, mass and approach direction to the target object are included in the abstract task description. All the properties and definitions in XML are hardware independent.

The abstract task is described through definition of nodes and edges. Both nodes and edges have properties that can be used to further inform of the intended action. The most important node property is its *type*, corresponding to one of the primitives introduced above or the special states *"success"*, or *"failure"*. The two latter types indicate end states of a task with either success or failure reported to the higher level controller. In addition, the parameters of the primitives are specified as node properties. For example, the hand preshape for grasping or the target position of the end-effector can be set through node properties. The edge properties describe the set of abstract events which



(a) Translation process

(b) Relationship and communication of the translator and factory

Figure 7.3: Translation process and components.

trigger the node transition. For example, the loss of a grasp can trigger a transition to another node.

The attributes are the key factor in selecting the manipulation primitives during the translation process explained in Sec. 7.2.2. An example abstract task and its XML definition, describing a simple grasp and lift manipulation, are shown in Figs. 7.1 and 7.2. Some of the elements have been left out for brevity, e.g. properties of the object and some of the common edges, e.g. timeout to the failure state. It should be noted that the task description does not need to be a sequence or a tree but it can be any directed graph, however, the simple form in the example is used to limit the size of the associated XML code shown.

7.2.2 Translation from ATD to ESTD

The translation process connects the Abstract Task Definition (ATD) and the Embodiment Specific Task Definition (ESTD). The translation takes the abstract task definition as an input, and translates it into an embodiment specific task definition. The high-level translation process is depicted in Fig. 7.3(a).

As can be seen in Fig. 7.3(a), the translation component needs input defining the configuration of the translation process, i.e., the target platform and the platform specific events and manipulation primitives used directly in the embodiment specific task description. The benefit of this arrangement is that the only hardware dependent blocks

shown in the figure are the manipulation primitives and events that are platform specific. The critical requirement of real-time operation for sensor-based control is also fulfilled as the embodiment specific task can be run as is, without any additional overhead from maintaining hardware independence.

The translation process requires a mapping component which produces the embodiment specific task description from the abstract description. In our case the mapping component is constructed from two sub-components, shown in Fig. 7.3(b). The first part, *translator*, consists of necessary book-keeping and the internal logic that is independent from the embodiment. The translator also constructs the final ESTD which is used to execute the desired abstract task. The second part, *factory*, handles the embodiment specific construction of the nodes and edges of the ESTD, i.e., the sensor based manipulation primitives and transition conditions. This division was made to reduce the implementation time of the factory, which needs to be implemented for each different embodiment. The relation and the communication between these two sub-components are shown in Fig. 7.3(b). The factory also receives object and environment information of the ATD in addition to a particular node or event and its properties. This gives the factory the complete information needed for the manipulation primitives and events. The translation process proceeds as shown in Fig. 7.3. First, each of the abstract nodes is mapped independently by the factory to a suitable embodiment specific manipulation primitive. Then, the abstract events (transitions) are processed in a similar fashion.

The embodiment specific factory, uses abstract primitive parameters and environment information to choose a suitable embodiment specific controller and its parameters. Typically, each type of abstract primitive is mapped to a certain corresponding embodiment specific primitive, although it is possible that this relation is not one-to-one or even static. For example, it is possible to map different abstract arm movement primitives to a single embodiment specific primitive if that primitive can be parametrized in a suitable fashion, as we show in Table. 7.3. In addition to choosing the type of the controller, the factory can deliver embodiment specific parameters to the controller. These can be used, for example, to communicate a collision free path for that particular embodiment. Thus, in this case, the factory will also act as an embodiment specific path planner. A similar process is in place for the transition events, that is, the factory produces computation nodes for sensor processing which use the available sensors of each embodiment to detect the events.

