ICT Call 7
ROBOHOW.COG
FP7-ICT-288533

Deliverable D6.1:

Description of the design and realization of the first version of the RoboHow.Cog plan language

July 31st, 2013

Project acronym:	ROBOHOW.COG
Project full title:	Web-enabled and Experience-based Cognitive Robots that Learn Complex Everyday Manipulation Tasks
Work Package:	WP 6
Document number:	D6.1
Document title:	Description of the design and realization of the first version of the RoboHow.Cog plan language
Version:	1.0
Delivery date:	July 31st, 2013
Nature:	Report
Dissemination level:	Restricted (RE)
Authors:	Jan Winkler (UNIHB) Georg Bartels (UNIHB) Michael Beetz (UNIHB)

The research leading to these results has received funding from the European Union Seventh Framework Programme FP7/2007-2013 under grant agreement n^{O}288533 ROBOHOW.COG.

Summary

Work package 6 is to design, implement, and empirically analyze a new generation of plan languages and computational models for plan-based robot control. The plan-based controllers will be based on CRAM (Cognitive Robot Abstract Machine) [Beetz et al., 2010]. Generic requirements on such plan languages are investigated and their applicability in practical scenarios are tested. The present deliverable describes the current state of the plan language design and realization.
Work package 6 develops a plan language that specifically serves the needs of RoboHow. This means that the behavior specification mechanisms of the plan language are to be able to use the visual feedback and failure detection provided by the robot action tracking system, that it can context-dependently select and reason about suitable motion specifications based on geometric constraints and objective functions, can execute and monitor routines generated through imitation learning, and specify competent perception-guided manipulation activities on a symbolic layer. These behavior specifications will be represented explicitly, transparently, and modularily, such that the robot can semantically reason about these aspects of robot behavior and modify these specifications based on predicted action flaws.
In the first year we have carefully studied two problem instances for which we designed and implemented robot plans:

- First, generalized "fetch and place" plans for which the robot is required to fetch various kinds of objects from various places, including table tops, drawers, cupboards, fridges, etc and placing the objects at locations that are qualitatively described. Here the challenge is that the tasks can be vaguely described and therefore the robot has to decide as part of the plan where to stand to find pick up an object, how to grasp it, with one or two hands, where to grasp it, how to lift it, etc. The reason for this task domain is that it is the most widely used manipulation capability of robots.

- Second, symbolic constraint-based movement plans for pancake flipping, which are chosen as a very challenging manipulation task and requiring tight interaction with the control engineering tasks in work package 3.

RoboHow's achievements in plan and plan language design are mainly along the following four dimensions, which are documented by three individual peer reviewed publications and one publication currently submitted for review.

Parameterizing Plans with Descriptions of Objects, Actions, and Locations Robot plan languages employed by most cognitive architectures are too restricted when it comes to flexibility and generality in performing complex manipulation tasks in real physical environments. The reason of an action must be clear to a robotic agent in order to enable it to competently handle plan

failures and unexpected events. Also, with background knowledge about the causal properties of a high-level plan, a robotic agent can take plan-improving measures and can project as well as judge the outcome of an action. Plans that are structured in such a semantically annotated way can not only be executed, but also reasoned about. While these mechanisms can ensure a flexible behaviour while executing a plan, they need input from a perception system to maintain congruence with the state of the real world. These aspects are explained in terms of a generalized pick and place scenario in [Winkler et al., 2012] as attached to this deliverable.

Extending Manipulation Plans for Perception Guided Execution Reasoning about uncertain environmental setups gives a robotic agent a multitude of new approaches towards a generalized pick and place task. Tightly integrating these reasoning capabilities into a plan language is an essential step forward in handling situations in which objects to act on are not necessarily visible. To reference such situations, language elements are introduced that allow for partial and ambiguous object descriptions without referring to specific object instances and refining and grounding these ambiguous descriptions as the information is stepwise acquired during plan execution, in particular when perceptually detecting and examining in real physical scenes. These concepts are described in a technical report on *"Plan-based Control for Generalized Fetch and Place Tasks"* as attached to this deliverable.

Movement-Aware Control of Manipulation Actions We introduce the described techniques in terms of robust, flexible, and reliable robot plan language features that are part of the CRAM plan language. A tight coupling between perception and plan execution and the benefits from this coupling are shown. The research done in WP6 is highly relevant to WP5, as generated data from autonomous plan execution is recorded in a way machine learning methods can benefit from.

Perception Guided Execution of Movment-Plans Based on Geometric Tool-Features Furthermore, when performing complex manipulation tasks such as pouring liquids in containers, flipping pancakes or wiping surfaces success highly depends on *how* the corresponding motions are executed. Robot plan languages need to provide rich representations for such aspects of movements in order to enable learning from past executions or chosing parameters for future tasks. We outline how we augmented the RoboHow plan language with movement-aware action specifications which are directly grounded in constraint-based controllers developed by WP3. The constraint-based nature of the movement parameterization also exhibits promising generalization properties and seamless integration into symbolic reasoning. The presented concepts are described in detail in [Kresse and Beetz, 2012] and [Bartels et al., 2013], as also attached to this deliverable.

Contributed Papers

Papers included in this deliverable are:

- Bartels, G., Kresse, I., and Beetz, M. (2013). Constraint-based movement representation grounded in geometric features. Submitted for review to Humanoids 2013.
- Kresse, I. and Beetz, M. (2012). Movement-aware action control – integrating symbolic and control-theoretic action execution. In *IEEE International Conference on Robotics and Automation (ICRA)*, St. Paul, MN, USA.
- Winkler, J., Balint-Benczedi, F., Beetz, M. (2013). Plan-Based Control for Generalized Fetch and Place Tasks. Technical Report.
- Winkler, J., Bartels, G., Mösenlechner, L., and Beetz, M. (2012). Knowledge enabled high-level task abstraction and execution. *First Annual Conference for Advances in Cognitive Systems*, 2(1):131–148.

Advances in Cognitive Systems 1 (2012) 1-6 Submitted 9/2012; published 12/2012

Knowledge Enabled High-Level Task Abstraction and Execution

Jan Winkler WINKLER@CS.UNI-BREMEN.DE
Georg Bartels GEORG.BARTELS@CS.UNI-BREMEN.DE
Institute for Artificial Intelligence, Universität Bremen, 28359 Bremen, Germany

Lorenz Mösenlechner MOESENLE@IN.TUM.DE
Intelligent Autonomous Systems, Technische Universität München, 80333 Munich, Germany

Michael Beetz BEETZ@CS.UNI-BREMEN.DE
Institute for Artificial Intelligence, Universität Bremen, 28359 Bremen, Germany

Abstract

This paper investigates issues in the plan design of cognition-enabled robotic agents performing everyday manipulation tasks. We believe that plan languages employed by most cognitive architectures are syntactically too restricted to specify the flexibility, generality, and robustness needed to perform physical manipulation tasks. As a consequence, the robotic agents often have to employ flexible plan execution systems. This causes two problems. First, robots cannot understand how their plans generate the flexible behavior and second, they cannot use the mechanisms of flexible plan execution for plan improvement. We report on our research on plan design for robotic agents performing human-scale everyday manipulation tasks, such as making pancakes and popcorn. We will stress three key factors in plan design. First, the use of vague descriptions of objects, locations, and actions that the robot can reason about and revise at execution time. Second, the plan language constructs needed for failure detection, propagation, and handling. Third, plan language constructs that represent and annotate the intentions of the robot explicitly even in concurrent percept driven plans. We clarify our concepts in terms of a generalized pick and place task. In this context, failure handling in uncertain environments is still an open topic for which we demonstrate a solution using our high level plan representation. We illustrate the advantages of our approach in a simulated environment.

1. Introduction

As autonomous robotics starts to tackle human-scale manipulation tasks in human living and working environments the need for cognitive mechanisms becomes more and more pressing. When we ask a robot to clean up it has to put all relevant objects in the right places: dirty cups into the dishwasher, clean cups back into the cupboard, the butter into the fridge, the cold coffee into the sink, and the unfinished bread into the trash can. When we ask the robot to set the table it has to arrange the needed objects appropriately based on who it believes to take part in the breakfast and what they will have. Even for a task as simple as *fetch me a glass of water* the robot has to decide where to stand, which grasp type to apply, where to place the fingers, and how to lift the glass.

The difference between a vague task description such as clean up, set the table, and fetch me a glass of water, and what we expect the robot to do in order to accomplish the task must be provided by the robot's learning, decision making, and planning capabilities. A system or robotic agent that employs such cognitive mechanisms to improve its task performance –in particular with regards to robustness, flexibility, adaptivity, and efficiency– is what we call *cognition-enabled* (Beetz et al., 2012).

The result of these learning, decision making, and planning capabilities should produce robot behavior that is as natural, efficient, robust, and flexible as if we gave the same command to a human operator to remotely control the robot with a game console. To get proper intuitions about how it should look when a robot accomplishes tasks such as clean up it is very illustrative to watch the videos accompanying the design paper for the PR1 robot, in which a PR1 robot is manually controlled to perform human-scale manipulation tasks such as cleaning up the living room (Stanford University, 2008). Compared to the behavior generated by today's advanced autonomous manipulation platforms it is evident that the manually controlled robot moves smoothly and continuously such that the start and end of the individual actions are not detectable and that it immediately detects execution failures and recovers from them gracefully.

We introduce the high level plan environment CRAM (Cognitive Robot Abstract Machine) and its abstracting structure in terms of general high level plans and robot specific subtasks (Beetz, Mösenlechner, & Tenorth, 2010). In this environment, a classical pick and place problem is modeled. Abstraction features like goals, subplans and process modules allow for flexible plan execution and reasoning. Therefore, an approach for analysis of executed plans by extracting annotated execution traces from plan execution components is explained. An overall evaluation of the system is described and related work as well as an outlook on future research topics in this field are given.

2. Methodology

We believe that a high performance level will be difficult to reach without the robot's plans being properly designed to specify how the robot is to respond to sensory data in order to accomplish its goals. That is, the plans have to specify concurrent percept guided behavior. Most cognitive systems and architectures assume that plans are partially ordered sets or even sequences of actions. There are at least two fundamental problems if we assume that the robot behavior above is generated by partially ordered action plans. First, the flexible and robust behavior cannot be understood in terms of a partially ordered action plan and therefore a robot could not diagnose what in its plan caused a certain flawed behavior. Second, if the purpose of the cognitive mechanisms is to improve the performance of the robot plans then this is best done by making the plans more sensible and tolerant to plan failures that the robot expects. However, in partially ordered action plans there is neither space for failures nor are there control structures that can be used for failure detection and recovery.

In this paper we describe plans which we have carefully designed such that they can achieve high performance behavior for robot manipulation activities in realistic environments. In this context we will propose three concepts that we found to be key factors for the flexibility and robustness of the robot plans.

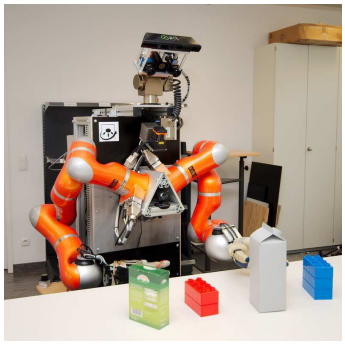


Figure 1. Different objects mostly require different grasping techniques and may not obstruct each other when placing them.

2.1 Designators as Entity Descriptions

Designators contain vague symbolic descriptions of entities such as objects, locations or actions. These can describe an object to be picked up by the robot, but also the hand movement that pushes a spatula under a pancake without damaging it (Beetz et al., 2011) or the grasp type to use when picking an object from a table.

Such entity descriptions can be vague in the beginning and can be refined as the robot gains more information. As once aquired information about an object could be wrong, the designator's description can be revoked and replaced by the newer, correct information. A reasoning component decides whether a designator can be translated into effective instructions for the robot, like fetching a cup from the table, making designators a versatile plan parametrization component. Conceptual handling mechanisms for this vague information enable the robot to reason about what to do if a description turns out to be ambiguous, i.e. information for certain decisions is missing. For example, if a robot is to put a parcel containing a bomb into a toilet in order to defuse it (Moore, 1985) and it faces two or more identical parcels, it should put all of them into the toilet. In this situation, the plan that targets *one* parcel must be performed on *each* of the detected parcels before the execution is complete. On the contrary, if the robot is to bring a clean cup from the kitchen, then most likely any clean cup will do. This plan targets only *one* cup and terminates after it was performed *once*. Whether the given plan is executed once or multiple times strongly depends on the infered situational context and information.

2.2 Failures, Detection and Recovery

Since plan execution in the real world is a complex task and holds many minor details that are not modeled in simulated environments, dynamic plan components have to remodel plans on a constant basis and failure handling code needs to be part of the plan framework. Failure handling can be done on a comparatively low execution level in order to keep the high level plans short and simple. Since this approach makes failure-triggered dynamic replanning difficult, we abstract the failure management and integrate it into the respective high level elements. This way, we can reason about the occured failures with the aid of high level knowledge.

Problems that arise during execution are propagated through the tree of subplans and goals until a suitable failure handling mechanism is triggered. As pick and place tasks are a common element in most high level plans, robust plan execution and failure handling are explained in context of such a scenario.

Dynamic replanning based on failure diagnosis was described in (Beetz, 2000). The reasoning mechanisms used in the current system do not reflect such an abstract level of failure handling but specialize on a number of obvious challenges anticipated during manipulation, perception and navigation tasks. As this number of potential exceptions rises, a more comprehensive handling and repair algorithm for plans is intended.

2.3 Plans that Specify Flexible Robot Behavior

Formulating plans as robot control programs that are transparent for a reasoning system allows for inferring what the robot did, when it did it and, most importantly, why it did it. In CRAM, this is achieved by annotating plans using Prolog expressions. For instance, by calling a plan *(achieve (object-in-hand ?object))*, we state that if the plan succeeds, the robot believes that the object described by the object designator bound to *?object* is in the gripper. In addition to *achieve*, the current system supports *perceive-object* to state that an object has to be perceived, *perceive-state* to check if a specific logical expression describing a state in the world holds, *perform* to state that an actual action has to be performed and *at-location* to state that a specific code block has to be executed at a specific location. These mechanisms cover the basic components for the herein presented pick and place example. Reasoning about the current belief state can be done during run time and therefore enables for context dependent failure solutions. Since the herein presented plans are dynamic, flexible structures, their properties can be changed during run time without completely restarting all plans.

In classical AI approaches, environmental preconditions are assumed well defined in a *closed world scenario* during planning and plan execution. This assumption holds for most laboratory conditions, but ceases to work out quickly in the real world. Once grasped objects might slip from the gripper, very similar objects might be placed next to each other which causes perceptual routines to fail and even an arbitrary number of external agents might change the environmental conditions unnoticed in between. Without proper failure handling and plan correction, these situations will result in wrong actions or even an overall failing of plan execution. Either way, the outcome of the plan execution is uncertain. CRAM serves the need for a flexible plan representation format that is either extensible during compile time through the use of a plan library or during execution time if necessary.

3. CRAM Plan Framework to Make Robust High-Level Plans More General

3.1 Plan Design

In classical robot control architectures, plans consist of partially ordered atomic actions in a purely symbolic, mostly relatively abstract domain. This is necessary to keep the state space small and planning feasible while sacrificing the ability to represent the state of the world accurately enough.

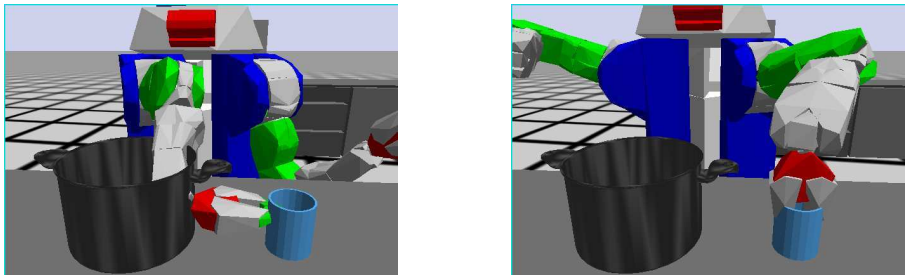


Figure 2. When grasping a cup that is standing left of a pot, it is better to use the left arm.

It turns out that in pick and place plans it is more important to take the current state of the world at a geometric level into account than to find a sequence of actions for placing objects. In other words, the sequence of actions for picking and placing objects does not vary while parameters such as *where to stand*, *which arm to use* etc. strongly influence the success or failure of a plan. For instance, if the robot's task is to grasp a cup and it needs to choose which arm to use, the result mainly depends on the geometric configuration of the environment. Figure 2 shows that if the cup is standing close to a pot, it is better to use the left arm because the grasping trajectory will be much easier to calculate and to execute. These decisions should not be made before they are needed in order to take into account the complete belief state, which means that they need to be made at execution time.

In CRAM, plan parameters such as arm trajectories, poses for objects and for the robot, objects themselves, and other actions such as navigation are represented as designators. Designators are symbolic descriptions. These descriptions are lists of key-value-pairs, each representing a property of the designator. For instance, to describe any location on the table for the plate, we write:

(a location (on table) (for plate))

Please note that this is just a compact representation of a conjunction of constraints. Designators are resolved using a Prolog reasoning system and the result of the designator resolution is based on the current belief state and content of knowledge bases such as KnowRob (Tenorth & Beetz, 2012). With the help of designators plan parameters are turned from numeric values that are hard to reason about into transparent logical representations of these parameters. One important aspect of this logical representation is that it becomes possible to write very general high level plans that allow for executing the same plan on different robots or in simulation. This is achieved by using designators as commands for lower level components, so called process modules.

For high-level plans, process modules appear as black boxes with a well defined interface. High-level plans interact with the world through process modules. The number of process modules depends on the robot and the context in which actions are executed. For manipulation in human households, we define four process modules: manipulation, navigation, perception and moving the head with its cameras mounted on top. Process modules are initialized when the robot starts operating, for instance, when a top level program is started. Commands are sent as designators to the process modules. When a process module receives a designator, it translates the symbolic representation of the designator to actual action parameters.

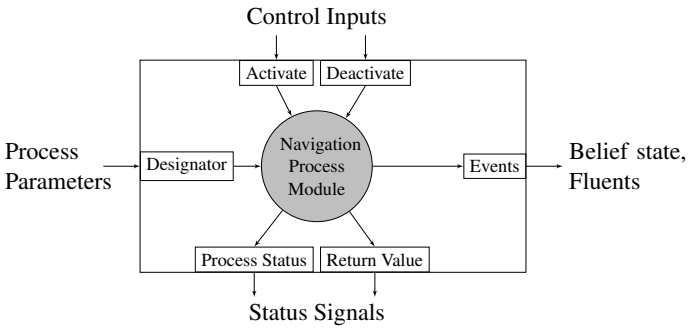


Figure 3. Process module encapsulating a navigation control process. The input is a location designator, symbolically describing the goal pose and the context this code is executed in (e.g. *(location (to see) (obj cup))*). The feedback events provide information about the status of the current navigation task. The process module can be activated and deactivated and provides success and failure signals.

For instance, it calculates a trajectory for manipulation. After executing the action, the process module returns a result value or, if an error occurs, throws that error to allow the high level plan to handle it. Figure 3 shows the navigation process module as an example.

Besides versatile mechanisms for parametrizing plans at run time, powerful failure handling mechanisms are important to achieve robust behavior. For instance, if grasping fails because an object could not be reached, a possible way to handle that error is to navigate to a different location. In classical architectures replanning would be necessary since navigation to a different location is a separate action. In CRAM, a new solution for the plan parameter *location to stand for grasping* is generated from the updated current belief state and the grasping action is retried.

The implementation of the pick up plan uses the special CRAM macro *with-failure-handling* that allows to execute code whenever a failure is thrown. In addition to normal exception handling (i.e. either rethrowing an exception or handling) known from languages such as C++, Java or Python, *with-failure-handling* allows to execute a retry.

An execution environment for robots that exhibits cognitive behavior needs to provide means for self awareness. The robot needs to understand what it did as well as when and why it did it. This is not only a valuable tool for debugging but it also allows for error handling that relies on a deeper understanding of the task the robot performed. For instance, let us consider a relatively complex high level plan for setting the table for breakfast. It consists of the following (unordered) plan steps:

- Put the plate on the table.
- Put the knife on the table.
- Put the napkin on the table.
- Put the cup on the table.
- Put the bread basket on the table.

