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.

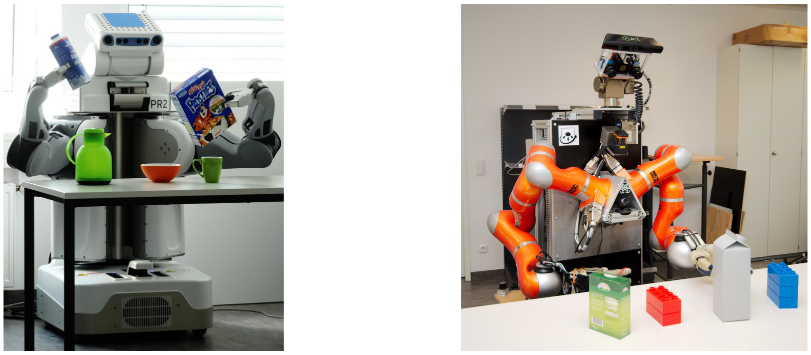

Figure 1. Different objects mostly require different grasping techniques and may not obstruct each other when placing them.

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.

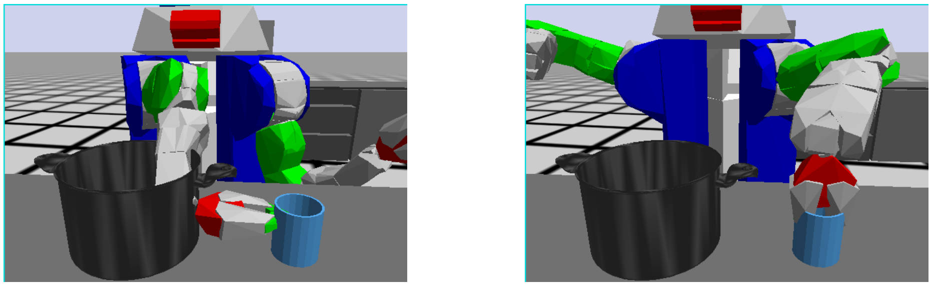

Figure 2. When grasping a cup that is standing left of a pot, it is better to use the left arm.

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.

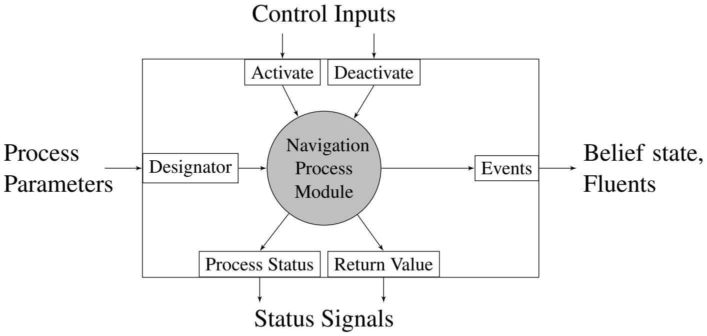

Figure 3. Process module encapsulating a navigation control process. The input is a location designator, symbolically describing the goal pose and the context this code is executed in (e.g. *(location (to see) (obj cup)))*. The feedback events provide information about the status of the current navigation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

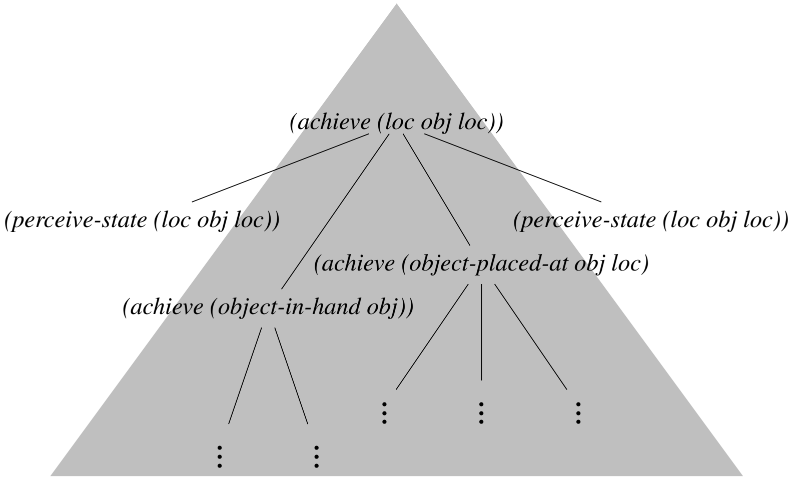
(achieve (loc obj loc))
(perceive-state (loc obj loc))
(perceive-state (loc obj loc))
(achieve (object-placed-at obj loc)
(achieve (object-in-hand obj))

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.

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

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))))))
```

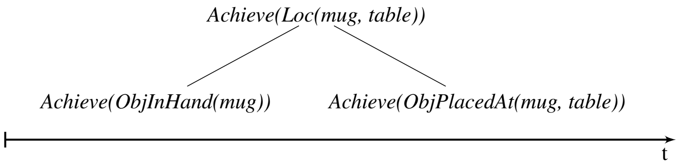

Figure 5. Excerpt from an execution trace in context of the pick and place scenario.

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.

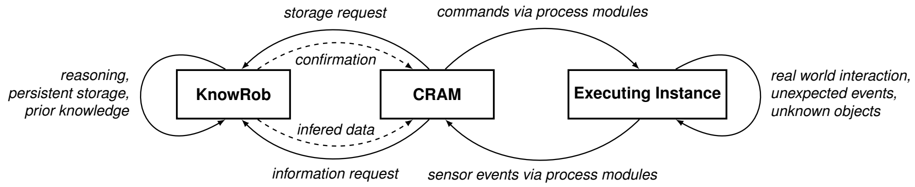

Figure 6. Flow of actions and information between CRAM, KnowRob and real or simulated robots. CRAM emits the control flow and queries both, KnowRob and the executing instance.

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.