For free motions in a collision free space and guarded motions, common primitive controllers can be used over several embodiments. This is possible by having common control and sensor interfaces for the arm, which in our case perform either Cartesian or joint space velocity control. Thus, we can use manipulation primitives that use the arm velocity control for all hardware platforms without modifications just by setting appropriate parameters through the embodiment specific factory. The same applies to

Abstract	*Tombatossals*	*Extra parameters*	*Control and sensors used*
Grasp	Robust grasp	Pregrasp size	Arm control, FT and tactile sensors
Move	Transport	-	Arm control
Push	Transport	-	Arm control
Transport	Transport	Obstacles, constraints	Arm control
Place	Place	Contact threshold	Arm control, Force-torque sensor
Slide	Slide	Slide force threshold	Arm control, Force-torque sensor
Release	Release	Hand position	Arm control

Table 7.3: Mapping of abstract primitives to Tombatossals implementation. Extra parameters show the parameters that are not in the abstract definition given in Table 7.1 but can be specified for the embodiment specific primitive.

the transition conditions, for example a timeout transition condition can be used across all platforms as the condition relies on measurement of time which should be available in every platform.

The rules that are observed in the translation process are simple:

- Each node in the ATD must correspond to one node in the ESTD.
- Each edge in the ATD must correspond to one edge in the ESTD.
- Each edge label in the ATD can be represented by one or more edge labels in the ESTD.

These rules ensure that the execution of the ESTD can be traced back to the original abstract task description. This allows the system to report back failures to higher level so that the higher level system operating on the abstract task description is able to reason using the same concepts. The possibility to represent an abstract transition condition by more than one embodiment specific ones allows, for example, to check the success of multiple manipulation primitives, such as separate arm and hand controllers, with a single success transition condition in the ATD.

While the translator component is universal across all embodiments, the factory component needs to be built specifically for each platform. The complexity of the factory affects the flexibility of the final system. A simple factory with fixed mappings between abstract and embodiment specific primitives and events is sufficient for many relatively simple tasks. Complex factories considering for example path planning for redundant manipulators or the choice of a grasping primitive among several are possible and discussed more in Sec. 7.4. If the factory is unable to find a suitable mapping for any reason, the mapping fails which is reported back to the task level. However, it should be noted that the factory is often fairly simple to implement because there are only a limited number of abstract primitives, event types and parameters, and the factory needs to consider only one primitive or event of an abstract task at a time.

Abstract	*Melfa*	*Extra parameters*	*Control and sensors*
Grasp	Grasp	grasp force	Arm control, tactile sensors
Move	Move	-	Arm control
Push	Move	-	Arm control
Transport	Transport	Motion constraints	Arm control, tactile
Place	Transport	Motion constraints	Arm control, tactile
Slide	-	-	-
Release	Release	Hand position	Arm control

Table 7.4: Primitives for the Melfa platform. Extra parameters show the parameters that are not in the abstract definition given in Table 7.1 but can be specified for the embodiment specific primitive.

7.3 Experimental validation

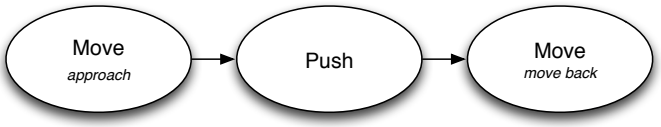
We demonstrate the mapping of the abstract state machine by showing a pick and place task that needs first clearing the path to the object to be grasped. This is achieved by developing two simple abstract task descriptions, the first to push an object away (see Fig. 7.4(a)) to clear the path to perform the second action: a simple pick and place (see Fig. 7.4(b)).

To enable mapping of the ATD, we implemented the translation component described in Section 7.2.2 for two different platforms, *Tombatossals* and a 6-DOF Melfa RV-3SB arm with a 1-DOF WRT-102 gripper from Weiss Robotics. The WRT-102 gripper is based on the PG-70 parallel jaw griper from Schunk. The implementation included the required platform specific controllers for the different nodes in the ATD and the platform specific transitions, as well as the required configuration information.