In situations where objects need to be placed closely to each other, objects might accidentally be pushed around or objects need to be moved out of the way before an action becomes possible. For instance, when putting down the bread basket, the cup might be in the way and the robot might decide to move it to a temporary location.

Table 1. A simple plan that moves an object to a specific location, i.e. achieves the goal *(loc ?object ?location)*. This is done by calling two other plans to achieve their respective goals, namely *(object-in-hand ?object)* and *(object-placed-at ?object ?location)*.

```
(def-goal (achieve (loc ?object ?location))
  (unless (perceive-state '(loc ,?object ,?location))
    (achieve '(object-in-hand ,?object))
    (achieve '(object-placed-at ,?object ,?location))
    (perceive-state '(loc ,?object ,?location))))
```

While this does not influence the success or failure of a single plan step, it definitely changes the overall outcome of the top level plan because one goal that has been achieved initially was undone later. Detecting this by just using a mechanism that is based on exception handling is not possible and a deeper understanding of the plan is necessary.

CRAM plans can be reasoned about because they are not just high level controllers but contain semantic annotations that indicate the purpose of specific code parts. For instance, the plan for moving an object to a specific location is defined as in Table 1.
Instead of using simple function names for the plan and its subplans, we use expressions that are grounded in an underlying knowledge base. The expression *(achieve <occasion>)* indicates that if the corresponding plan terminates successfully, the logical expression <*occasion*> needs to hold in the robot's belief state. This implies that it is the responsibility of a developer of a plan that achieves an <*occasion*> to carefully design this plan to ensure –independently of the starting conditions– that after successful execution of the plan <*occasion*> really holds.

In addition to *achieve*, CRAM currently provides *perceive-object* to indicate that an object described by an object designator has to be found, *perceive-state* to check if a specific occasion already holds, *perform* to indicate that an actual action described by an action designator is to be executed and *at-location* to indicate that a code block has to be executed at a specific location described by a location designator.

Since CRAM plans call subplans, they form a hierarchy of goals as shown in Figure 4. This hierarchy expresses the causal relationship between plans and subplans. The fact that a plan is a subplan of its parent indicates that it directly helps to achieve the parent's occasion.

Obviously, building up a library of robust plan that achieve basic goals such as pick and place is very time-demanding. We expect, however that high usage of these sub-plans in more sophisticated household chores will be well worth the effort spent.

3.2 Plan Execution Analysis through Execution Traces

Executing a plan creates a plan tree as described in the previous part. This tree contains information about when a plan was executed, why it was executed, what the values of its parameters were, if it threw an error and what the result value was. In addition, process modules create events that are stored on a time line and allow for reasoning about the actual low level actions that happened. For instance, if the robot changed its location, the event *(location-changed robot)* is generated and stored on the time line. The resulting execution traces include the plan tree, plan parameters, failures and the result value, but also lower level information, for instance, arm trajectories.

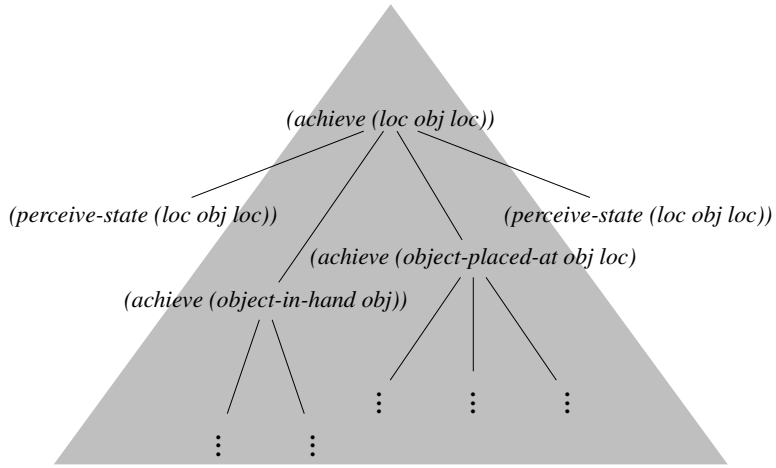


Figure 4. Example of a plan tree that is generated by *(achieve (loc obj loc))*. It shows the relationship between the subplans. For instance, the reason for executing the statement *(achieve (object-in-hand obj)* was to place the object at location *loc*.

Table 2. Sample Prolog query to gather unachieved goals from an execution trace after a plan execution.

$$\begin{aligned}\text{UnachievedGoalError}(?o) \Leftarrow\ & \text{TopLevel}(?tt) \wedge \text{Task}(?tsk) \\ & \wedge \text{Subtask}(?tt, ?tsk) \wedge \text{TaskGoal}(?tsk, ?o) \\ & \wedge \text{TaskEnd}(?tt, ?t) \wedge \neg\text{Holds}(?o, ?t)\end{aligned}$$

In addition to the plan tree, the execution trace contains the time line with the events that occurred during action execution in the process modules and fluent values. It provides enough information to restore the complete execution context at any point in time.

After an execution trace has been recorded, it can be serialized and stored in a file on a hard drive for later use. A number of Prolog predicates provide an expressive interface to query the execution trace and allow to formulate complex conditions such as unfulfilled plan goals. The corresponding query can be stated as show in Table 2.

A more detailed description of execution traces can be found in (Mösenlechner, Demmel, & Beetz, 2010) and its use in combination with simulation based temporal projection to optimize plans by applying transformation rules is described in (Mösenlechner & Beetz, 2009).

4. Pick and Place Scenario

The features offered by CRAM plans are utilized to model a simple evaluation scenario of robust and autonomous pick and place tasks. In our setup a robot agent is placed in a world that contains two tables with several household objects on top of them. The task of the robot is to pick one of the objects and to subsequently place it at a specified position. While this task seems to be rather easy, mastering it requires extensive reasoning and error correction capabilities on the part of our cognitive agent.

Table 3. Sample goal definition that describes the process of getting an object into the robot's *hand*. Entry points for failure handling are declared for different error cases within *with-failure-handling*.

```
(def-goal (achieve (object-in-hand ?obj-desig))
  (with-designators
      ((pickup-action-desig (action `((type trajectory) (to grasp)
                                      (obj ,?obj-desig))))
       (lifting-action-desig (action `((type trajectory) (to lift)
                                       (obj ,?obj-desig))))
       (pickup-location (location `((to execute) (action ,pickup-action-desig)
                                    (action ,lifting-action-desig)))))
    (unless (perceive-state `(object-in-hand ?obj-desig))
      (with-failure-handling
          ((object-lost (f)
             (retry))
           (navigation-failure (f) ...)
           (object-not-found (f) ...)
           (manipulation-pose-unreachable (f) ...))
        (perceive-object 'a ?obj-desig)
        (at-location (pickup-location)
          (reduce `(achieve (grasped ,?obj-desig))
                  `(perform ,pickup-action-desig))
          (perform lifting-action-desig))
        (unless (perceive-state `(object-in-hand ,?obj-desig))
          (fail 'manipulation-failed))))))
```

Table 4. Code example of a top level function that picks an object and places it at a given location.

```
(def-top-level-cram-function pick-and-place-scenario (obj-desig loc-desig)
  (with-process-modules
    (achieve `(loc ,obj-desig ,loc-desig))))
```

4.1 Scenario Setup in a Simulated Environment

Table 4 shows the working top level function *pick-and-place-scenario* that enables the robotic agent to robustly and autonomously perform this task with each of the objects in the world and for each of the available supporting surfaces. The two plan parameters are an object designator denoting the object to grasp and a location designator that symbolically represents the desired put down location. After declaring the appropriate process modules to be started for the current platform using *with-process-modules*, which is shown in Table 5, it calls to achieve the goal *(loc ?object ?location)* which in turn executes two actions in sequential order (see again Table 1). First, the robot should get the specified object into its gripper by trying to achieve the goal *object-in-hand*. Once this has been done successfully, the put down goal *object-placed-at* is to be achieved. An example task parametrization of taking a grey mug from the table and putting it somewhere on the counter is shown in Table 6.

As one can see, the given top level function is designed to be as general as possible. In fact, it is missing various key information, making it very ambiguous.

Table 5. The macro *with-process-modules* defines the set of process modules active for the code part described by *body*. The macro *with-process-modules-running* called herein initializes the specified modules.

```
(defmacro with-process-modules (&body body)
  `(with-process-modules-running
       (pr2-manip-pm:pr2-manipulation-process-module
        pr2-navigation-process-module:pr2-navigation-process-module
        gazebo-perception-pm:gazebo-perception-process-module
        point-head-process-module:point-head-process-module)
     ,@body))
```

Table 6. Two designators that can be used to parameterize the top level plan to pick up a grey mug from the table and then put it on the counter, and the appropriate call of the top level function.

```
(with-designators ((object-designator (object '((type mug) (color grey)
                                                (location (on table)))))
                   (location-designator '(location ((on counter)))))
  (pick-and-place-scenario object-designator location-designator))
```

It is this ambiguity that ensures its generality. By underspecifying the behavior of the agent we leave room for results of reasoning processes that rely on context and robot-specific information that are only available during execution.

The specification of the designators for the example task further leverages this idea: The sample designators in Table 6 contain only symbolic information, e.g. *(location ((on table)))*, that does not relate to a specific homogenous transform or any other subsymbolic information, before execution starts.

For example, the high level plan that achieves the goal *object-in-hand* is shown in Table 3. As a first step, three designators of different kinds are initialized. These include actions for grasping and lifting the object as well as a location at which these actions should be executed. Note how designators can use other designators as values to build more complicated specifications, e.g. a base location that is suitable to execute both the pick up and lifting action. Furthermore, the same mechanism is used to incorporate the object designator from the plan parametrization into the grasping and lifting actions. As a result, inspecting the first grasp action designator after initialization will denote an action that grasps a grey mug that is located on the table – and nothing more.

After the designators have been initialized, a call to *perceive-state* triggers a perceptual routine to make sure that the desired goal of *object-in-hand* has not already been fulfilled. Subsequently, the macro *with-failure-handling* starts the failure handling mechanism. It is used to register, as an integral part of the plan, which possible error cases can occur and how to react to them. The error case of losing an already grasped object is depicted in more detail: In this situation, the object is just regrasped using the same *object-in-hand* goal.

Making possible error cases and the respective correcting actions part of the high level plans renders these plans more robust and less specific in terms of predefined environmental conditions. As a result, the overall plan library contains less high level plans. In fact, it contains only one high level plan for achieving *object-in-hand*, thus removing the need for a mechanism that chooses the right plan for achieving *object-in-hand* based on pre-condition perception.

Table 7. Goal definition for perceiving an object matching a given description *?obj-desig*. Three different locations are checked for the object's presence before an *object-not-found* error is escalated to the next higher abstraction level.

```
(def-goal (perceive-object a ?obj-desig)
  (with-designators
      ((obj-loc-desig (location `((of ,?obj-desig))))
       (view-loc-desig (location `((to see) (obj ,?obj-desig)
                                    (location ,obj-loc-desig))))
       (perceive-action-desig (action `((to perceive) (obj ,?obj-desig)))))
    (let ((obj-loc-retry-cnt 0))
      (with-failure-handling
          ((object-not-found (f)
             (when (< obj-loc-retry 3)
               (incf obj-loc-retry)
               (setf obj-loc-desig (next-solution obj-loc-desig))
               (retry)))
           (navigation-failure (f) ...))
        (at-location (view-loc-desig)
          (achieve `(looking-at ,?obj-loc-desig))
          (perform perceive-action-desig))))))
```

The first actual step in the high-level plan that achieves *object-in-hand* is a call to perceive one object of a certain kind, i.e., any object that matches the given symbolic description of the object that shall be grasped. *Perceive-object* itself is yet another very general high level plan.
The details of its implementation can be seen in Table 7. In case of successful perception the object designator now holds additional subsymbolic information, such as the position, orientation and size of the object, which have been added by the perception process module.

After the object has been perceived, the *at-location* macro calls the navigation process module to navigate to a base pose that is suitable to execute both manipulation actions and makes sure that the action is executed at this location. Within our plans *at-location* has special semantics that extend beyond a simple sequence of navigation and manipulation goals. First of all, all calls of *achieve*, *perform*, etc. inside the body of *at-location* are considered after the desired location has been reached. Additionally, the macro includes an extra monitoring task that ensures that the robot stays at the specified location while performing the body functions. For more details on the rational behind *at-location* please refer to (Beetz, 2002).

In order to perform these navigation tasks, the abstract location designator must be resolved into specific numeric values to be used as goals for the low level navigation controllers. It uses advanced reasoning mechanisms that consider the given symbolic specification inside *pickup-location* and both the symbolic and subsymbolic information provided by the object designator which itself is part of the location designator. For further details on this spatial reasoning which is used to resolve location designators please refer to (Mösenlechner & Beetz, 2011).

At this location both the pick up and lifting action are performed in sequence by passing the respective action designators to the manipulation process module. In order to add semantic information the pick up action and the goal *grasped* are wrapped inside the *reduce* macro.

What *reduce* does is to achieve a goal (first parameter) by calling a given function (second parameter), thus attaching goal intentions to arbitrary function calls. The *reduce* macro is particularly useful in two cases: When annotating semantic meaning that is not included in the plan library, i.e. a new goal, or when achieving a goal in a specific situation is possible with behavior that is different than the one specified in the corresponding plan from the plan library. Note that the implementation of *(def-goal <goal>)* involves an implicit *reduce*.

During post execution reasoning such semantic annotations allow the CRAM system to figure out why certain execution calls have been made. Being part of the execution trace afterwards, these annotations add valuable information for reasoning mechanisms and help while interpreting the situational context. Finally, another call to *perceive-state* tries to verify that the intended goal has been achieved, otherwise a manipulation failure is thrown which can be processed in the calling high level or top level plan. If it succeeds, the high level plan will terminate without throwing a failure, indicating successful execution.

To further elaborate on the workings of the failure handling system, assume for a moment that the grey mug is actually not at its expected location, i.e., on the table. As a result, the perceptual plan *perceive-object* (see Table 7) will throw an *object-not-found* failure. The corresponding failure handling routine within *perceive-object* will catch this failure and re-try three times to perceive the object at different resolution results for its symbolic specification *on table*, which will fail again.

Then, *perceive-object* will also fail and propagate the *object-not-found* failure to its calling plan, *object-in-hand*, which in turn catches it, too. This plan can react to the error differently because it has more context information. It could, for example, select a different symbolic location description for the mug, e.g. *on counter*, and retry itself using the changed plan parametrization.

A very probable example from the scenario at hand could be that some objects are not reachable at all, i.e. are out of the robot's gripper reach. Assume we gave the scenario an object description that does not match one singular object instance by specifying it's unique name, but rather the object type. If we now have multiple mugs, for example, and tell the robot to get any one mug, a failure handling strategy has to be present. The natural human approach would be to try to get the first instance we see and, in case we can't reach it, try the next one. This process goes on until we either succeed or fail for all mugs and give up. In this example, the decision to try a different mug happens on an even higher abstraction level that the choice of different base positions. An example implementation of this strategy is shown in Table 8.

Such a propagation cascade of generic failure handling is an elegant feature which can be used to react to the same execution error on various abstraction layers in different ways. As a result we can design high level plans which are more robust to execution errors and avoid replanning in a huge variety of situations. Additionally, since error handling is an explicit part of the plans, it is also semantically annotated and not hidden behind an impermeable layer of abstraction.

4.2 Analysis of Plan Execution based on Execution Traces

In the given scenario, the resulting execution trace after the plan was executed consists of the initial *pick-and-place-scenario* plan and the subsequent calls. These cover the *object-in-hand*, *object-placed-at*, and all contained subtasks. A part of the execution trace from the pick and place example is shown in Figure 5. Executed plan portions and their respective data is saved on a time line.

Table 8. All objects that match the description in *object-designator* are retrieved and *object-in-hand* is tried on each of them until one succeeds. The caught error *manipulation-pose-unreachable* that can be signalled by *object-in-hand* triggers trying the next object as long as objects remain.

```
(let ((perceived-objects (perceive-object 'all object-desig)))
  (when perceived-objects
    (with-failure-handling
        ((manipulation-pose-unreachable (f)
           (setf perceived-objects (rest perceived-objects))
           (when perceived-objects
             (retry))))
      (achieve `(object-in-hand ,(first perceived-objects)))))))
```

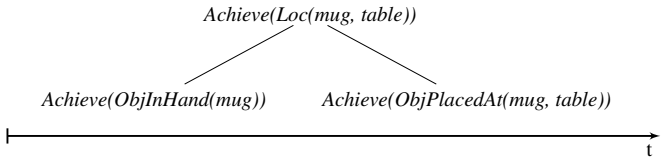


Figure 5. Excerpt from an execution trace in context of the pick and place scenario.

In addition to the time correlation, the task tree structure is also saved for each executed subplan or goal, allowing for complete reconstruction of the current belief state at every point in time during the execution. When trying to achieve the goal *Achieve(Loc(mug, table))*, the appropriate subgoals *Achieve(ObjInHand(mug))* and *Achieve(ObjPlacedAt(mug, table))* are tried and registered on the time line. The current belief state, object descriptions, the robot's pose, and arm trajectories are represented by fluents and are also saved on the time line.

Using this data, reasoning components can answer questions about why the robot drove around a table before it grasped a cup. The answer could be that the cup was not reachable from the former position and the new position looked more promising. In conjunction with a knowledge base with a given set of information as described in section 4.4, the robot could decide to look into a cupboard during runtime when looking for cups. A query to the execution trace could afterwards reveal that it didn't look into the drawers because usually there are no cups in it.

The gain from execution traces is the ability to reason about occurences in the actual experiment. Errors that came up during execution can lead to plan improvements this way by enquiring the reason for the failure. For example, we assume that grasping an object with multiple handles fails due to an unsuitable handle for the operating robot. The execution trace will include the call to a grasping function as well as the handle and gripper parametrization, i.e., which handle to grasp with which gripper. Although failed grasps can have a multitude of reasons, one specific solution to try is to choose a different handle and retry the grasp. This can be formulated as a simple and general rule for all objects with handles: *If grasping one handle fails and the object has more than one handle, try a different one until all handles have been tried.* Once this rule has been worked out, it can then be inserted back into the knowledge base as information for future plans.

4.3 Virtual Handles as a Tool to Resolve Grasping Actions

Action designator resolution involves decision making that depends on the objects involved, the current robot hardware and the desired and undesired effects of the action. In order to make this process more flexible and robust, we again propose to take a knowledge-enabled approach and make the influencing parameters explicit as designator properties. As a result this information is then used by our integrated prolog reasoning system whose decisions are also logged in the execution traces.

Regarding grasping actions the system needs to decide where to grasp an object, which arms to use, the pregrasp and grasp wrist orientations, and the respective hand configurations. We propose to use the designator property *virtual handle* as a hook to fill in grasping-related object information that are used during resolution of grasping action designators. Virtual handles denote parts of objects that can be grasped. As such the type of information that they hold is not restricted, i.e., a cylindrical shape primitive representing the entire handle of a mug, a single grasping point in the middle of the handle of a mug, a set of grasping points along the handle of a mug, or a set of grasping points distributed over the entire mug that yield a caging grasp are all valid virtual handles of objects.

Consequently, virtual handles can be used as hooks for very different established grasping approaches that, e.g. employ perception to detect grasping points (Saxena, Driemeyer, & Ng, 2008), define grasping points on primitive shapes (Miller et al., 2003), or deploy a knowledge base that has been created using a grasp planner using detailed CAD-models as input (Goldfeder et al., 2009). In our evaluation scenario we manually added virtual handles with relative grasping poses to the objects in our knowledge base prior to execution. During perception the perception process module automatically accesses this information in the knowledge base and adds it to the corresponding object designators.

With the help of virtual handles the reasoning system can infer the correct grasping parameters during action designator resolution. In our evaluation scenario we employ the following rules to decide which arm to use and which virtual handle to grasp:

- Only free arms are available for grasping – the internal belief state keeps track of which arms are currently employed.
- Arms have to reach a virtual handle without collisions – this information is provided by a routine that does constraint aware, IK-based reachability checking.
- The minimal number of arms has to be used for grasping – objects come with this property in our underlying knowledge base.
- From all the arm-virtual-handle-tuples fulfilling the above rules the one that minimizes the Euclidean distances along the grasping trajectory is selected.

Obviously, the behavior of this decision process can be easily altered by adding or removing virtual handles of objects or incorporating further rules for the prolog-based filtering of candidate virtual handles. For example, one could add a rule that mugs filled with liquids should not be grasped from within and that if the liquid is hot only the actual handle of the mug is an allowed virtual handle.

As a conclusion, moving the decision making into the designator resolution and basing it on designator properties such as virtual handles allows to keep a high level of abstraction for the plans while also making these decisions transparent for post-execution reasoning.

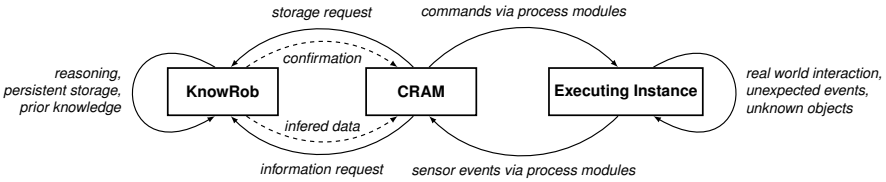


Figure 6. Flow of actions and information between CRAM, KnowRob and real or simulated robots. CRAM emits the control flow and queries both, KnowRob and the executing instance.

4.4 Knowledge Inference in Abstraction Components

The knowledge needed to mediate between high level plans and low level execution is not always included in the plans and must therefore be infered. This is achieved through the KnowRob knowledge system (Tenorth & Beetz, 2012) which also does knowledge processing and reasoning. The communication between the KnowRob and CRAM components takes place through Prolog queries.

An arbitrary number of solutions is generated for each query. Besides inference and reasoning, the KnowRob components can act as persistent storage facility, accumulating knowledge collected by the robot. In terms of our scenario, possible knowledge to store are the known locations of objects. Once the robot has put down an object, storing the current location in the knowledge system helps to find it later on. Before the robot can find an object, it needs at least a vague description of the 3D model data or characteristic colors it is looking for. After finding it and before executing a grasp action, the handle poses for the object in question must be acquired. This kind of data is ideally present in the knowledge base for all available objects so that CRAM components and process modules have this information readily available.

5. Related work

So far, we have presented our view on how to generate flexible and robust robot behavior through a carefully designed reactive high level plan language. Alternatively, other groups have proposed to use hierarchies of state automata as control engines for their cognitive agents. Within these frameworks the currently activated state determines the behavior of the robot, and each of these states may typically contain nested state automata. Internal events or external sensory signals trigger jump conditions which cause state changes. The structure of these hierarchical state automata is usually defined using an explicit (Albu-Schaffer et al., 2007) or implicit (Bohren & Cousins, 2010; Srinivasa et al., 2010) scripting language. Unfortunately, these approaches face a combinatorical explosion of the number of states once one tries to model sophisticated behavior for real-world scenarios. High level languages such as the plan representation used in CRAM, however, represent behavior specification implementations that are far more modular and transparent.

An approach evaluating the idea of modules and task trees is Simmons TCA (Simmons, 1990), which bridges the gap between task-level planners and real-time control systems. While having strong application in navigational and perceptual tasks, TCA tackles no mobile manipulation scenarios. In contrast to TCA, we put more effort in manipulating dynamic environments, which in turn change through the robot's actions or those of external agents.

PRS (Ingrand et al., 1996) reflects a classical *belief-desire-intention* architecture combined with a task graph and a plan library attached to it.

While also relying on goals that have to be achieved and concurrent tasks running besides them, PRS acts as a reactive plan language and mainly focusses on low level, reactive failure handling whereas we introduce multiple competence levels for failure management.

3T architectures state another way of specifying sophisticated robot control programs (Bonasso et al., 1997; Gat, 1998). Typically, these systems consist of 3 layers. On top, there is a layer which represents the desired behavior as a partially ordered sequence of abstract actions. The lowest level, however, is very reactive and provides concurrent skills. Finally, there is the intermittent sequencing layer which is supposed to mediate between the other two parts of the system. Specifically, reactive plan languages (Firby, Prokopwicz, & Swain, 1995; Gat, 1996; Ingrand et al., 1996) have been proposed to act as the middle sequencing layer. Unfortunately, the task of mapping a partially ordered sequence of abstract actions on concurrent reactive skills has proven to be extremely difficult (Simmons, 1990; Simmons, 1994). Thus, we conclude that it is vital to endow the plan language with control structures which reflect the parallel nature of the actions that they shall achieve.

The plan representation employed in CRAM is just such a high level reactive plan language which explicitly supports concurrent actions, while maintaining the modularity and transparency of high level languages. A predecessor of the current CRAM system is RPL (Beetz, 2002), which focussed on navigation plans of a robotic office courier. Several important properties of plan representation languages, e.g. representational and inferential adequacy and representational and inferential efficiency, are proposed, defined and discussed in the light of the given application. The RPL system inherits key features from Firbys RAPs (Firby, Prokopwicz, & Swain, 1995) and further extends this idea as well as basic principles from Schoppers Universal Plans (Schoppers, 1987), like not forcing the initial situation for a certain action. The presented control engine in RPL exhibits several advanced plan management features. These include plan adaptation based on perception of opportunities for behavior improvement and partial order execution of subtasks, which the current CRAM framework does not yet possess. The system presented in our current paper, however, is able to generate and reason about more complex manipulation activities than the one presented in (Beetz, 2002).

6. Conclusion

In this paper, we presented the use of a flexible high level plan environment in a pick and place scenario, which is a common element in robotic applications. The techniques here are implemented in context of the CRAM high level plan framework. Knowledge intense tasks are formulated more independently from the utilized robot architecture by separating robot dependent process modules from plan components. We illustrated propagating failure handling techniques and dynamic plan parameterization through designators as well as a generic representation for virtual object handles. The shown methods allow for reasoning about the currently executed plans and on-demand reparametrization. For later analysis and offline learning algorithms, annotated plan execution traces are recorded during runtime. We believe that robots in the real world have to deal with a much more unstable environment and unforseen situations than in a simulation system.
Therefore, a flexible error handling framework and the ability to reason about vague and ambiguous information is vital for continuous execution of underspecified high level plans.

A solid knowledge representation and persistent knowledge storage as well as reflection about the actions a robot took, raise the success rate in future plan executions.

In future work, we intend a closer connection between the introduced components. We consider building a solid plan library of scenario components with a wide range of applicability. With the presented techniques, human-scale everyday activities, like making pancakes or popcorn, can be modeled. Detailed information necessary for these processes can be reasoned about in a reasoning engine and a flexible failure handling engine covers the most common problems that can come up in real-world scenarios.

Acknowledgements

This work has been supported by EU FP7 Projects RoboHow[1] (grant no. 288533), SAPHARI[2] (grant no. 287513) and the DFG Project BayCogRob within the DFG Priority Programme 1527 for Autonomous Learning.

References

Albu-Schaffer, A., Haddadin, S., Ott, C., Stemmer, A., Wimbock, T., & Hirzinger, G. (2007). The DLR lightweight robot – design and control concepts for robots in human environments. *Industrial Robot: An International Journal*, *34*, 376–385.

Beetz, M. (2000). *Concurrent reactive plans: Anticipating and forestalling execution failures*, vol. 1772 of *Lecture Notes in Artificial Intelligence*. Berlin, Heidelberg, New York: Springer.

Beetz, M. (2002). Plan representation for robotic agents. *Proceedings of the Sixth International Conference on AI Planning and Scheduling* (pp. 223–232). Menlo Park, CA: AAAI Press.

Beetz, M., Jain, D., Mösenlechner, L., Tenorth, M., Kunze, L., Blodow, N., & Pangercic, D. (2012). Cognition-enabled autonomous robot control for the realization of home chore task intelligence. *Proceedings of the IEEE*, *100*, 2454–2471.

Beetz, M., Klank, U., Kresse, I., Maldonado, A., Mösenlechner, L., Pangercic, D., Rühr, T., & Tenorth, M. (2011). Robotic roommates making pancakes. *Proceedings of the Eleventh IEEE-RAS International Conference on Humanoid Robots*. Bled, Slovenia.

Beetz, M., Mösenlechner, L., & Tenorth, M. (2010). CRAM – A Cognitive Robot Abstract Machine for everyday manipulation in human environments. *Proceedings of the IEEE/RSJ International Conference on Intelligent Robots and Systems* (pp. 1012–1017). Taipei, Taiwan.

Bohren, J., & Cousins, S. (2010). The SMACH high-level executive. *IEEE Robotics and Automation Magazine*, *17*, 18–20.

Bonasso, P., Firby, J., Gat, E., Kortenkamp, D., Miller, D., & Slack, M. (1997). Experiences with an architecture for intelligent, reactive agents. *Journal of Experimental and Theoretical Artificial Intelligence*, *9*, 237–256.

Firby, R., Prokopwicz, P., & Swain, M. (1995). Plan representations for picking up trash. *Proceedings of the IEEE International Conference on Tools with Artificial Intelligence*, (pp. 496–497).

1. http://www.robohow.eu/
2. http://www.saphari.eu/

Gat, E. (1996). ESL: A language for supporting robust plan execution in embedded autonomous agents. *Proceedings of the AAAI Fall Symposium Issues in Plan Execution*. Cambridge, MA.

Gat, E. (1998). On three-layer architectures. In P. Bonasso, D. Kortenkamp, & R. Murphy (Eds.), *Artificial intelligence and mobile robots*. Cambridge, MA: MIT Press.

Goldfeder, C., Ciocarlie, M., Dang, H., & Allen, P. K. (2009). The Columbia grasp database. *IEEE International Conference on Robotics and Automation* (pp. 1710–1716). Kobe, Japan.

Ingrand, F. F., Chatila, R., Alami, R., & Robert, F. (1996). PRS: A high level supervision and control language for autonomous mobile robots. *Proceedings of the IEEE International Conference on Robotics and Automation* (pp. 43–49). Minneapolis.

Miller, A. T., Knoop, S., Christensen, H. I., & Allen, P. K. (2003). Automatic grasp planning using shape primitives. *Proceedings of the IEEE International Conference on Robotics and Automation* (pp. 1824–1829). Taipei, Taiwan.

Moore, R. C. (1985). A formal theory of knowledge and action. In J. R. Hobbs, & R. C. Moore (Eds.), *Formal theories of the commonsense world*. Norwood, NJ: Ablex.

Mösenlechner, L., & Beetz, M. (2009). Using physics- and sensor-based simulation for high-fidelity temporal projection of realistic robot behavior. *Proceedings of the Nineteenth International Conference on Automated Planning and Scheduling*.

Mösenlechner, L., & Beetz, M. (2011). Parameterizing actions to have the appropriate effects. *Proceedings of the IEEE/RSJ International Conference on Intelligent Robots and Systems*. San Francisco, CA.

Mösenlechner, L., Demmel, N., & Beetz, M. (2010). Becoming action-aware through reasoning about logged plan execution traces. *Proceedings of the IEEE/RSJ International Conference on Intelligent Robots and Systems* (pp. 2231–2236). Taipei, Taiwan.

Saxena, A., Driemeyer, J., & Ng, A. Y. (2008). Robotic grasping of novel objects using vision. *International Journal of Robotics Research*, *27*, 157–173.

Schoppers, M. (1987). Universal plans for reactive robots in unpredictable environments. *Proceedings of the Tenth International Joint Conference on Artificial Intelligence* (pp. 1039–1046). San Francisco, CA: Morgan Kaufmann Publishers Inc.

Simmons, R. (1990). *Concurrent planning and execution for a walking robot* (Technical Report CMU-RI-TR-90-16). Robotics Institute, Carnegie Mellon University, Pittsburgh, PA.

Simmons, R. (1994). Structured control for autonomous robots. *IEEE Transactions on Robotics and Automation*, *10*, 34 –43.

Srinivasa, S., Ferguson, D., Helfrich, C., Berenson, D., Romea, A. C., Diankov, R., Gallagher, G., Hollinger, G., Kuffner, J., & Vandeweghe, J. M. (2010). HERB: A home exploring robotic butler. *Autonomous Robots*, *28*, 5–20.

Stanford University (2008). Stanford personal robotics program. `http://personalrobotics.stanford.edu/`. Accessed August 16, 2012.

Tenorth, M., & Beetz, M. (2012). Knowledge processing for autonomous robot control. *Proceedings of the AAAI Spring Symposium on Designing Intelligent Robots: Reintegrating AI*. Stanford, CA: AAAI Press.

Plan-based Control for Generalized Fetch and Place Tasks

Jan Winkler* winkler@cs.uni-bremen.de
Ferenc Balint-Benczedi* balintbe@cs.uni-bremen.de
Michael Beetz* beetz@cs.uni-bremen.de

***Abstract*— This paper examines the principle structure and reasoning demands of generalized fetch and place tasks when performed by a cognition-enabled robotic agent. We believe that pick and place plans depend on information that is not necessarily available when starting to perform the task. Consequently, they fail when not considering currently hidden locations behind objects or furniture, in cupboards or drawers. Our approach during a *fetch and place* plan is to concentrate on identifying hidden locations through visibility reasoning and querying an external knowledge source for possible drawer and cupboard locations for an object in question. Following a sketchy plan design approach, we define plans as vague as possible and use the agent's reasoning capabilities to refine location, object and context descriptions with input from knowledge sources, the perception system, and the agent's current belief state as they become available. Hence we call our plans *generalized* fetch and place plans as most information about the object to pick up, it's current and future location, and the task context are omitted in their design. We believe that the profound integration of perceptive abilities into search and manipulation tasks is inevitable in dynamic, human living environments. Thus, we propose a tight coupling between high-level plan control and semantic object annotation through a dedicated perception system.**

I. Introduction

For most robotic service agents variants of fetching objects will be the tasks to be performed most frequently. Consider a robot assistant. If a human forgets to bring the remote control, the robot is to bring it. If the robot is to make pancakes, it has to get the spatula and the ingredients first. If it is to clean the table, it has to take the objects from the table and put them where they belong, or if it is to set the table, the robot has to fetch the needed items and arrange them appropriately on the table. All these tasks require fetching objects.

In this paper we consider the most common of these fetching tasks, namely the fetch and place tasks. In these tasks the robot gets descriptions of objects it has to fetch and of locations where it has to place them. Based on the object descriptions, the robot has to find the intended objects, make them accessible, e.g. by opening doors and drawers, pick them up, and hold them. The robot then has to carry the objects to a location from which it can reach the destinations and place them at the appropriate location.

We call these tasks *generalized* fetch and place tasks because the tasks apply to different kinds of objects, located in different kinds of locations and to be put at various destinations. Typically the objects to be fetched and the locations where they should be placed are only vaguely specified. Therefore the robot has to reason about which objects to get, where to find them, how to handle them, and where to place them in order to accomplish its tasks appropriately. For example, using generalized fetch and place plans a programmer can provide the following concise plan as a means for accomplishing the task "*clean the table*": for all items on the table, put them where they belong. Or for the task "*set the table*": get the needed items and arrange them on the table appropriately. Figure 1 shows the subplans and phases of a generalized fetch and place plan.

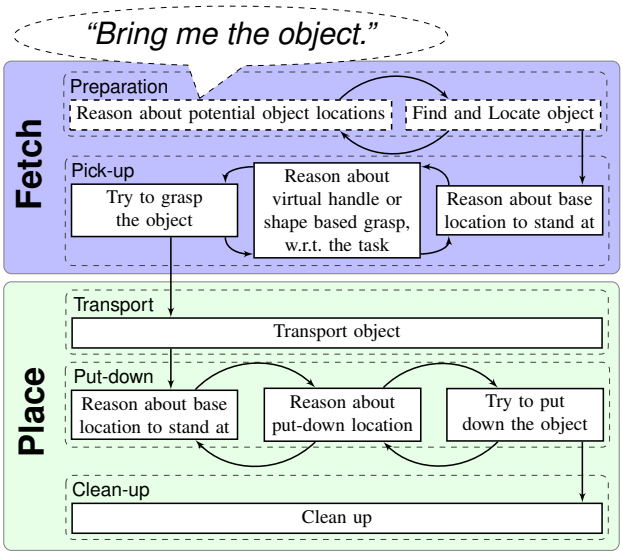


Fig. 1. Flow of steps when executing a generalized fetch and place plan. The fetch and place plans are separated and show the five different phases during plan execution.

Such sketchy plan descriptions [3] provide clues for what the robot is supposed to do. Descriptions such as *all items on the table* are translated into perceived object instances. These are updated with information such as pose, semantic object parts from a knowledge base, and object states whenever new details are acquired. Locations such as *on table* are taken as a hint and with input from a semantic map and a perception system are converted into a bounding box for the specific supporting plane. Such gathered information poses the context for reasoning methods about appropriate plan step parameterizations for the plan execution.

The provision of plan libraries and the execution of plans for generalized fetch and place tasks imposes several research challenges. These challenges include the number and spectrum of control decisions that have to be made, the amount of knowledge and information needed to decide

* Institute for Artificial Intelligence and the TZI (Center for Computing Technologies), Universität Bremen, Germany.

competently, and the perceptual capabilities needed to extract this information from the data provided by the robot's sensors. Figure 2 shows a robot grasping an object in the real world with an overlaid visualization of it's internally simulated belief state.

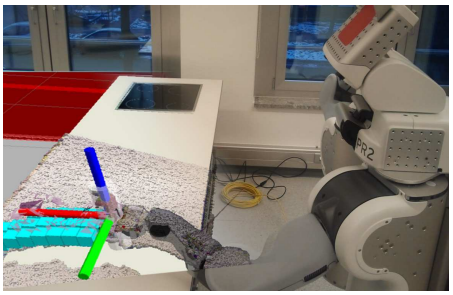


Fig. 2. PR2 robot grasping an object. The real scene is overlaid by the internally simulated equivalent that is used for occlusion reasoning.

In this paper, we (1) propose a plan structure for generalized fetch and place tasks, (2) describe the common knowledge and reasoning methods needed for such plans, and (3) sketch the perceptual capabilities that these plans require.

In the remainder of the paper we proceed as follows. Section II will describe the high-level plan design. In Section III we will then describe some critical mechanisms for context-dependent decision making. Section IV will detail perception capabilities that are needed to make the proposed plans operational. We conclude with experimental results in Section V.

II. Plans for Generalized Fetch and Place

The top-level plan for generalized fetch and place is a simple sketchy plan that asks the robot to fetch the respective object, carry it to a place from where it can reach the destination of the object and place it there.

Fetch and Place tasks consist of five main phases which each have to obey situational and task dependent constraints. For fetching an object, these phases are a *preparation* and a *pick-up* phase. Once the object is in the robot's possession, a *transportation* phase, a *put-down* phase, and a *clean-up* phase conclude the place task.

Within the preparation, the robot perceives the object to pick up and estimate it's pose. Finding an object might include opening drawers, looking into cupboards and navigating around obstacles to get a better view of the environment. When the object was found, picking it strongly depends on the task at hand, i.e. constraints avoiding the obstruction of openings, parts to use for grasping or parts to avoid. Transportation is constrained by possible poses or angles the object has to be kept in or which should be avoided, e.g. for containers filled with liquid which should not be spilled. When putting down the object, enough space on the supporting plane has to be ensured as well as a correct *un-grasping* in order to not damage or misplace the object accidentally. Finally, during clean-up, the robot might need to close a drawer or fridge door to complete the task.

While this plan looks simple it gains its power from the descriptions of the object to be fetched, the description of the destination and the task and scene context descriptions that are maintained. These descriptions can be rather vague in the beginning but get refined and revised as the plan gets executed. The refinement and revision of these descriptions and accompanying failure detection and recovery mechanisms provide the robot with the necessary means to accomplish generalized fetch and place tasks flexibly and robustly.

In order to present the design of the high-level plan, we structure our discussion into a description of the representational structures used by the plan and then according to the main subplans: the fetch plan and the place plan.

A. Representational Structures

Before we detail and explain the plans used by the top-level plan we will first introduce the representational structures used by the plans for reasoning purposes. These representational structures are object, location, scene context, and task context descriptions. The entity descriptions presented hereafter are represented using designator structures for objects, locations, and actions as introduced in [8].