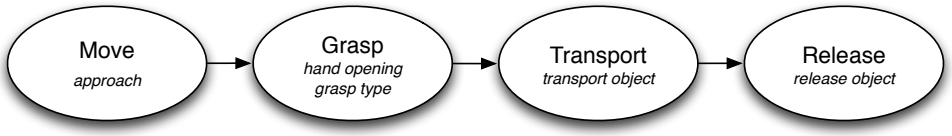
While the translation and the different requirements for the primitives are shown for Tombatossals in Table 7.3, the same information is available for the Melfa platform in Table 7.4. The Melfa platform does not utilize as much sensor feedback in the primitives due to the difference in hardware. The primitives for the Melfa platform are, in general, different from the primitives presented in Chapter 3 for Tombatossals as the SDH hand integrated into Tombatossals is much more capable in terms of DOF for example. The effects of the use of different embodiments can be seen, for example, in the *grasp* primitive. The Melfa robot is not able to do any of the corrections that *Tombatossals* does, it is only possible to perform the force adaptation using the tactile sensors.

For the implementation of the manipulation primitives for the Melfa platform, we used simple strategies. The basic guidelines used for the implementation are provided in the following list:

- Grasp: Closes the gripper until the desired *grasp force* is detected by the tactile sensors.



(a) Push object abstract task definition.



(b) Pick and place abstract task definition.

Figure 7.4: Abstract task definitions tested on Tombatossals and Melfa platforms.

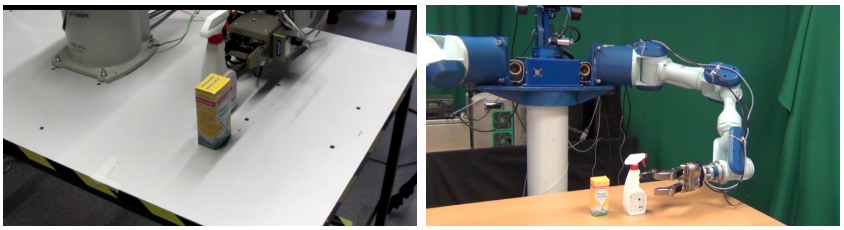
- Move: This is a very similar implementation of the transport primitive described in Chapter 3 for Tombatossals.
- Transport: Behaves exactly as the move primitive but keeps applying force with the gripper. Otherwise the transported item would be loosened by the gripper.
- Release: Open the gripper until the fingers reach the *hand position* parameter.

As a result, shown in Fig. 7.5, we were able to push away one object and grasp the second one based only on the sensor data from the hand and the arm, when given estimates of the pose of the objects. Using the same abstract task definitions for both platforms shows clearly that we are able to use abstraction and then turn this abstract information to platform specific primitives and transitions used in the sensor-based control.

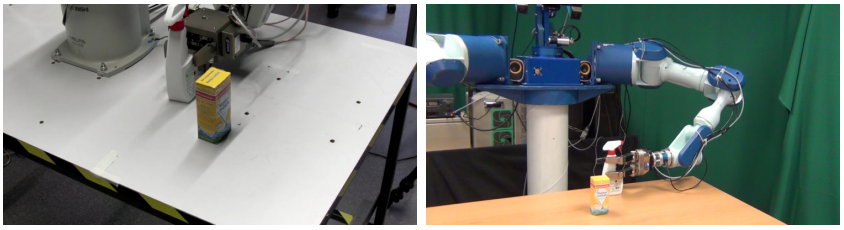
In the context of the demonstration we used the same Cartesian controllers for both arms. On the other hand, the hands are too different in terms of kinematics and sensors so that each hand had its own implementation of control. Also the transitions for grasp stability or instability were customized for each of the platforms in order to effectively use the different sensor capabilities available on the platforms. It should be noted that the task was nevertheless described using only the abstract description, without any embodiment specific information.

7.4 Discussion

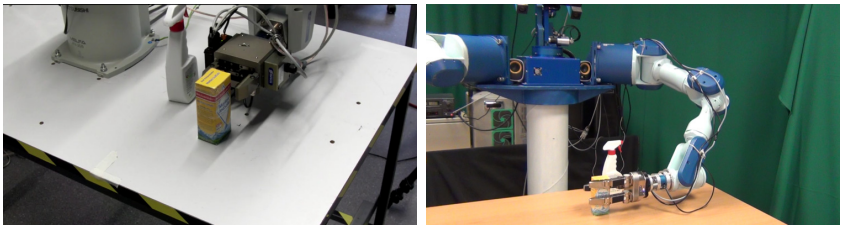
In the approach presented in this chapter, some of the manipulation primitives implemented in Chapter 3 have been implemented for the Melfa platform. Though all of them are intended to have the same behaviour and effects, their implementation can