1) Object Descriptions: Object descriptions describe objects in terms of attribute value pairs. For example, the description `(an object ((category pot)))` describes an object of the category pot, `(an object ((category pot) (volume 5l)))` describes a pot that can hold 5 liters, or `(the object ((category pot) (on table)))` specifies a unique pot that can be found on the table.

These descriptions are typical for the way users describe them in tasks given to the robot. They are, however, not sufficient as parameters for low-level robot plans as they do not specify the accurate positions and shapes of the object that might be needed by the low-level plan for grasping the object. Thus while executing the plan the robot has to refine and revise object descriptions to fill in the details that are needed to make the low-level plans operational.

Object descriptions are refined through reasoning and perception. We will explain these methods in Sections III. and IV.

People describe objects in many ways and eventually our robots have to be capable of making sense of all these descriptions. At the moment we have only a limited vocabulary to describe objects. Some attributes that we can already interpret include `(type handle)`, `(type box)`, `(type round)`, and `(color red)`.

Important variants of object descriptions that are on our research agenda are collection categories (e.g. *grocery item*), qualitative spatial descriptions (the object to the right of the plate), object states (the clean plates), etc. A key challenge in extending the expressiveness of object descriptions is that the robot must be capable of perceptually grounding object descriptions. That is, given a captured image the robot has to decide whether an object in view satisfies a given object description.

2) Location Descriptions: Location descriptions describe places at which objects can be found, placed, and where the robot could position itself in order to perform some action.

An important resource for describing and interpreting location descriptions is the semantic object model of the environment [6]. Semantic object maps represent the environment in terms of floors, walls, ceiling and pieces of furniture including cupboards, tables, counters, sinks, and electrical devices such as fridge, oven, or dishwasher. The furniture pieces are hierarchically structured into doors with angles and handles and drawers. Drawers and doors have articulation models that specify how doors and drawers can be opened and closed.

Based on the environment model locations of objects can be described as being on a supporting surface (in particular those contained in the semantic map such as `(on table)`, `(on counter)`, etc.) and inside of container objects (also contained in map such as `(in drawer)`, `(in cupboard)`, etc.).

Location descriptions can also be used to specify the place from which the robot should perform an action. The description

```
(a location ((to execute)
             (action (an action
                         ((to grasp)
                          (obj pot-12)))))))
```

specifies a place from which the object `pot-12` can be picked up.

3) Relevant Scene Context Descriptions: We consider the relevant scene context to be the neighborhood of a location where a manipulation action should be performed. In this case, the neighborhood is defined as the objects that might impact the way the manipulation action is executed. Example: objects that affect the reaching trajectory because the robot has to reach above or around the object.

At the moment we use a simple realization for the relevant scene context, which is the area of supporting surfaces such as tables and counters and the volume of container objects such as drawers and cupboards. That is we assume that all objects on the supporting surface are relevant for fetching any other object on the same surface.

4) Task Context Descriptions: Plans consist of meaningful purposes that, given the robot has all necessary information at it's disposal, are to be reasoned about and fulfilled. We call such purposes tasks that can vary largely. From fetch and place tasks to tidying up a room and pouring liquid into a bowl, all tasks have a certain context that must be taken into account when performing them. As a task is constrained by it's context, the decision in single task steps depend on it. For fetch and place tasks for example, a drawer could be opened to find an object in there. Independent of the actual content, the drawer is closed afterwards. For room tidying tasks, a drawer could be left open intentionally to put more objects in it so that it must not be opened again later on. When objects are to be transported from one place to another, the overall purpose could constrain all parts of a fetch and place task. An object's openings are not to be obstructed when grasping it for pouring tasks and it has to be held steady during transport when there's liquid in it.

Task context descriptions are acquired and reasoned about whenever a plan starts or a monitored action yields an unexpected outcome, failure or new opportunity. An external knowledge source holds rules for tasks while a perception system keeps track of the current state of the environment.

B. The Fetch Plan

The top-level structure of the fetch plan is to first find and locate the intended object, to move into a pose from which the object can be (well) reached, and then pick the object up. This covers the first two phases of a fetch and place task, i.e. *preparation* and *pick-up*. The code for this plan is depicted in Figure 3 and explained thereafter.

```
(def-goal (achieve (object-in-hand ?obj))                          10
  (with-contexts ((tsk-contxt ...)                                 11
                  (scene-contxt ...))                              12
    (with-designators ((obj-acted-on ...)                          13
                       (pick-up-action                             14
                        (an action                                 15
                          `((to pick-up)                           16
                            (obj ,obj-acted-on))))                 17
                       (pick-up-loc                                18
                        (a location                                19
                          `((to execute)                           20
                            (action                                21
                              ,pick-up-action)))))                 22
      (with-failure-handling (...)                                 23
        (find-and-locate obj-acted-on)                             24
        (n-times 5                                                 25
          (at-location (pick-up-loc)                               26
            (perform pick-up-action))                              27
          until (holds (object-in-hand                             28
                        obj-acted-on)))))))                        29
```

Fig. 3. Sample plan for achieving the goal *object-in-hand*. Scene and task context are taken into account. Locations are generated with respect to the actions (*pick-up*) that are to be executed. The *find-and-locate* function takes over reasoning and perception mechanisms for locating the object.

The fetch object plan is structured as follows. First, the context, object, and location descriptions are defined and initialized (lines 11-22). The first plan step is the `find-and-locate` subplan, which is to detect and localize the object denoted by the object description `obj-acted-on`. As a side effect the plan refines the object description of `obj-acted-on` by adding a 3D position attribute and a reconstructed model attribute to the description. Based on the estimated object position the plan step `at-location` infers a suitable position from which the object can be picked up. The `at-location` step (line 23) asks the robot to navigate to this position and stay there until the pick-up (line 27) is completed. The pick-up step is repeated at most 5 times or until the object is in the hand in order to make the plan more robust.

1) The Find-and-Locate Plan: The plan step `find-and-locate` perceives the environment and takes actions towards finding the (possibly hidden) location of the object to pick up.

Locations for where the object might be, can be given within the object description in the form *(at (on kitchen-counter))*. If no direct hints are found in the description, an external knowledge source is queried for locations where this object type or instance is usually found. In case this approach fails, a blind search in the environment with a time constraint is started. If still no object was found after the reserved search time is up, an error is thrown to which a higher level plan can react accordingly. Currently, objects can be searched for either `on` supporting surfaces or `in` containers such as drawers or cupboards, assuming that a perception system can perceive the object when the container is open.

Based on present object location assumptions, the next most probable potential location is checked for the object's presence. Using a *lazy location hypothesis*, the next probable location is only generated in case the last one didn't yield sufficient results. Two information sources are taken into account during this process. From a semantic map of the environment, drawers and cupboards are identified that, according to a knowledge base, could hold the object (based on it's type, or earlier search situations) with a certain probability. This includes opening drawers and cupboards if the object is believed to be inside them. Second, currently occluded regions are identified and searched. Section III-C describes our occlusion reasoning approach.

2) The Pick-up Plan: In order to actually execute a grasping action, several object and context dependent details must be worked out. These include a pregrasp and a grasp pose which depend on the object pose to pick and the robot arm side to use. Prior to the action execution, a path optimal set of arm side/object pose assignments is calculated, iterating over all possible combinations. The optimization algorithm makes sure to only generate valid solutions with respect to collision free poses.

The grasping technique to use when picking up an object depends on the semantic object information available in it's description. If an external knowledge source supplied the plan with information about virtual, graspable handles on the object, these are tried first. Virtual handles are semantic object parts that were either entered manually or identified through reasoning mechanisms based on a model of the object. They include the information about where the handle is and what it's orientation is when grasping it.

Figure 4 shows the selection process for the grasping technique. First, the occurence of virtual handles is checked (line 13). When arms are available for grasping (line 14), the optimal grasp for the object is calculated (line 15). If this technique fails, for example due to missing handles, a grasp based on the object shape is calculated. If both techniques fail, an error is thrown and a higher level structure is for example deciding about a new location to stand at in order to grasp the object.

C. The Place Plan

Within the place plan, a suitable location for the robot to stand is generated from which it can reach the put-down location. After navigating onto that position, the object in the robot's gripper is put down to the designated location. This concludes the fetch and place task as the last three phases, i.e. *transportation*, *put-down*, and *clean-up* are covered. Figure 5 shows the sample code for this action which is afterwards explained.

```
(<- (action-desig ?desig (grasp ?obj ?grasps))                10
  (desig-prop ?desig (to grasp))                              11
  (desig-prop ?desig (obj ?obj))                              12
  (handles ?obj ?handles)                                     13
  (available-arms ?obj ?arms)                                 14
  (optimal-grasp ?obj ?arms ?grasps))                         15
```

Fig. 4. Prolog patterns for selecting an appropriate grasping technique, based on the semantic virtual handle information on the object to grasp. When handles are available on the object, a handle based grasp is calculated.

```
(def-goal (achieve (object-placed-at ?obj ?loc))              10
  (with-contexts ((tsk-contxt ...)                            11
                  (scene-contxt ...))                         12
    (with-designators ((object-acted-on ...)                  13
                       (put-down-action                       14
                        (an action                            15
                          '((to put-down)                     16
                            (obj ,object-acted-on)            17
                            (at ,?loc))))                     18
                       (put-down-loc                          19
                        (a location                           20
                          '((to execute)                      21
                            (action                           22
                              ,put-down-action)))))           23
      (with-failure-handling (...)                            24
        (n-times 3                                            25
          (at-location (put-down-loc)                         26
            (perform put-down-action))                        27
          until (holds (loc object-acted-on                   28
                       ?loc)))))))                            29
```

Fig. 5. Sample plan for achieving the goal *object-placed-at*. Scene and task context are taken into account. Locations are generated with respect to the actions (*put-down*) that are to be executed.

The place plan is structured similar to the fetch plan shown before. After context, object, and location descriptions are defined and initialized (lines 11-23), the robot is advised to navigate near the designated put-down location (line 26). Finally, after arriving there, the actual put-down action is performed (line 27). Effect-aware placement takes the environment and possible obstacles into account, as described in III-B. This process is retried a number of times before yielding an error to be handled by failure handling code.

As with the pickup plan, the putdown plan consists of several phases. After placing the object on the surface of a supporting plane (e.g. a table), an ungrasp algorithm is executed that opens the gripper and moves the robot's arm away from the placed object to not throw it over. The exact ungrasp poses depend on how the object was grasped in the first place, e.g. whether it was grasped on a handle or with a push grasp.

III. Reasoning for Sketchy Plan Execution

Sketchy plans are not specified in a way a robot could directly execute. Instead, extensive reasoning capabilities fill

the sketchy and vaguely described parts with more specific parameters. During generalized fetch and place tasks, object and location descriptions as well as occlusion reasoning, grasp assignment and environmental awareness make up potentially very information rich components. Building on top of McDermott's approach as presented and further extended by Firby in [4], plans are provided that can then be executed on a cognitive agent.

A. Object Description Refinement and Revision

Describing objects in a human natural way means empathizing their dominant features. A robot's plan should therefore be able to handle very vague information about the objects to act on. Descriptions like `((type cup) (in drawer))` should therefore be a starting point and are to be refined during the course of plan execution. The fact that the object is `(in drawer)` advises the robot to navigate near the drawer and look for it's handle. The handle is then grasped and the drawer is pulled according to an articulation model of the container object, saved in the semantic map. It then looks into the opened drawer. Since the description of the object is very unspecific, any cup-like object fits the requirements.

When looking for objects, perception algorithms deliver 3D pose information that is appended to the already available object information. This way, the object description is refined until all information for the task at hand is available.

B. Generating Poses from Location Descriptions

Poses are generated from an ensemble of costmap generators that are chosen according to the location description. Overlapping poses from all involved generators are collected and weighted by how good they fit the description. This way, pose samples for spatial relations can be graded by how much they actually fit the requested properties. A probability distribution over these samples is generated as shown in Figure 6. When a location description is resolved to an actual pose, a sample is drawn from this distribution. If the drawn sample, according to the calling higher level plan, does not fit the intended purpose good enough, a new sample is drawn following the *lazy location hypothesis* as described in Section II-B.1.

Besides spatial relations, such as `on`, `in`, and `left-of`, filter functions like `visible` or `reachable` are present.

C. Occlusion Reasoning

To harness the power of occlusion knowledge, resulting poses for the robot to stand at are vital. Figure 7 shows the resulting poses for the robot to stand at in order to see occluded regions as described in Figure 8. This location distribution is described by "*what are positions where the cup could stand given that I cannot see it and which position could I move to in order to see it*".

D. Reasoning about Suitable Grasps

When picking up an object, the actual grasp depends on the object structure, identified semantic object parts and the intent of the action. After a handled object was identified using it's CAD model and an approximate pose was perceived, a valid grasp technique is to pick the object up on one of it's handles. The relative pose of handles on an object are saved alongside the CAD model in an external knowledge source as described in [7]. For objects without graspable semantic parts such as handles, object shape based grasping methods such as a push grasp must be applied. We evaluate the effectiveness

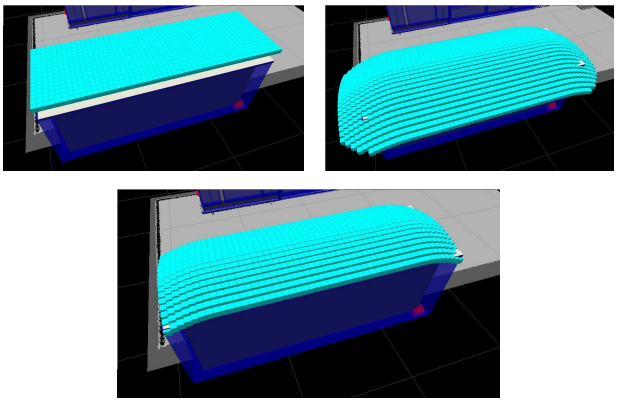


Fig. 6. Two probability distributions generated from the *(on table)* and *(near center)* descriptions, and their merged distribution.

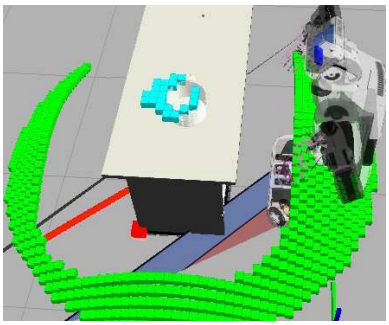


Fig. 7. Possible locations for the robot to stand at (green), in order to see places that cannot be seen behind an object (blue).

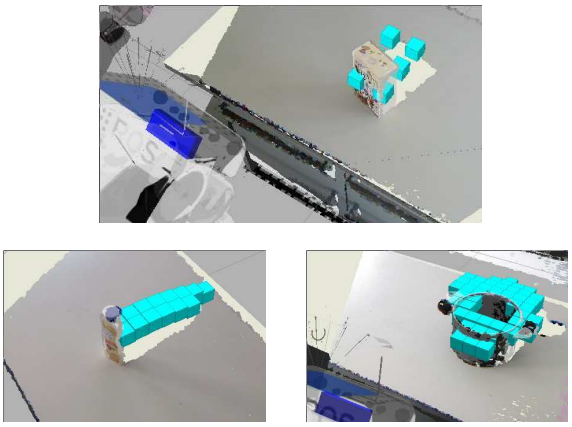


Fig. 8. Probability distribution for locations on a table which cannot currently be seen by the robot that are occluded by an object *obj*. It is described by *((on table) (invisible-for robot) (occluded-by obj))*. The visibility reasoning is based on a virtual instance of the object *obj*. The point cloud for the table top is overlaid.

of such a push grasp technique in Section V.

IV. Perception for Sketchy Plan Execution

As stated in Section III, in order to be able to execute the goals defined in a sketchy plan the robot needs to rely on reasoning. One of the main information sources in order to do this comes from the perception capabilities. The perception system, RoboSherlock, that we are developing [1] is capable of performing these perception tasks, and generates, updates, maintains, and memorizes object descriptions. RoboSherlock is similar to the system presented in [2] in the sense that it aims at filling the gap between semantic perception and knowledge processing capabilities, but tackling the problem with a different approach.

Current capabilities include object detection in semantic regions, object categorization, identifying geometric primitives, CAD model fitting with semantic object part identification, and texture-based object recognition.

RoboSherlock employs ensembles of experts, multiple methods that solve the same tasks but have different strengths and weaknesses, for the different perception tasks. The individual methods come from our previous work or are open source algorithms that we have integrated into our system.

RoboSherlock allows us to keep track of objects that we have encountered during a task execution, annotating them with their location in the semantic map. During plan execution we are able to query for the last seen position of a certain object, and therefore prioritizing the search regions in the environment. For example if the robot is looking for an object that is usually found in the fridge (e.g. a milk box), but has seen it on the table shortly before it would first look for it at the previous location and only then search in the usual places (e.g. the fridge). Having a scene memory also allows us to formulate more complex queries, asking about other aspects of an object, e.g. color, and shape.

Another aspect of executing a fetch and place task is to successfully identify places in the scene where an object can be safely placed. We find these so called 'free spaces' by looking for supporting planes in the environment which have sufficient space for placing the object in hand and are reachable by the robot.

Besides these more abstract perceptual tasks, as mentioned earlier, RoboSherlock also implements individual methods for solving specific tasks, making use of results from multiple similar algorithms in order to come up with a better solution. Some of these experts deliver information that is appended to the already available object, thus refining the existing description. Others enable reasoning for aspects of the task at hand. For example, a CAD model fitting algorithm fits 3D models to objects, making it possible to compute exact 6DOF poses, but also to identify regions of the scene that are occluded by the object at hand as shown in Figure 6.

A. Semantic Location Constraints

An overall situational awareness makes the robot competent while executing it's tasks. A multitude of different constraints originate from knowledge bases, context and perception. Physical constraints can result in a rich description of potential object locations, as in

```
(a location `((on table)
              (invisible-for robot)
              (occluded-by ,obj-1)
              (near center)))
```

When evaluating the different location descriptions, a *location costmap* is generated for each. Overlaying these results in a probability distribution from which locations can be sampled that satisfy the given location description. An example of two overlaid distributions is shown in Figure 6.

Location descriptions can be enriched with information for very specific situations. In Figure 8 a probability distribution is shown for locations on the table which cannot be seen from the current location because they are occluded by a certain object.

In order to generate possible location solutions for potentially invisible places, a semantic map of the current environment is used. This map gives a coarse hint to where such a location is. The visibility is then calculated using an extension of the *bullet reasoning system*, as introduced by Mösenlechner et. al., in [5].

B. Perception-Enabled Environment Awareness

On the basis of prior information from the semantic map and simulated visibility, missing information from the environment is identified. When starting a pick and place action, a cognitive agent's first task is to find the object in question. The location of the object is either visible, invisible or the object is not existing within the search region. In case of direct visibility, the process is complete. When no visible instances of the object can be found, perception-supported algorithms might find one in drawers, cupboards, or in currently invisible locations. Knowing what cannot be seen at the moment identifies the area that must be inspected.

As active manipulation within the environment must be precisely controlled, correct poses for objects and drawer handles are necessary to do grasping. Since semantic maps only allow for an approximation towards reality and strongly depend on a very accurate localization, a matching is done when looking for specific parts. Figure 9 shows the match of drawer handles from a semantic map with perceived handles from the RoboSherlock system.

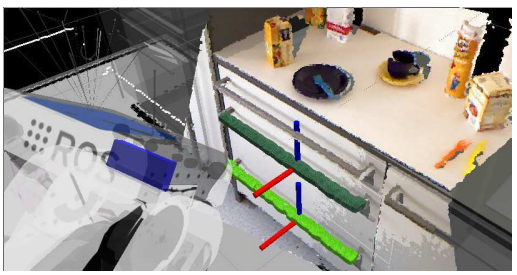


Fig. 9. Matching of semantically defined drawer handles with real handle instances, detected using the RoboSherlock perception system.

Since object instances are not always explicitly mentioned in a semantic map and their location can change through interaction with the robot, or through manipulation by an unknown external agent, the belief about them is updated through the ROBOSHERLOCK perception system whenever neccessary.

Perceived object from the perception system are then reflected in a simulated environment using their CAD models. Objects for which no CAD models are available are represented by a box or a cylinder shape of approximate dimensions.

V. EVALUATION OF PUSH-GRASPS ON DIFFERENTLY SHAPED OBJECTS

When relying on naive grasp techniques like push grasps for picking up unknown objects, the direction from where the gripper is to approach the object highly impacts the success rate. Figure 10 shows the estimated pose of an unknown object, based on depth information from a 3D camera and grasps from the front and side. The visible surface portion of the object identifies the plane center point while not supplying the three-dimensional object centroid.

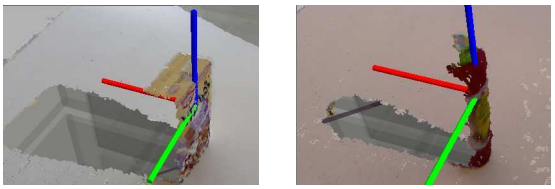


Fig. 10. Object centroid for unknown objects is placed on the visible surface portion. Front grasps are more successful than side grasps as they don't rely on correct centroid depth information for enclosing objects when grasping.

Table I shows the success rate for differently shaped objects with unknown proportions. Their pose is estimated using depth information. Front and side grasps are applied and successful grasps are denoted.

TABLE I

SUCCESS RATE FOR FRONT AND SIDE GRASPING OF UNKNOWN OBJECTS BASED ON DEPTH INFORMATION OF THEIR VISIBLE SURFACE.

Object Type	Side Grasps			Front Grasps		
	Total	Success	Rate	Total	Success	Rate
Round	30	21	70%	30	26	87%
Rectangular		16	53%		25	83%

Both, picking up and transporting an object can be constrained by the current task context and the object state. For pouring tasks, a grasp must not obstruct the object's opening. For objects that contain liquids, like a cup or a pot, the object must not be tilted over a certain maximum angle to not spill the contained material.

VI. DISCUSSION

The concept of Generalized Pick and Place plans is applicable to a wide range of tasks, including manipulation goals in human living environments. When performing cooking tasks in a kitchen, ingredients must be collected and put near the cooking place. Doing shopping in a supermarket environment with unknown object setup heavily relies on perception driven manipulation. Tidying up a room is ultimately made up of pick and place actions that are not further specified.

The baseline information necessary for performing all of these tasks is where the objects are or could be, what their exact pose is and whether their grasping semantics are known or must be approximated. The presented techniques and results are directed towards solving these challenges. Finding objects by identifying still unknown and uninspected locations gives a rough idea where to look and a closer inspection using the perception system gives hints about their pose, their state, and kind, supplying information and constraints for grasping algorithms.

With a supporting semantic map, hidden locations such as drawers and cupboards can be identified and included in the search region. Opening doors or drawers itself heavily relies on handle detection and grasping algorithms, enabling a transfer of the presented techniques into this application.

The concept of virtual handles on objects enables a cognition-enabled agent to connect certain grasps on an object with semantic meaning for the current task at hand. Having a bottle with two virtual handles, i.e. one in the middle and one on the cap, both could be used for carrying while the latter is prohibited for pouring tasks. Drawers and cupboards have handles with an associated articulation model for opening them, relying on a very similar representation as object part handles and using the same grasping mechanisms.

The sketchy character of our plan designs and the executing reasoning mechanisms make it possible to give short and vague task descriptions to the agent. Our generalized plans specialize themselves upon execution based on the task and context information which is updated continuously by the perception system and the agent's belief state. Specialized situations like the decision of whether to use a handle on an object or not and in case, which handle to use with respect to the current task, are implicitly reasoned about when executing a plan. This technique leaves open a lot of other incidents and situation-dependent plan alterations which are not yet covered by our system. These include different radii of handles, maximum pressure to exert on any given handle, how many grasped handles are necessary to support the weight of an object and what handle combination on multi-handle objects is the most reliable one with respect to grasp stability. On the basis of our system, we believe that a wide range of reasoning mechanisms about details that go without saying help towards a competent robot behavior in everyday human-scale activities.

VII. FUTURE WORK

To identify dynamic properties of objects, specialized annotator algorithms need to combine manipulative and perceptive actions. By dynamic properties, we mean characteristics of an object that can be altered over time, e.g. a bowl can be empty or full, dirty or clean, and a book could be open

or closed. Using these dynamical properties, states of the objects can be inferred and used as constraints for plan generation (e.g. *pick up the dirty cup*). In future work we intend to investigate perception techniques for identifying different states of an object as well as how these states affect the task context and therefore the plan execution.

ACKNOWLEDGEMENTS

We would like to thank Ross Kidson for his work on handle detection code.

This work has been supported by EU FP7 Projects RoboHow[1] (grant no. 288533), SAPHARI[2] (grant no. 287513), ACAT (grant no. 600578), and the DFG Project BayCogRob within the DFG Priority Programme 1527 for Autonomous Learning.

REFERENCES

[1] Michael Beetz, Nico Blodow, Ferenc Balint-Benczedi, Florian Seidel, Christian Kerl, and Zoltan-Csaba Marton. Roboscherlock: Unstructured information processing for robot perception. In *Robotics: Science and Systems Conference (RSS)*, Berlin, Germany, May 24–28 2013. Submitted for review.

[2] Michael Beetz, Nico Blodow, Ulrich Klank, Zoltan Csaba Marton, Dejan Pangercic, and Radu Bogdan Rusu. CoP-Man – Perception for Mobile Pick-and-Place in Human Living Environments. In *Proceedings of the 22nd IEEE/RSJ International Conference on Intelligent Robots and Systems (IROS) Workshop on Semantic Perception for Mobile Manipulation*, St. Louis, MO, USA, October 11-15 2009. Invited paper.

[3] J. Firby. Adaptive execution in complex dynamic worlds. Technical report 672, Yale University, Departmentof Computer Science, 1989.

[4] R.J. Firby, P.N. Prokopwicz, and M.R. Swain. Plan representations for picking up trash. *IEEE International Conference on Tools with Artificial Intelligence*, pages 496–497, 1995.

[5] Lorenz Mösenlechner and Michael Beetz. Fast temporal projection using accurate physics-based geometric reasoning. In *IEEE International Conference on Robotics and Automation (ICRA)*, Karlsruhe, Germany, May 6–10 2013. Accepted for publication.

[6] Dejan Pangercic, Moritz Tenorth, Benjamin Pitzer, and Michael Beetz. Semantic object maps for robotic housework - representation, acquisition and use. In *2012 IEEE/RSJ International Conference on Intelligent Robots and Systems (IROS)*, Vilamoura, Portugal, October, 7–12 2012.

[7] Moritz Tenorth, Stefan Profanter, Ferenc Balint Benczedi, and Michael Beetz. Taking a common-sense look at objects of daily use – decomposing cad models into their functional parts. In *Proceedings of the 26th IEEE/RSJ International Conference on Intelligent Robots and Systems (IROS)*, Tokyo, Japan, November 3–7 2013. Submitted for review.

[8] Jan Winkler, Georg Bartels, Lorenz Mösenlechner, and Michael Beetz. Knowledge Enabled High-Level Task Abstraction and Execution. *First Annual Conference for Advances in Cognitive Systems*, 2(1):131–148, December 2012.

[1] `http://www.robohow.eu/`
[2] `http://www.saphari.eu/`

Movement-aware Action Control — Integrating Symbolic and Control-theoretic Action Execution

Ingo Kresse and Michael Beetz
Intelligent Autonomous Systems Group, Technische Universität München
Boltzmannstr. 3, D-85748 Garching, {kresse, beetz}@cs.tum.edu

***Abstract*— In this paper we propose a bridge between a symbolic reasoning system and a task function based controller. We suggest to use modular position- and force constraints, which are represented as action-object-object triples on the symbolic side and as task function parameters on the controller side. This description is a considerably more fine-grained interface than what has been seen in high-level robot control systems before. It can preserve the 'null space' of the task and make it available to the control level. We demonstrate how a symbolic description can be translated to a control-level description that is executable on the robot. We describe the relation to existing robot knowledge bases and indicate information sources for generating constraints on the symbolic side. On the control side we then show how our approach outperforms a traditional controller, by exploiting the task's null space, leading to a significantly extended work space.**

I. Introduction

Over the last decades a number of control paradigms have been developed for specifying the activities to be performed by autonomous robots. These paradigms include behavior-based control [1], emergent and developmental approaches [2], control-theoretic [3], [4] and symbolic methods [5]. Within this spectrum of control paradigms the control-theoretic as well as the symbolic approaches aim at engineering high-performance task execution of complex real-world tasks. However, both approaches differ substantially with respect to their level of abstraction and the conceptual apparatus they use for action specification.

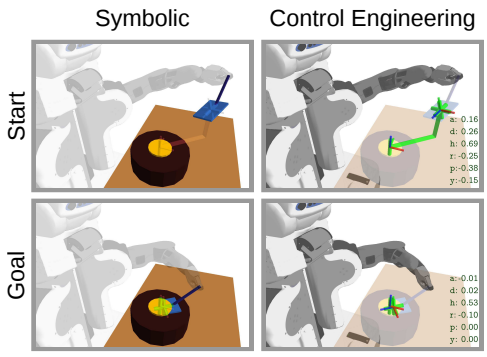


Fig. 1. Illustration of symbolic and control engineering approaches. A symbolic approach on the left relates objects to each other using abstract concepts. A control approach on the right specifies desired tool positions in terms of the joint angles of a 6-DOF virtual kinematic chain. For a detailed description of the coordinates, refer to section III-A

Symbolic approaches enable the programmers to specify tasks in terms of objects that are to be manipulated and used, the desired effects and undesired side effects. They allow to specify actions such as "push the spatula under the pancake in order to flip the pancake." As a consequence, the robot can reason about what it is doing, how, and why. They are, however, unable to specify fine-grained motions or reactions to deal with disturbances and are therefore not able to ensure stable, controllable, and optimal movement performance.

Control engineering approaches have complementary strengths and weaknesses. They specify actions as fine-grained movements by specifying coordinate systems or frames and geometric-, motion-, and force constraints on how frames should move with respect to each other. In a number of control frameworks, such as iTaSC [6], the programmer can then ensure that the frames move in a stable, controllable and optimal way. The conceptual apparatus that these frameworks provide does however not allow us to talk about the effects of movements on object states.

The need for integrating the two paradigms has been recognized for a long time and researchers have proposed a number of software architectures that featured three layers of abstraction, often called 3T architectures. The basic idea behind 3T architectures is simple. The architectures use symbolic action specification and execution at the top layer and control engineering methods at the bottom layer. As both paradigms do not fit properly together, the researchers have introduced an intermediate layer for bridging the differences. This intermediate layer is typically implemented by a reactive situated task management system, which provides the means for triggering task execution, for making task execution sensible to the situational context, for managing concurrent task execution, and for achieving robustness in the presence of task plan failure.

3-layer architectures [7] constitute a shallow integration that enables researchers to keep their symbolic and control engineering approaches without changing them. For obvious reasons, this shallow integration turned out to be of limited use. As the abstract symbolic layer does not have the notions of concurrency, situatedness, failure and failure recovery, there is no way how this layer can understand the behavior generated at the lowest level and thereby diagnose the weaknesses in its intended course of action. In addition, the abstract layer does not provide the control structures needed for making the behavior flexible, robust, and concurrent.

A promising alternative is to couple the symbolic and the control layer directly by means of shared data structures that specify movements as first-class objects. As constraint-based specifications are successful representations both in artificial intelligence and in control engineering, we use constraint-based representations as an interlingua.

For bridging the gap between the symbolic side and the control side, we propose a constraint based movement specification that is fine-grained, modular and transparent. It avoids the over-specification of tasks, and thus retains the null space that can be exploited by the robots control level. This leads to more dexterity and an increased effective work space.

The rest of the paper is structured as follows: In Section II, we give an overview of our system and introduce a constraint specification which is general enough to represent common household tasks and transparent enough to be reasoned about by a high level controller. Section III explains our implementation in detail. In Section IV we apply our approach on an everyday manipulation task, namely pancake flipping with a spatula and execute this task on a robot. This demonstration is evaluated in Section V before we conclude in Section VI.

II. System Overview

Movement-aware action control is performed through the interaction of three system components:

1) the high-level action specification that is extended with means for movement specification;
2) the low-level constraint-based movement specification; and
3) the component that maps the high-level movement specification onto the respective low-level one.

In the remainder of this section we will describe these components in more detail.

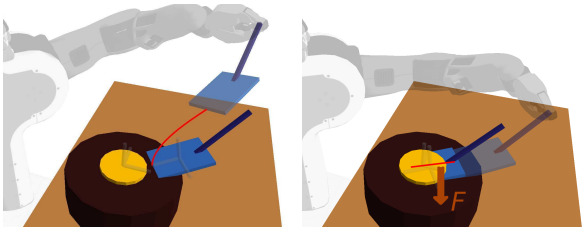
A. High-level Action Specifications

At the high level actions are typically described very abstractly in terms of objects, types, properties, qualitative spatial relations like 'on', or 'towards' and the effects that are to be achieved.

Take the example of pancake making: The robot has to push its spatula under the pancake in order to flip it. The action has to be performed in a way that the robot can lift the pancake safely and such that the robot does not damage the pancake.

The movement needed to perform this action successfully is complex and needs a lot of sophistication. Kunze et al. [8] discuss some issues in reasoning about appropriate action parameterizations to achieve the desired and avoid the undesired effects. Here we want to concentrate on the appropriate representations of the required movements and how these abstract movement specifications can be mapped on to task-function control systems.

Before we talk about the representation of movements let us first see what kind of movement is needed to push the spatula under the pancake in a competent manner. Figure 2 shows the movement which consists of two phases: first, approaching the pancake oven with the spatula until the spatula is in contact with the oven. Second, moving the spatula towards the center of the pancake. For the second movement phase the robot should firmly push the spatula onto the oven to make sure that it is getting under the pancake without damaging it. When touching the pancake the robot should make a small quick move to separate the pancake from the oven it is sticking to.



(a) Approach towards the pancake. (b) Pushing the spatula under the pancake.

Fig. 2. Example task of push the spatula under the pancake.

This sophisticated movement is specified in the following movement representation that we design, develop and investigate in this paper:

```
(perform
  (movement-plan
    (:tag approach-oven
      (a movement-phase
        (object-moved ?blade)
        (destination (at (task-end approach-oven))
          (a pose
            (next-to pancake)))
        (object-orientation (throughout approach-oven)
          (an orientation
            (pointing-towards (center ?pancake))))
        (object-orientation (throughout approach-oven)
          (an orientation
            (horizontal)))))
    (:tag move-under
      (a movement-phase
        (object-moved ?blade)
        (destination (at (task-end move-under))
        (a pose
          (under pancake)))
        (force (throughout move-under)
          (onto oven))))
    :order-constraints
      (:after move-under approach-oven (:allow superposition))))
```

This movement plan consists of the two phases approach-oven and move-under, both phases move the blade of the spatula. The first phase moves the spatula to a pose next to the pancake while respecting the additional constraints of pointing the spatula towards the pancake and keeping it horizontal. The second phase specifies a goal under the pancake with the additional constraint of applying a force onto the oven. The phases of this example are strictly ordered.

This language symbolically describes a movement as a (partially ordered) sequence of phases which in turn are sets of constraints.

A constraint is described by a hand, tool or in-hand object that should be moved, a preposition like next-to or above and an object relative to which the hand/tool shall be positioned.

A special constraint is called destination. Technically it is treated as a normal constraint, it is just a label for the most important aspect of the movement phase. This distinction is

intended for reasoning about the movement plan.

Each constraint has a tag that specifies whether it shall be fulfilled during the whole movement phase or only at the end of the movement. Although this tag is not essential for the controller specification, it is useful for monitoring: When a constraint that shall be fulfilled during the movement is violated at some point, then this is an indication that the movement is not executed correctly.

B. Low-level Constraint-based Movement Specification

A controller that executes such a movement description must synthesize a control law, given the two objects and the action verb. It then must execute a set of such constraints simultaneously and run a sequence of such constraint sets.

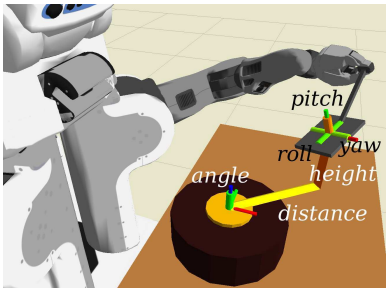


Fig. 3. Virtual linkage with annotated axes.

In order to fulfill these requirements we use a controller based on the task function approach: One defines a function that maps the robots pose to task relevant coordinates like distances or angles between scene objects. In addition, the derivative (Jacobian matrix) of this function is computed, which describes how the robot movements affect these coordinates. Using gradient descent, the robot is then controlled to the desired values of its task coordinates.

For example, one task-relevant coordinate could be the height of the spatula over the oven. For moving down the spatula onto the oven, the height of the spatula is computed in every control cycle. Also, the derivative of the height w.r.t. robot joint movements is computed numerically. Using this derivative, the robot moves its joints in a way that moves the spatula down as fast as possible.

This approach allows us to specify a robotic task in an arbitrary (possibly low-dimensional) coordinate system and keep a notion of the task's null space, i.e. the robot movements that do not affect any task coordinates. This null-space can then be exploited by the robot for secondary tasks like obstacle avoidance or joint limit avoidance. The extra degrees-of-freedom of redundant manipulators are taken into account just as naturally.

For our tasks, we decided to span a 6-DOF virtual linkage between two objects (see Fig. 3) and control its virtual joints over time. For the first three joints we use cylinder coordinates for positioning the spatula tool w.r.t. the oven. This choice is motivated by the rotational symmetry of the oven. For orienting the tool we chose RPY angles which roughly correspond to the tasks of aligning the front edge, aligning the side edge and pointing direction.

A virtual joint can be "switched off", to indicate that its position is not important for the current task and can be exploited by the secondary constraints. Additionally, the virtual joints can be given a desired range of positions. As long as the virtual joint is inside this range, the secondary constraints can exploit this degree of freedom.

On this virtual linkage level we represent a constraint as a triple of $positions$, $forces$ and $weights$ where $positions \in \mathbb{R}^{2\times 6}$ are six ranges of allowed positions, $forces \in \mathbb{R}^{2\times 6}$ are six ranges of allowed forces and $weights \in \mathbb{B}^6$ are six Boolean values describing whether a joint is important at the moment.

C. Translation

In order to translate a constraint from the symbolic high level description into virtual linkage terms, the following tasks must be solved:

- extract from the action type whether it is a position- or a force constraint
- associate object symbols with perceived objects in the scene.
- extract from the action symbol a) the linkage joints to use and b) the desired joint positions/forces or ranges of positions/forces.

The interpretation of the action type is solved by a simple mapping (action-type $\rightarrow$ position, force).

The association of object symbols to the scene is assumed to be done by the perception system and has been studied before [9]. We assume that for every object we can determine its location and extents (see Figure 4).

For a more fine-grained object examination, geometrical features like edges or plane segments can be extracted from the objects, as demonstrated in [10]. This allows us to refer to specific object parts and allows more precise movement specifications.

In order to find the linkage joint we first semantically annotate the joints of the virtual linkage. The first three axes of the linkage, which form cylinder coordinates, are labeled "angle", "distance" and "height", while the last three axes can labeled "roll", "pitch" and "yaw".

In general, a symbolic constraint may imply constraints on several virtual linkage axes. For example, a constraint to keep the spatula above the oven, maps to a) keeping the

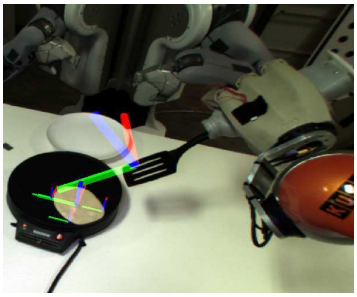


Fig. 4. Localization of objects and tools.

height greater than zero and b) keeping the radius smaller that the radius of the oven.

Each symbolic constraint is decomposed into one or more elementary constraints which consist of an axis name and a comparison operator (smaller, bigger, etc.). After looking up the current positions of both object parts, the ranges for the desired positions and forces are generated.

Some symbolic constraints may not contain a second object. In this case we assume that they can be completed on the symbolic level using either context information or default values.