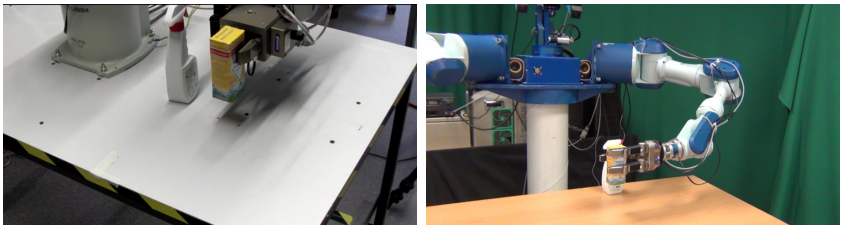
(a) Approach



(b) Push



(c) Grasp



(d) Lift

(e) Release

Figure 7.5: Action execution on different platforms. Left column: Melfa RV-3SB with PG70 gripper. Right column: *Tombatossals*

vary significantly due to the mechanic, kinematic, and perceptual differences between embodiments. Especially in the case of the grasping primitives, the differences in hand construction (number of fingers, number of joints, number of actuators) and available sensors are on a level which makes automatic construction of primitives a grand challenge. In order to better exploit the advantages of each embodiment, it would be possible to use a more detailed abstract description of the grasping primitive, for example, differentiating between grasp types such as enveloping grasps, power grasps, precision grasps, or hook grasps. Nevertheless, the state-of-the art in grasping does not currently allow this level of abstract information to be used automatically and therefore each of the grasp flavours would need to be adapted manually, depending on the hand characteristics. This is the case especially if the full reactive capabilities of the embodiment are to be used.

On the other hand, primitives related primarily to control of arm motions can be general so that the same manipulation primitive implementation can be used on multiple embodiments, as we demonstrated experimentally. However, in order to enable the full capabilities of an embodiment to be used, the path planning of the arm motions needs to consider the particular embodiment. This means that the factory component of the translation process is embodiment dependent, at least to some extent. Nevertheless, the planning of collision free paths between end-effector poses can be performed using openly available software libraries and therefore the implementation of the factory is possible with reasonable effort.

The position of the factory component is central in the approach. Differing capabilities of different embodiments, for example the size of the workspace, have the effect that there is no way to guarantee that an abstract plan would be translatable to any embodiment. Without requiring certain capabilities, it cannot be known with a certainty that a specific abstract plan can be executed on a specific embodiment. In the longer term, general principles on how embodiments could automatically instantiate sensor-based primitives would offer great benefits. However, a complete solution would need to 1) analyse and abstract a skill performed by an existing system and 2) be able to map the abstract skill to the present embodiment.

7.5 Conclusion

This chapter presented an abstraction framework allowing multiple embodiments. The main contribution is the abstraction framework and translation mechanism. We showed experimentally that the transfer of action plans is possible between different system setups while retaining the specific reactive capabilities of each embodiment.

These results complement the results from the RoboEarth project, where similar results have been shown for the higher level planning without the viewpoint of reactive primitives presented in this thesis. The results encourage us to believe that manipulation

problems can be solved in complex, unstructured scenarios while retaining hardware independence on a higher level. However, immediate feedback capabilities seem essential in coping with the complexity of the world.

The embodiment specific manipulation primitives currently require careful design for each embodiment. Procedures which could automatically at least bootstrap the building of the controllers, or even construct the controllers, would be very valuable. It seems that the use of machine learning techniques would be an interesting and possibly profitable avenue of research in this direction. This approach would most likely require high quality simulations of the embodiment in order to provide training data for the learning approaches, a possible application of the simulation engines presented in Chapter 5.

The abstraction mechanisms presented, implemented and validated in this chapter were published in [Laaksonen et al., 2010] and extended in [Felip et al., 2013].