A complementary method is to generate positions for objects based on constraints like 'free' or 'stable' using a physics engine [11]. However, our focus lies on generating executable robot motions rather than only positions.

III. Implementation

In this section we present our control system for constraint-based movements. We first motivate how we construct the task coordinate system using a virtual linkage, explain how we control the joints of this linkage to desired ranges and apply forces by exploiting the impedance control of our robot arms. We then describe how we exploit the null space of this task to avoid robot joint limits and finally illustrate how the symbolic constraints are interpreted by our controller.

For the sake of simplicity we concentrate on tasks with one object and either one hand or tool. However, the underlying control system iTaSC has been shown to support multi-robot scenarios [4], so an extension to two-arm tasks is possible.

A. Virtual-Linkage Specification

For defining the virtual linkage task we first have to define coordinate systems for the tool (or hand) and the object. It is placed in the center of the object or tool, onto symmetry axes (if present) and is aligned with important plane- or line segments.

Next, we span one of two common linkages: For objects that have an axis of symmetry we use a combination of cylinder coordinates and roll-pitch-yaw, otherwise we use Cartesian coordinates with roll-pitch-yaw. These coordinate systems uniquely cover the six degrees of freedom in the transform between the two objects. Each of these axes is labeled with a name for linking with the higher level constraints: We use 'angle', 'distance' and 'height' for the first three axes that form cylinder coordinates and 'roll', 'pitch' and 'yaw' for the last three axes which define the orientation.

In order to execute more complex tasks, several of these chains can be combined and executed simultaneously. If some constraints are conflicting, then a hierarchy can be built in order to execute one constraint in the null space of the other. Instead of a 6-dof-chain, it is also possible to specify different constraints like the alignment of two line segments. For our validation task, however, one 6-dof chain was versatile enough.

B. Controller Design

We then control this virtual linkage by specifying desired ranges for its axes. This approach is motivated by the fact that many symbolic constraints map to inequality expressions, e.g. "keep the spatula above the oven" implies that the height of the spatula should be greater than the height of the oven. Such inequalities can be solved elegantly with more specialized methods as in [12]. However, a simpler approach was sufficient for our scenario.

On this low (virtual linkage) level we represent a constraint as follows:

$$constraint_{ll} = \langle positions,\ forces,\ weights \rangle \tag{1}$$

where $positions \in \mathbb{R}^{2\times 6}$ are six ranges of allowed positions, $forces \in \mathbb{R}^{2\times 6}$ are six ranges of allowed forces and $weights \in \mathbb{B}^6$ are six Boolean values describing whether a joint is important at the moment.

For every task axis with the current position y and the allowed range $[y_{lo}, y_{hi}]$ we implement the position constraints as a P-controller with a 'dead zone' inside the allowed range. Additionally, when y is inside the allowed range, we lower the weight w of the constraint so that lower level tasks can exploit that axis:

$$\dot{y} = \begin{cases} K_p(y_{hi} - m - y) & : \ y > y_{hi} - m \\ K_p(y_{lo} - m - y) & : \ y < y_{lo} + m \\ 0 & : \ \text{otherwise} \end{cases} \tag{2a}$$

$$w = \begin{cases} 0 & : \ y_{lo} + m < y < y_{hi} - m \\ 1/m(y - y_{hi}) + 1 & : \ y_{lo} < y < y_{lo} + m \\ 1/m(y_{lo} - y) + 1 & : \ y_{hi} - m < y < y_{hi} \\ 1 & : \ \text{otherwise} \end{cases} \tag{2b}$$

Here, y is the current task angle, $\dot{y}_p$ is the desired speed of the task angle, K_p is a plant-dependent control coefficient and m is a margin by which the controller shall push inside the allowed range. This is visualized in Figure 5-left. The null space that is defined by such a task significantly extends the elbow null space that is already present in our KUKA LWR arms.

The weight w is used when the $weight$ of the constraint is true - otherwise it is set to zero which completely disables the constraint.

The parameter m smoothes the jump that would occur due to the sudden deactivation of a constraint and is of the size

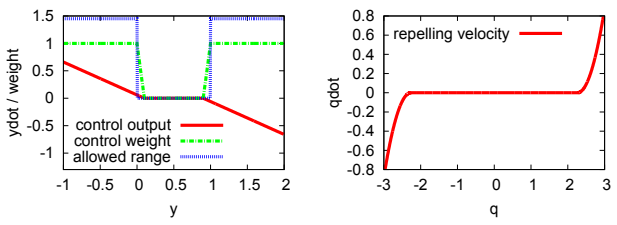


Fig. 5. Left: Controller that keeps a task angle inside a predefined range. Right:Behavior of the joint limit avoidance running in the null-space of the task

of the inaccuracies that are introduced by the joint friction in the impedance controlled robots. The parameter K_p is a proportional gain coefficient and is chosen as high as possible without introducing oscillations.

Using the task's Jacobian matrix (i.e. the derivative of every task angle w.r.t. movements around every joint axis of the robot) the controller can transform the desired task velocities into robot joint velocities.

Since our robot has impedance control, we can apply forces by offsetting the position. The required offset is computed using the controllers spring constant K_F.

$$\Delta y_f = K_F(F_{desired} - F_{measured}) \tag{3}$$

For applying a force on a given object, we propose the following approach: The position range has priority over force and limits the arm movements. When the actual task angle is inside the allowed range, then an offset is added to the desired task angle until either the desired force is reached or the allowed position range is left.

C. Null-space constraint

In order to exploit the null space of the task we use a simple joint limit avoidance task. The joint limits are only avoided when the joint angle is within a distance of d from the limit, the maximum repelling velocity is h and the function is continuous and differentiable over the whole joint range.

A piecewise quadratic function which is depicted in Figure 5-right fulfills these requirements:

$$\dot{q}_a = \begin{cases} {}^{h}/_{d^2}(q_{lo} - q)^2 & : q < q_{lo} + d \\ -{}^{h}/_{d^2}(q_{hi} - q)^2 & : q > q_{hi} - d \\ 0 & : \text{otherwise} \end{cases} \tag{4}$$

We project the resulting joint velocity into the null space of the task using the tasks Jacobian $\mathbf{J}$:

$$\dot{\mathbf{q}}_{proj} = (\mathbf{I} - \mathbf{J}^{+}\mathbf{J})\dot{\mathbf{q}}_a \tag{5}$$

There are more capable methods for joint limit avoidance in [13] or [14] but this method was sufficient for the given task.

D. Translation

On the high level side we obtain a triple (action, object1, object2), where object2 can be empty and is completed with the object 'world', representing merely the vector of gravity.

We then perform a synonym look-up on the action words in order to simplify the next step.

Then we map the action to a set of elementary constraints, each consisting of an axis, a comparison and a margin). Here, axis is the name of a virtual linkage joint, comparison is one of $\{>, <, =\}$ and margin is a real number. This expresses that the first object's value of axis should be higher/lower/equal to the second object's value by a certain margin.

In order to compute the desired range we take advantage of one more piece of knowledge that is commonly available to a robot: The sizes of the objects involved. For instance, if two objects have an extension in the vertical direction and and one object should be above the other, then the sizes need to be examined in order to compute the correct distance. This can be computed by simple interval arithmetics.

The resulting intervals from these primitive constraint triples are then stored in the six ranges and weights so they can be executed by the controller.

This intermediate layer – still mostly symbolic – expresses knowledge that is hard to find in natural language resources. It is thus specified by hand, but it could be an interesting learning task using annotated tracking data from human demonstrations.

The low level specification allows to perform a simple check whether two constraints c_1 and c_2 are compatible:

$$\begin{aligned} compatible_i(c_1, c_2) = \quad & pos_i(c_1) \cap pos_i(c_2) \neq \{\} \\ & \vee weight_i(c_1) = 0 \\ & \vee weight_i(c_2) = 0 \quad \forall i \in 1, ..6 \end{aligned} \tag{6}$$

Similarly, two constraints can be merged by intersecting its position and force ranges and and-ing their weights.

When the reference frames of the two constraints differ, then this test can be computed probabilistically: For random poses, the constraint Jacobians $\mathbf{J}_1$ and $\mathbf{J}_2$ are determined and their ranks are computed. When the condition

$$rank(\mathbf{J}_1) + rank(\mathbf{J}_1) = rank([\mathbf{J}_1\mathbf{J}_2]) \tag{7}$$

holds then the constraints c_1 and c_2 are not conflicting. They may still interfere with each other (otherwise, the columns of $\mathbf{J}_1$ need to be orthogonal with all columns of $\mathbf{J}_2$). However, the rank condition implies that all the constraints are instantaneously solvable.

IV. Example Task

In order to evaluate this framework, we consider the example task of flipping a pancake with a spatula. Assuming that the robot has its tool already in the hand and is standing in front of an oven with the pancake on it. We specify the following symbolic constraints:

c_p = (constraint point-towards spatula oven)
c_h = (constraint keep-horizontal spatula)
c_n = (constraint move-next-to spatula pancake)
c_u = (constraint move-under spatula pancake)
c_l = (constraint lift, spatula)
c_o = (constraint keep-over spatula oven)
c_f = (constraint flip, spatula)

These constraints are combined into four steps which need to be executed:

$s_1 = (c_p\ c_h\ c_n)$
$s_2 = (c_p\ c_h\ c_u)$
$s_3 = (c_h\ c_o\ c_l)$
$s_4 = (c_o\ c_f)$

where the steps s_1 and s_2 correspond to the movements approach-oven and move-under in section II. Each step is completed when its constraints are satisfied. We use the following relations to translate these constraints into elementary constraints (ec) which in turn are converted to virtual linkage coordinates:

point-towards	→	'((ec pitch = 0 0.2))
from-left	→	'((ec angle = 0.5π 1.0))
move-under	→	'((ec height $<$ 0 0) (ec distance = 0 0))
move-over	→	'((ec height $>$ 0.15 0) (ec distance $<$ radius(obj2) 0))
move-next-to	→	'((ec height = 0 0) (ec distance = 0.03 0))
keep-horizontal	→	'((ec roll = 0 0.05) (ec yaw = 0 0.05))
lift	→	'((ec height $>$ 0.2 0))
flip	→	'((ec roll = 0 0.15) (ec yaw = 0.6π 0.1))

These mappings specify a set of 4-tuples: A task axis name, a relation, a fixed offset, which is necessary for constraints like 'flip' and finally a tolerance. The numbers had to be completed manually. However, given the rest of this description, it might be possible to extracted them from human demonstrations.

In this list of actions there are two classes: Actions that relate two objects with respect to each other and actions on one object. For the second class, the "world" could be filled in.

For some actions, the verb move could be replaced by keep. In our system this leads to the same constraint, but this distinction can be a hint to monitoring: move expresses that the constraint is violated at the beginning and holds at the end of the movement. keep, on the other hand, expresses that the constraint should always hold. If the actual behavior is different, then this is a hint that the execution is not running smooth.

Some actions are underspecified in this verbal form. The action turn, for instance, just specifies some rotation around the object center. With the context knowledge, that gravity is playing a role in this step, it can be inferred that a rotation around the direction of gravity has no effect but a rotation perpendicular to gravity has the maximum effect.

In Figure 6-left the advantages of the constraint-based specification becomes apparent: It depicts several possible positions that are allowed by the approach constraint s_1. It is left up to the robot to instantaneously choose one that is currently reachable. This separates the task specification from the robots embodiment. Figure 6-right shows the effect of combining constraints: The grey spatulas are instances of lift-poses, while the red spatulas show poses that are allowed when the 'above oven' constraint is taken away.

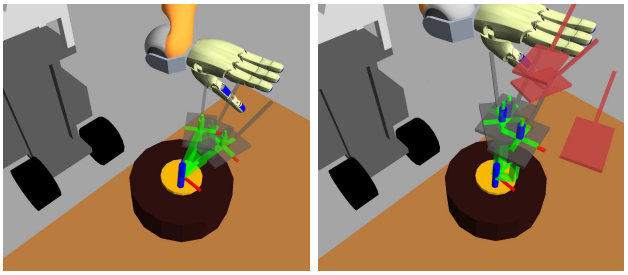


Fig. 6. Left: Allowed instances for the approach pose. Right:Allowed instances for the lift pose (Grey: with 'above oven' constraint, red: without that constraint)

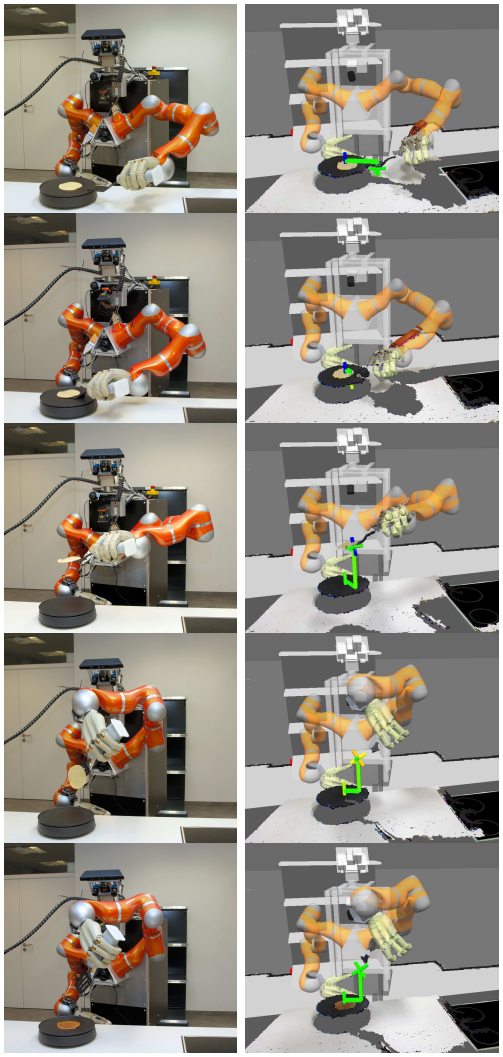


Fig. 7. A pancake flipping of TUM-ROSIE using the proposed constraint-based controller

This specification was run on the robot TUM-ROSIE, which is equipped with two impedance-controlled lightweight arm from KUKA, two DLR-HIT hands from Schunk. These arms are mounted on a Mecanum-wheel omnidirectional base, together with a sensor head containing (amongst others) a high-resolution stereo camera pair and a kinect.

In order to successfully execute this task, an extra constraint had to be introduced, which assures that the spatula remains on one side of the oven. This was required to avoid a singularity of the RPY-angles which surfaced during flipping. We are currently considering more adequate angle representations, which are both adequate for verbal descriptions and free of singularities.

V. Evaluation

Allowing a robot to exploit a task's null space can significantly increase its work space. In order to evaluate this advantage we tested the pancake-baking task in simulation.

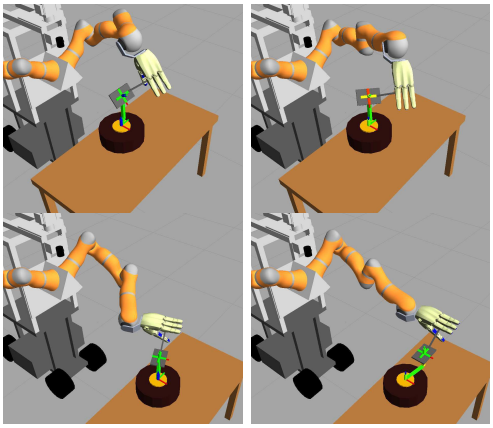


Fig. 8. Snapshots of the workspace evaluation (left: constraint based approach, right: pose based approach)

We generated 30 random poses for the oven in a 0.3 x 0.3 x 0.4 meter space and execute the pancake flipping once with our constraint based controller and once using fixed relative poses between the oven and the spatula which were taken from one trial of the constraint based controller. This was executed by activating all six constraints and setting them to the recorded positions, leaving no room for optimization. Since this scene is rather static, the same performance could have been achieved by planning suitable poses in advance. A randomized planning algorithm which can take into account the same kind of constraints is presented in [15].

The results are shown in the following table:

	approach	under	lift	flip	total
constraint-based	30	30	30	15	115
pose-based	21	23	12	0	56

The constraint based controller can reach more than twice as many poses than the pose based control. Only the flipping pose was posing problems for our approach: Being already a difficult pose to reach, we had to constrain it unnecessarily in order to avoid RPY singularities. A more specialized angle representation may lead to even more successes.

Figure 8 illustrates why the constraint based approach was more successful: When the pose-based controller has to stretch the robot arm in order to reach its sideways approach pose, the constraint based controller exploits the symmetry of the oven and chooses a more dexterous and convenient pose.

VI. Conclusion

We presented a modular way of specifying motions using constraints, which tightly links symbolic reasoning to control theoretic execution. We showed how these constraints can be combined, reasoned about, and monitored during execution. We demonstrated the execution of such constraints and show that they can significantly increase the robots dexterity.

Acknowledgments

This work is supported in part by the EU FP7 Project *RoboHow* (grant number 288533).

References

[1] M. J. Matarić, "Learning in behavior-based multi-robot systems: policies, models, and other agents," *Cognitive Systems Research*, vol. 2, no. 1, pp. 81 – 93, 2001.

[2] G. Metta, G. Sandini, and J. Konczak, "A developmental approach to visually-guided reaching in artificial systems," *Neural Networks*, vol. 12, no. 10, pp. 1413 – 1427, 1999.

[3] O. Khatib, "A unified approach for motion and force control of robot manipulators: The operational space formulation," *Robotics and Automation, IEEE Journal of*, vol. 3, no. 1, pp. 43–53, February 1987.

[4] R. Smits, T. D. Laet, K. Claes, H. Bruyninckx, and J. D. Schutter, "iTASC: A Tool for Multi-Sensor Integration in Robot Manipulation," in *Multisensor Fusion and Integration for Intelligent Systems*, ser. Lecture Notes in Electrical Engineering, H. K. H. Hahn and S. Lee, Eds. Springer, 2009, vol. 35, pp. 235–254.

[5] S. Cambon, F. Gravot, and R. Alami, "A robot task planner that merges symbolic and geometric reasoning." in *Proceedings of the 16th European Conference on Artificial Intelligence (ECAI)*, 2004, pp. 895–899. [Online]. Available: http://www.laas.fr/ scambon/

[6] J. De Schutter, T. De Laet, J. Rutgeerts, W. Decré, R. Smits, E. Aertbeliën, K. Claes, and H. Bruyninckx, "Constraint-based task specification and estimation for sensor-based robot systems in the presence of geometric uncertainty," *Int. J. Rob. Res.*, vol. 26, no. 5, pp. 433–455, 2007.

[7] P. Bonasso, J. Firby, E. Gat, D. Kortenkamp, D. Miller, and M. Slack, "Experiences with an Architecture for Intelligent, Reactive Agents," *Journal of Experimental and Theoretical Artificial Intelligence*, vol. 9, no. 1, 1997.

[8] L. Kunze, M. E. Dolha, and M. Beetz, "Logic Programming with Simulation-based Temporal Projection for Everyday Robot Object Manipulation," in *2011 IEEE/RSJ International Conference on Intelligent Robots and Systems (IROS)*, San Francisco, CA, USA, September, 25–30 2011, best Student Paper Finalist.

[9] D. Pangercic, M. Tenorth, D. Jain, and M. Beetz, "Combining Perception and Knowledge Processing for Everyday Manipulation," in *IEEE/RSJ International Conference on Intelligent Robots and Systems (IROS)*, Taipei, Taiwan, October 18-22 2010, pp. 1065–1071.

[10] I. Kresse, U. Klank, and M. Beetz, "Multimodal autonomous tool analyses and appropriate application," in *11th IEEE-RAS International Conference on Humanoid Robots*, Bled, Slovenia, October, 26–28 2011.

[11] L. Mösenlechner and M. Beetz, "Parameterizing Actions to have the Appropriate Effects," in *IEEE/RSJ International Conference on Intelligent Robots and Systems (IROS)*, San Francisco, CA, USA, September 25–30 2011.

[12] W. Decré, R. Smits, H. Bruyninckx, and J. De Schutter, "Extending itasc to support inequality constraints and non-instantaneous task specification," in *ICRA'09: Proceedings of the 2009 IEEE international conference on Robotics and Automation*. Piscataway, NJ, USA: IEEE Press, 2009, pp. 1875–1882.

[13] E. Marchand, F. Chaumette, and A. Rizzo, "Using the task function approach to avoid robot joint limits and kinematic singularities in visual servoing," in *Intelligent Robots and Systems '96, IROS 96, Proceedings of the 1996 IEEE/RSJ International Conference on*, vol. 3, nov 1996, pp. 1083 –1090 vol.3.

[14] F. Chaumette and E. Marchand, "A redundancy-based iterative approach for avoiding joint limits: application to visual servoing," *Robotics and Automation, IEEE Transactions on*, vol. 17, no. 5, pp. 719 –730, oct. 2001.

[15] D. Berenson, S. Srinivasa, D. Ferguson, and J. Kuffner, "Manipulation planning on constraint manifolds," in *IEEE International Conference on Robotics and Automation (ICRA)*, May 2009.

Constraint-based Movement Representation grounded in Geometric Features

Georg Bartels*
georg.bartels@cs.uni-bremen.de

Ingo Kresse†
kresse@cs.tum.edu

Michael Beetz*
beetz@cs.uni-bremen.de

***Abstract*—Robots that are to master everyday manipulation tasks need both: The ability to reason about actions, objects and action effects, and the ability to perform sophisticated movement control. To bridge the gap between these two worlds, we consider the problem of connecting symbolic action representation with strategies from motion control engineering. We present a system using the task function approach [13] to define a common symbolic movement description language which defines motions as sets of symbolic constraints. We define these constraints using geometric features, like points, lines, and planes, grounding the description in the visual percepts of the robot. Additionally, we propose to assemble task functions by stacking 1-D feature functions, which leads to a modular movement specification. We evaluate and validate our approach on the task of flipping pancakes with a robot, showcasing the robustness and flexibility of the proposed movement representation.**

I. Introduction

Robots which are to master everyday manipulation tasks will have to put screws into nuts, push spatulas under pancakes, cut bread into pieces, and so on. Such robots need both: First, the ability to reason about actions, objects, and the effects of actions on objects and second, the ability to perform sophisticated movement control.

Action formalisms in artificial intelligence are designed to reason about actions and their effects. They do, however, abstract away from the way movements are performed. Indeed, reasoning about actions in the context of more sophisticated movements becomes quickly infeasible as has become evident in the context of the well studied "egg cracking" problem [11]. Abstracting away from *how* actions are performed in terms of movements results in actions having non-deterministic effects and the formalisms being incapable of of explaining how effects depend on the particular form of movement.

Representations in robot learning [1, 12] and control engineering, on the other hand, specify sophisticated movement control but only in terms of coordinate frames [4] or low-level state and control variables [5]. They typically abstract away from objects and action effects. This is unsatisfactory because in manipulation the robot is to perform a movement to have a desired effect on objects.

To get the best of both worlds, we have investigated the coupling of methods for symbolic action representation in Artificial Intelligence with methods in control engineering in previous work [7]. The goal was to provide formalisms that are strong in expressing *how* movements are to be executed while still allowing the robot to reason about the effects that complex movements might have on objects and scenes.

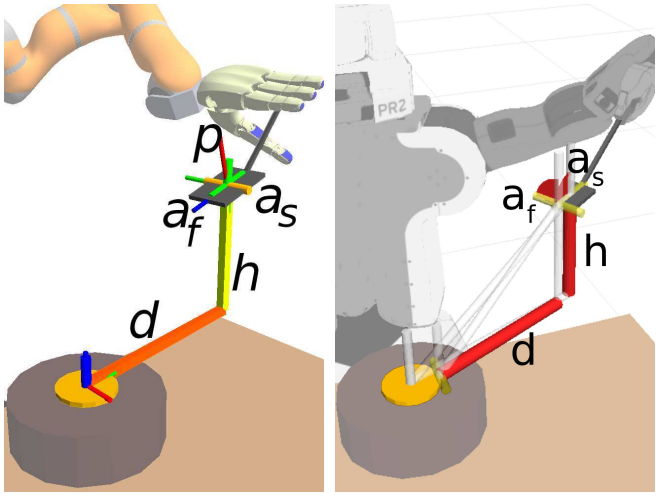


Fig. 1. Two robots making a pancake: (left) related work with restriction to use full frames as control features and a necessity to group constraints in 6-DOF virtual kinematic chains (right) the proposed approach with visually-grounded features using 1-D constraint functions.

We modelled motions as sets of partially ordered movement constraints which can be expressed in terms of objects and their parts. Constraint-based movement descriptions allow the programmer to assert what is essential for the success of the task while keeping other movement aspects free for optimizing the movement and keeping flexibility. Reasoning may then exploit the modular specification by relating action effects to parts of the movement description, e.g. the pancake will be pushed off if angle between the spatula and the oven is too steep.

In this paper, we suggest a novel constraint-based movement description language and its execution controller which outperforms our predecessor system [7] as it

1) enables the robot to firmly anchor the features employed for motion control in the perceptual apparatus of the system to allow feedback-driven execution and high-level error monitoring.
2) avoids stability issues or kinematic singularities in order to generate higher performance movements.
3) imposes less specification restrictions to fascilitate manual and autonomous generation of motion commands.

* Georg Bartels, and Michael Beetz are with the Institute for Artificial Intelligence and the TZI (Center for Computing Technologies), University of Bremen, Germany. † Ingo Kresse is with Technische Universität München, Germany.

To evaluate our system and validate our claims we chose to use the same task as in [7]. We use our low-level system to have a robot push a spatula under a pancake and flip it as depicted in Figure 1. This is a very challenging application both from a low-level and high-level point of view. Additionally, it allows us to directly compare our new system to its predecessor [7].

The remainder of this paper has the following structure: After a brief system overview we present our constraint-based motion representation in great detail. Secondly, we introduce the tools we employ for visualization and numeric analysis of constraints. Afterwards, we evaluate the system using the task of pushing a spatula under a pancake, and compare our movement description language to that of [7]. We present validation of our approach by performing the task of flipping a pancake with a real-world robot. Finally, we will conclude the paper with a summary and an outlook on future work.

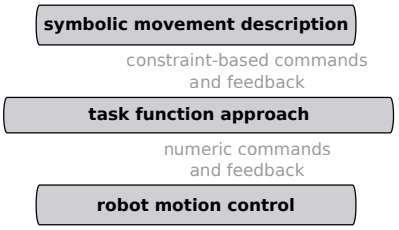


Fig. 2. Conceptual overview of the system: The task function approach maps symbol movement constraints onto numeric control commands and visa versa.

II. System Overview

Consider as an example the pushing of a spatula under a pancake. Informally, this could be described as a movement where (1) *the spatula is always points towards the center of the pancake* (seen from above), (2) *the spatula has touches the oven outside of the pancake*, (3) *after the touching the robot pushes the spatula firmly onto the oven while moving towards the center*, and so on.

We propose to model motions such as pancake flipping as partially ordered sets of movement constraints, i.e. motion phases. We believe that a useful movement description language has to exhibit certain desired properties:

- symbolic and qualitative constraints generalize well over different spatial and kinematic setups
- composable constraints sets allow automatic generation
- motion constraints expressed in terms of perceivable object parts allow tracking-based monitoring
- the motion specification language has to cover the richness of state-of-the-art motion control theory

This last requirement asks for a mapping of the constraint-based movement description language into established control engineering frameworks, such as [2, 6, 4, 9]. We believe this is necessary to generate high quality movements which in turn is necessary to successfully achieve sophisticated manipulations such as pancake flipping. We employ the task function approach to perform this mapping from symbol movement constraints down to numeric feedback control of the robot. Figure 2 depicts a conceptual view of the system.

Finally, we provide a short sample movement description in Figure 3. The example corresponds to the informal describtion of pancake flipping from the beginning of this section. We show sample constraints and one feature definition for the first motion phase, i.e. positioning the spatula front edge over the pancake oven. Each *constraint* relates one *tool feature* on the spatula to one *object-feature* on the oven by using a *feature function*. *Ranges* express the desired goal state of a constraint.

```
features{
  feature{
    name = spatula-front-axis
    ref_frame = /spatula_blade
    type = LINE
    position = [0.0, 0.0, blade_length/2.0]
    direction = [0.0, 0.1, 0.0]}
  ...}
constraints{
  constraint{
    tool_feature = spatula-front-axis
    object_feature = oven-center-point
    function = distance
    range [0.0; oven_radius]}
  constraint{
    tool_feature = spatula-front-axis
    object_feature = oven-plane
    function = height
    range [0.1; +inf]}
  ...}
```

Fig. 3. Sample constraint and feature definitions for the first phase of pancake flipping, positioning the front of the spatula over the oven. The relevant syntactic elements of the language are shown in bold font.

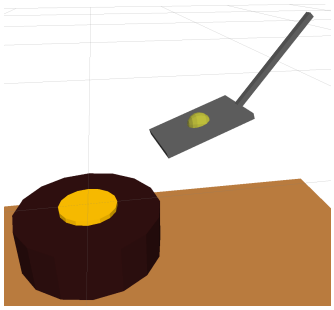
III. Low-level Constraint Representation

Consider directly translating a high level constraint like *"keep the main axis of the spatula pointed at the center of the oven"* into the control law of a robotic manipulator, while also preserving its semantic information. In this section we present the low-level constraint representation and execution we use to achieve this.

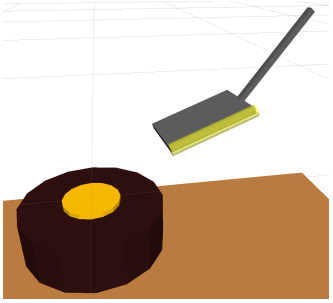
A. Task Function Approach

The control system we propose uses the task function approach [13]. A task function is a differentiable function of the robot's posture. Using it's derivative, the robot is controlled such that the task function yields a desired value. It is also possible to define a task function over different sensor data e.g. the measurement of a force-torque sensor, but the derivative still has to be a function of the robot configuration to enable control.

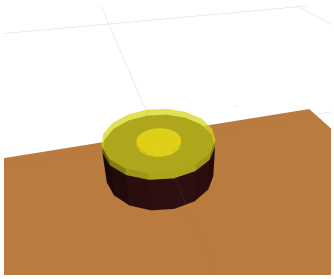
In particular, we take the approach used in iTaSC [15], where the task functions are defined over the pose of an object that is to be manipulated with respect to a tool which is attached to the robots end-effector. This makes it much easier to generalize to different robots or e.g. two-arm setups where another robot arm is holding the object.



(a) A point feature used to model the center of a spatula.



(b) A line-segment feature which represents the side edge of a spatula.



(c) The oven is modeled use a plane-segment feature.

Fig. 4. The three geometric features supported by the system with example usages: The point, line-segment and plane-segment feature.

Consider the poses of the tool $\mathbf{x}_t$ and the object that shall be manipulated $\mathbf{x}_o$. Then, we define the task function that scores the relationship between both objects as $\mathbf{y} = \mathbf{f}_t(\mathbf{x}_t, \mathbf{x}_o)$. Differentiating the task function with respect to time yields:

$$\dot{\mathbf{y}} = \frac{\partial \mathbf{f}_t}{\partial \mathbf{x}_t}\dot{\mathbf{x}}_t + \frac{\partial \mathbf{f}_t}{\partial \mathbf{x}_o}\dot{\mathbf{x}}_o. \tag{1}$$

Assuming that we can only control the tool and not the object, we simplify (1) to

$$\dot{\mathbf{y}} = \frac{\partial \mathbf{f}_t}{\partial \mathbf{x}_t}\dot{\mathbf{x}}_t = \mathbf{H}\mathbf{t}_t, \tag{2}$$

where $\mathbf{H}$ is called the interaction matrix and $\mathbf{t}_t$ obviously denotes the twist of the tool. As we assume the tool to be rigidly attached to the gripper of the robot we use the robot Jacobian $\mathbf{J}_R$ to derive the control law which relates joint velocities $\dot{\mathbf{q}}$ to the derivative of the feature function:

$$\dot{\mathbf{y}} = \mathbf{H}\mathbf{J}_R\dot{\mathbf{q}}. \tag{3}$$

Calculating the weighted pseudo-inverse of $\mathbf{H}\mathbf{J}_R$ we can derive the necessary joint velocities to obtain the desired changes in the task function values. Please note, that we also assumed a good calibration of the tool within the gripper. In previous work we demonstrated how this can be done automatically using visual features of tools like lines, holes or concavities. [8]. Note, it is necessary to transform the reference points and -frames of both $\mathbf{H}$ and $\mathbf{J}_R$ before combining them and pseudo-inverting the result [3].

In this paper, we propose to construct the task function $\mathbf{f}_t$ by stacking a set of n scalar-valued *feature functions* $f_{fi}(\mathbf{x}_t, \mathbf{x}_o)$:

$$\mathbf{f}_t(\mathbf{x}_t, \mathbf{x}_o) = \begin{bmatrix} f_{f1}(\mathbf{x}_t, \mathbf{x}_o) \\ \vdots \\ f_{fn}(\mathbf{x}_t, \mathbf{x}_o) \end{bmatrix}. \tag{4}$$

Each of the feature functions $f_{fi}(\ldots)$ expresses to which extent a specific spatial relationship between the features of tool and of the object, such as perpendicularity or distance, holds. Thus, they directly map constraints like *"keep the main axis of the spatula pointed at the center of the oven"* into the control law for the robot.

B. Geometric Features

The basic building blocks of the motion specification framework we present are its *geometric features*. They represent remarkable parts of the tool and object that shall be related to one another. Currently, we have the following geometric features implemented in our system: A point feature, which might represent a small button of an oven, a line feature, e.g. denoting the edge of a tool or a pointing device such as a finger, and a plane feature, that can be used to model planar support surfaces. All of the features have the following properties:

- a reference frame, w.r.t which they are expressed
- an origin vector, denoting the Cartesian position of the feature
- a direction vector, denoting the main orientation of the feature (note that for points this is ignored and for planes this corresponds to the normal of the plane)

Figure 4 shows several objects with sample features visualized that are used to model key parts of the interacting objects.

C. Feature Functions

Using the geometric features we define feature functions f_{fi} that map two features onto a scalar value. As already pointed out, feature functions need to be differentiable and depend on the pose of the robot because the tool feature is attached to the end-effector. Also recall that we build the interaction matrix $\mathbf{H}$ by assembling the partial derivatives of the feature functions w.r.t tool motions $\partial\mathbf{x}_t$. The presented system contains – among others – the following feature functions which we use for evaluation:

- *perpendicular*: Equals zero if both feature directions are perpendicular to one another.
- *height*: corresponds to the length of vector between the origins of the two features, projected onto the direction of the first feature.
- *distance*: denotes the length of the vector between the origins of the two features, projected onto the perpendicular of the direction of the first feature.
- *pointing_at*: equals zero if the direction of the first feature is pointing at the second one, i.e. is directed at some point

on the line defined by the origin and direction vectors of the second feature.

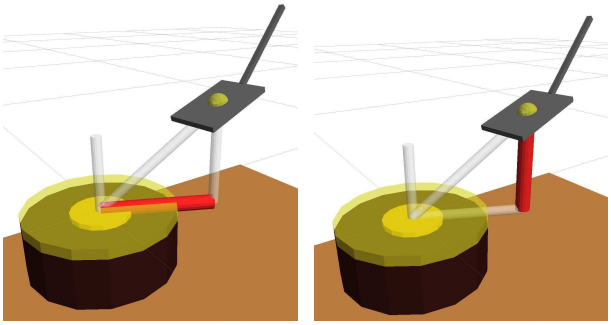


Fig. 5. Visualization of the qualitative meaning of the projected distance (left) and height (right) feature functions between the center of a plane segment and a point.

Using feature functions allows us to express spatial relationships between geometric features as numerical values. For example, we can express that a line in a given situation is pointing more at the center of a plane than in a second setup. Figure 5 shows exemplary visualizations of the height and projected distance feature functions evaluated on a point and plane-segment feature, respectively. Please note, to evaluate the feature functions we assume to get the homogeneous transformation between the reference frames of the features from perception.

D. Constraints

Finally, we define *constraints* as sets that contain a pair of geometric features, one feature function to describe the intended relationship and a desired value y_d for the feature function. For example, to have a line feature $line_{axis}$ – corresponding to the main axis of our spatula – pointing at the plane $plane_{oven}$ which represents the oven, we just need to specify $y_1 = f_{f1}(\ldots) = pointing\_at(line_{axis}, plane_{oven})$ and $y_d = 0$.

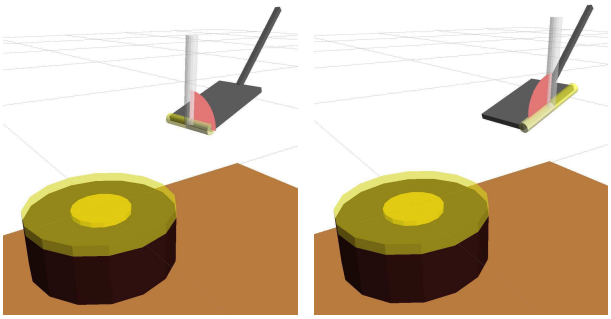


Fig. 6. Aligning a spatula with an oven using two alignment constraints: Restricting the front line-segment of the spatula to be very close to perpendicular to the oven plane's normal causes a good contact (left), while the perpendicular constraint between the side edge and the oven regulates the angle between spatula and oven (right).

A further example is given in Figure 6. Using a set of three constraints we determine the relative orientation of the spatula with respect to the oven. First of all, the front line of the spatula shall be perpendicular to the oven plane's normal, i.e. desired value zero. This constraint causes the front edge of the spatula to form a nice contact with the surface of the oven. Additionally, we set up a perpendicular constraint between the side edge of the spatula and the oven plane which has a non-zero desired value. This value determines the outcome of the pushing action, and learning from demonstration could be a means obtaining it. Finally, we added a constraint requiring the main axis of the spatula to point to the center of the plane segment representing the oven (not shown in Figure 6). This example highlights how a set of constraints expresses complex spatial relationships in a modular and feature-based way.

E. Controllers for Range-Based Constraints

To incorporate more flexibility into the constraint definitions, we allow for desired ranges $[y_{lo}, y_{hi}]$ instead of just a single desired value y_d for each constraint output. This is useful because symbolic constraints often translate to inequality expressions, e.g. "hold the spatula *over* the pancake". Given the current output values and the desired ranges, the controller calculates the desired velocity $\dot{y}_{des}$ of each constraint. These are then used to solve equation 3 for the desired robot joint velocities $\dot{q}_{des}$ which are sent to the robot hardware.

Furthermore, each 1-D constraint function controller also controls its corresponding weighting factor w influencing the calculation of the weighted pseudo-inverse of $\mathbf{HJ}_R$. Whenever the output value of a constraint function is within the desired range, the controller lowers the weight of that constraint. Reducing the weight of a constraint effectively extends its instantaneous null space, i.e. gives more freedom to other conflicting and unfulfilled constraints.

Since this part of the system is almost identical in design to our previous system we refer the reader to [7] and our code repository for further details.

IV. Analysis of Constraint Sets

A. Visualization of Constraint Sets

Given a set of constraints such as *have a tilted angle between the main spatula axis and the oven plane* and *have no distance between the front edge of the spatula and the oven plane*, we want to visualize the tool motion each of them causes. By visually observing alignment properties or locations of movement axes users may better judge the correctness of constraint sets *during* motion execution.

We present a method to visualize the effect of any task function defined over Cartesian transformations. The input for our method is the interaction matrix $\mathbf{H}$ and the poses (positions $\mathbf{p}$) set in relation by the task function. Therefore our method applies to any task function defined over Cartesian transformations, e.g. the type of task functions we present or virtual kinematic chains. It even generalizes to robot Jacobians.

As shown in Figure 7, we compute and display the instantaneous movements (twists) that affect one and only one constraint at a time. Rotational constraints are displayed by their axis of rotation, translational constraints by their direction.

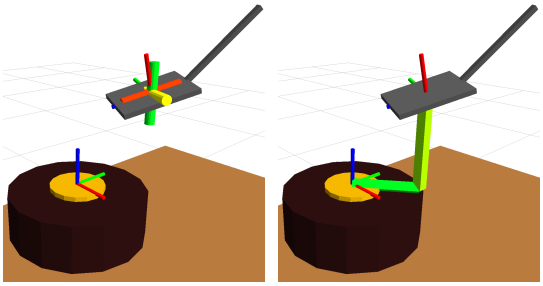


Fig. 7. Visualization of rotational twists (left) and translational twists (right)

1) Computing Independent Twists: Given the interaction matrix $\mathbf{H}$, we first compute instantaneous movements (twists) that affect one and only one constraint at a time. By this definition, we call them *orthogonal*.

These twists are more useful for display purposes. In particular, our angular alignment constraints do not depend on a position and thus the location of their instantaneous rotation axis is not defined. However, by demanding orthogonality with other constraints, we can infer the location of the axis as a place where rotations around it do not affect other constraints.

Consider the task interaction matrix from equation 2: It computes how a particular twist affects the constraints. Here, we seek the inverse: Given a particular $\dot{\mathbf{y}}$ (one, where only one constraint is non-null), we want to compute the twist that affects this (and only this) constraint. This is computed by the inverse of the interaction matrix $\mathbf{H}^+$.

If the twists in the rows of $\mathbf{H}$ are *linearly independent*, the inverse matrix $\mathbf{H}^+$ yields columns of mutually *orthogonal* twists where each twist affects only it's respective constraint.

2) Visualization of Twists: For visualizing twists, we draw some ideas from screw theory [10] and Plücker line representations [14]. A twist can be interpreted as a rotation around an axis and a simultaneous translation along the same axis (screw movement). For rotational joints, this axis can be visualized: The segment of the line that is closest to the origin is displayed using a cylinder.

For pure translational movements, the location of this axis is not defined, only its orientation. This type of movement can be visualized as a line, showing the direction. However, it still must be decided where to place this line and how long it should be.

When displaying twists, three possibly different coordinate systems are of interest:

- The reference frame, in which directions are expressed
- The reference point, around which pure rotations happen
- The target frame, to decide the line piece to show

The reference frame and -point must be known in order to interpret a twist. For visualization, however, we change the reference frame of the twist to the target frame. We refer the reader to [3] for a describtion of how to achieve this.

For the visualization of the instantaneous rotation axis, we exploit the fact that a twist can be re-interpreted as a Plücker line $\mathbf{t} = (\mathbf{q}, \mathbf{q}_0)$ where $\mathbf{q}$ is the rotational part and $\mathbf{q}_0$ is the translational part of the twist. The direction of the line is simply its directional part $\mathbf{q}$ and the position is the closest point of the line to the origin and is computed as:

$$\mathbf{p} = \frac{\mathbf{q} \times \mathbf{q}_0}{\|\mathbf{q}\|^2}$$

A short line segment around this point is displayed to visualize the twist.

If the twist is (nearly) a pure translation, then $\mathbf{q}_0$ is close to 0 and the axis would be placed (nearly) at infinity. For these constraints, only the direction vectors $(\mathbf{v}_1, ..., \mathbf{v}_n)$ are given, locations and lengths need to be inferred. This step takes into account the translation $\mathbf{p}$ between tool and object: The translations are required to form a path from the object origin to the tool origin, i.e. they must sum up to $\mathbf{p}$:

$$\mathbf{p} = \sum_{i=n}^{n} \alpha_i \mathbf{v}_i \tag{5}$$

In matrix form this becomes:

$$\mathbf{V}\alpha = \mathbf{p} \tag{6}$$

with $\mathbf{V} = [\mathbf{v}_1 \cdots \mathbf{v}_n]$. Solving this linear equation for α yields the lengths of the visualization vectors. Their locations are determined by concatenating these vector together in the order as they appear in the inverse interaction matrix $\mathbf{H}^+$ (see Figure 7, right).

We also move the target frame of the rotational constraints to the end point of the last translational constraint line, in order to maintain visual coherence and to avoid axis segments far away from other markers.

This visualization tool is well suited for virtual kinematic chains and other task functions that fully cover the 6 degrees of freedom between object and tool. We have used it during system development, design of the pancake flipping application, and to generate the simulation figures in this paper.

B. Combination / Comparison of Constraints

Imagine a situation in which a user of our system has assigned alignment constraints to both the left and right side of the spatula. Are there ways to detect that they depend on each other? Similarly, consider the alignments of the spatula's front edge, its side edge and its blade plane (all w.r.t. the oven). How to detect that only two of them are controllable at the same time? Both situations are displayed in Figure 8.

This kind of reasoning is crucial when autonomously combining constraints in the suggested modular fashion. Ideally, we would like the system to only accept constraint sets in which all constraints are independently controllable. Such a situation is given when the rows of the interaction matrix $\mathbf{H}$ are linearly independent. In this case it is possible to find an inverse matrix $\mathbf{H}^+$ in which each column contains a twist that affects one and only one constraint. In order to rule out local singularities – which might be avoided by the desired ranges of the constraints – we check the rank of $\mathbf{H}$ for many transformations and take the maximum.

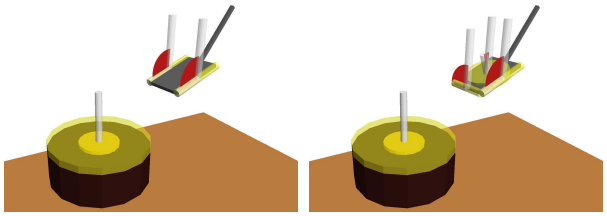


Fig. 8. Dependent alignment constraints (left image: left and right spatula edges, right image: left and front spatula edges and blade)

Another interesting question that can be answered with this tool is whether two sets of constraints are equivalent from a control point of view, i.e. if they are controlling the same degrees of freedom. This is true for any task function if

$$\text{rank}(\mathbf{H}_1) = \text{rank}(\mathbf{H}_2) = \text{rank}(\begin{bmatrix} \mathbf{H}_1 \\ \mathbf{H}_2 \end{bmatrix})$$

V. Evaluation & Validation

For evaluation we want to compare our abstract low-level movement control machine to its predecessor [7] because it is the most related approach in the literature. The evaluation considers three aspects of the proposed system: Singularities of the underlying representation, execution discontinuities, and the expressiveness of the movement descriptive language. As evaluation task we choose flipping of a pancake with a spatula.

To perform pancake flipping with a robot we spanned a 6-DOF VKC between the oven and the spatula (see Fig.1), and controlled its virtual joints over time in [7]. For the first three joints we used cylinder coordinates (angle, distance, and height, a, d, h) which was motivated by the rotational symmetry of the oven. For orienting the tool we chose RPY angles which roughly correspond to the constraints of aligning the front edge (a_f), aligning the side edge (a_s) and pointing direction (p).

We essentially re-create the same chain using feature-based constraints within our framework. As a result, we are able to compare both approaches while performing the same task.

A. Representation Singularities and Discontinuities

To detect a singularity, we can compute the rank of the interaction matrix $\mathbf{H}$ as described in section IV-B. Singularities represent poses in which two constraints become dependent. However, for a controller it is equally important to avoid discontinuities. To detect these, we compute the second derivatives using three equidistant samples (x_1, x_2, x_3) of the task function $\mathbf{f_t}$ around the current pose:

$$\mathbf{f_t}'' \approx \frac{\mathbf{f_t}(x_3) - 2\mathbf{f_t}(x_2) + \mathbf{f_t}(x_1)}{h^2} \quad (7)$$

Where h is the distance between the samples. If the value of $\mathbf{f_t}''$ exceeds a threshold, we assume to have found a discontinuity.

Using this investigation methodology, we compare the virtual linkage constraints with our feature-based constraints near a singularity. The spatula is moved over the center of the oven. At this place, two angles are undefined: 1) the direction where the spatula is coming from – it is already there – and 2) the direction in which the spatula is pointing (relative to the center) – it is always 'away' from the center. However, the alignment constraints for the front-edge and the side-edge of the spatula are semantically independent. Figure 10 visualizes the described tool pose. Near this singularity the following constraints become uncontrollable because of discontinuities:

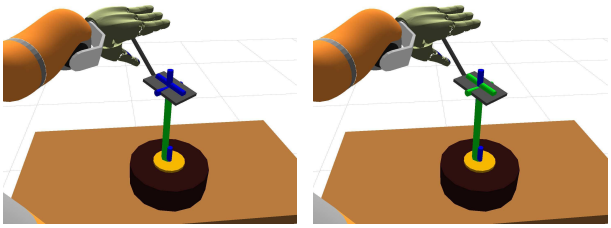


Fig. 10. Constraints at singular positions (left: RPY, right: feature-based). The green rotation/translation axes are fine, the blue axes have a discontinuity and are thus unusable for control

approach	angle	dist	height	align-front	align-side	point-at
virtual-linkage	X	.	.	X	X	X
feature-based	X	.	.	.	.	X

For the virtual-linkage solution, the angles depend on the cylinder coordinates: At the singularity at dist=0, all three angles become discontinuous. For the feature-based constraints, the align-front and align-side constraints are completely independent of the position and thus remain well-defined and controllable. This demonstrates that our approach avoids several discontinuities that otherwise require heuristics, e.g. always keeping a minimal distance from the oven center.

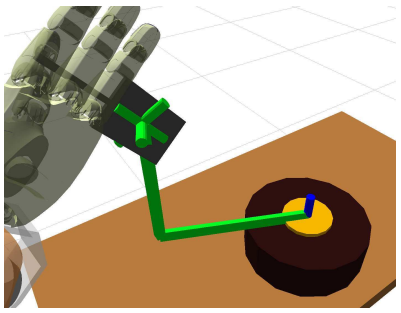


Fig. 11. A non-trivial orientation where RPY-angles loose their meaning.

Some discontinuities remain, inherent in the problem specification. However, the system can detect them, allowing automatic addition of extra constraints to avoid singularities.

Regarding the angle representations, consider the example given in Figure 11. The spatula is rotated such that both the front and the side edge are at roughly 45 degrees to the oven plane (a_f and a_s), pointing about 20 degrees left to the oven center (a_p).

Our feature-based constraints correctly report these values as (a_f=-39^o, a_s=50^o, p=18^o). The RPY angle representation looses this meaning: It reports (a_f=-17^o, a_s=-18^o, p=-73^o).

```
start_frame = pancake
end_frame = spatula
VKC = {rot-z,trans-x,trans-z,rot-x,rot-y,rot-z}
constraint1{
  semantics = ''approach-angle''
  range <unconstrained>}
constraint2{
  semantics = ''distance''
  range = [...]}
constraint3{
  semantics = ''height''
  range = [...]}

  ...

constraint6{
  semantics = ''align-front''
  range = [...]}
```

```
constraint{
  tool_feature = spatula-blade-plane
  object_feature = pancake-plane
  function = perpendicular
  range [...]}
constraint{
  tool_feature = spatula-main-axis
  object_feature = pancake-center-point
  function = pointing-at
  range [...]}

  ...

constraint{
  tool_feature = spatula-front-axis
  object_feature = pancake-rim-line
  function = distance
  range [...]}
```

(left) Movement description language in the predecessor controller [7]: Users need to define the entire fixed-sized 6-DOF VKC. It is noteworthy that constraints which are unconstrained during a motion still need to be modelled, e.g. constraint1. Additionally, the semantics of each constraint strongly depend upon context, and are provided as inputs. For example, constraint1 and constraint6, are both rotation around z. Both, however, carry different semantic meaning.
(right) Our newly suggested description: The amount of constraints is unrestricted. Each constraint is defined over its own set of features. Thus, it is possible to have more than two features of interest in a motion description. In our earlier description language this required an additional VKC for every new pair of frames. Finally, the feature functions themselves carry their own semantics. This removes the need for the high-level to connect semantics and function.

Fig. 9. Comparing two movement descriptions: On the left, the description as presented in [7] and on the right the newly suggested description.

While deviations in a_f and a_s might be tolerable in certain situations, the pointing direction has lost meaning completely. At the $(0, 0, 0)$-position, this meaning was still valid.

B. Expressiveness and Usefulness of the Movement Description

To evaluate the expressive differences between our proposed and its predecessor benchmark description language we display parts of a corresponding motion description in Figure 9. Both examples describe a sub-motion from a robotic pancake baking task.

When comparing both description, it becomes apparent that our improved language allows for more flexible movement specifications because constraints are allowed in any number and can be ordered arbitrarily. These properties make our representation more modular. Regarding [7], constraints need to be used in groups of six, all part of 6-DOF VKCs – a considerable specification restriction.

In our predecessor system the frames and the virtual joints in the VKCs had to be chosen with great care to ensure chain joints with semantic meaning. It is obviously difficult to automate this reasoning which is a prerequisite for an autonomous high-level system that uses our earlier abstract low-level movement control system. Our newly proposed system, however, encodes semantics in carefully designed feature functions relating object features. As these features are grounded in perception, a possible high-level system no longer needs to decide where to put control frames. Equally, human designers may also find the application programming task much easy.

C. Validation Experiments

We validated our system by performing the task of pancake flipping on a real-world robot. The robot employs both of its arms to complete the assignment, with the movements modelled and executed using the proposed approach. A video recording with detailed setup descriptions can be found in the supplemental material to this paper. Additionally, we modelled the task of pouring a liquid from a bottle (see Figure 12), and are assessing our movement describtion language on further tasks.

VI. Conclusion

We presented an abstract low-level movement control system. The system is able to perform motions that are specified in terms of sets of constraints over object parts relations, e.g. *move the spatula plane next to the oven plane*, *keep both planes parallel*, while *the main axis of the spatula is pointing at the rim of the pancake*. This interface provides a semantically higher and richer interface, enabling a symbolic high-level system to reason about the effects of actions in terms of the movement parameters that caused them.

Using geometric features, such as points, lines or planes, as control features is a key property of the presented description

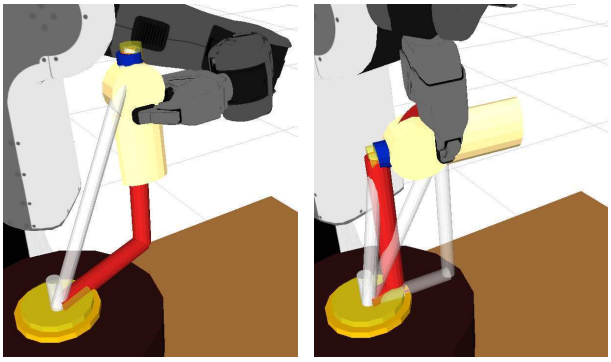


Fig. 12. Visualization of the PR2 pouring content from a bottle onto the pancake oven. The task was modelled with four constraints employing each a different feature function (height, distance, perpendicular, pointing-at). Only three features were necessary to complete the specification: bottle axis, bottle cap, and oven center plane.

language. It thus directly grounds the movement specification into the perceptual apparatus of the robot. Additionally, we do not employ an underlying modelling structure like virtual kinematic chains, which greatly improves the versatility and modularity of our system.

We evaluated and validated the system in a robotic pancake making experiment, and compared performance with its predecessor reference system [7]. Simulated and real-world experiments show the applicability of our approach. Additionally, we highlighted the improved modularity and better grounding of our motion representation. In terms of angular singularities and control discontinuities we showed that our proposed system avoids issues that the system presented in [7] encountered. Furthermore, we presented our approach to analyzing interaction matrices which greatly improves debugging of this type of system.

In the future, we will connect the abstract low-level movement control system to a reasoning high-level system which can exploit the expressiveness of the presented movement description. Consequently, drawing nearer to the goal of equipping robots with the manipulation competence humans exhibit everyday.

Acknowledgments

This work is supported in part by the EU FP7 Projects *RoboHow* (grant number 288533) and *SAPHARI* (grant number 287513). We thank Joris DeSchutter's and Herman Bruyninckx' group from KU Leuven for insightful discussions.

References

[1] Pieter Abbeel, Adam Coates, and Andrew Y. Ng. Autonomous helicopter aerobatics through apprenticeship learning. *The International Journal of Robotics Research*, 29(13):1608–1639, 2010.

[2] Albu-Schaffer, A., Haddadin, S., Ott, Ch, Stemmer, A., Wimbock, T., Hirzinger, and G. The DLR Lightweight Robot – Design and Control Concepts for Robots in Human Environments. *Industrial Robot: An International Journal*, 34(5):376–385, 2007. ISSN 0143-991X.

[3] Tinne De Laet, Steven Bellens, Ruben Smits, Erwin Aertbeliën, Herman Bruyninckx, and Joris De Schutter. Geometric relations between rigid bodies: Semantics for standardization. *IEEE Robotics and Automation Magazine*, 2012.

[4] Joris De Schutter, Tinne De Laet, Johan Rutgeerts, Wilm Decré, Ruben Smits, Erwin Aertbeliën, Kasper Claes, and Herman Bruyninckx. Constraint-based task specification and estimation for sensor-based robot systems in the presence of geometric uncertainty. *Int. J. Rob. Res.*, 26(5):433–455, 2007. ISSN 0278-3649.

[5] S.M. Khansari-Zadeh and A. Billard. Learning stable nonlinear dynamical systems with gaussian mixture models. *Robotics, IEEE Transactions on*, 27(5):943 –957, oct. 2011. ISSN 1552-3098.

[6] O. Khatib. A unified approach for motion and force control of robot manipulators: The operational space formulation. *Robotics and Automation, IEEE Journal of*, 3(1):43–53, February 1987. ISSN 0882-4967.

[7] Ingo Kresse and Michael Beetz. Movement-aware action control – integrating symbolic and control-theoretic action execution. In *IEEE International Conference on Robotics and Automation (ICRA)*, St. Paul, MN, USA, May 14–18 2012.

[8] Ingo Kresse, Ulrich Klank, and Michael Beetz. Multimodal autonomous tool analyses and appropriate application. In *11th IEEE-RAS International Conference on Humanoid Robots*, Bled, Slovenia, October, 26–28 2011.

[9] Nicolas Mansard, Oussama Khatib, and Abderrahmane Kheddar. A unified approach to integrate unilateral constraints in the stack of tasks. *IEEE Transactions on Robotics*, 25:670–685, 2009.

[10] Matthew T. Mason. *Mechanics of Robotic Manipulation*. MIT Press, 2001. ISBN 0262133962.

[11] L. Morgenstern. Mid-Sized Axiomatizations of Commonsense Problems: A Case Study in Egg Cracking. *Studia Logica*, 67(3):333–384, 2001.

[12] P. Pastor, M. Kalakrishnan, S. Chitta, E. Theodorou, and S. Schaal. Skill learning and task outcome prediction for manipulation. In *Robotics and Automation (ICRA), 2011 IEEE International Conference on*, pages 3828–3834, may 2011.

[13] C. Samson, M. Le Borgne, and B. Espiau. *Robot Control, the Task Function Approach*. Clarendon Press, Oxford, England, 1991.

[14] Ken Shoemake. Plücker coordinate tutorial. *Ray Tracing News*, 11(1), 1997.

[15] Ruben Smits, Tinne De Laet, Kasper Claes, Herman Bruyninckx, and Joris De Schutter. iTASC: A Tool for Multi-Sensor Integration in Robot Manipulation. In Hanseok Ko Hernsoo Hahn and Sukhan Lee, editors, *Multisensor Fusion and Integration for Intelligent Systems*, volume 35 of *Lecture Notes in Electrical Engineering*, pages 235–254. Springer, 2009.

Bibliography

[Bartels et al., 2013] Bartels, G., Kresse, I., and Beetz, M. (2013). Constraint-based movement representation grounded in geometric features. Submitted for review to Humanoids 2013.

[Beetz et al., 2010] Beetz, M., Mösenlechner, L., and Tenorth, M. (2010). CRAM – A Cognitive Robot Abstract Machine for Everyday Manipulation in Human Environments. In *IEEE/RSJ International Conference on Intelligent Robots and Systems*, pages 1012–1017, Taipei, Taiwan.

[Kresse and Beetz, 2012] Kresse, I. and Beetz, M. (2012). Movement-aware action control – integrating symbolic and control-theoretic action execution. In *IEEE International Conference on Robotics and Automation (ICRA)*, St. Paul, MN, USA.

[Winkler et al., 2012] Winkler, J., Bartels, G., Mösenlechner, L., and Beetz, M. (2012). Knowledge enabled high-level task abstraction and execution. *First Annual Conference for Advances in Cognitive Systems*, 2(1):131–148.