Designing and Implementing a Plan Library for a Simulated Household Robot

Armin Müller and **Michael Beetz**
Intelligent Autonomous Systems Group
Department of Informatics
Technische Universität München
Boltzmannstr. 3, D-85748 Garching
{muellear,beetz}@cs.tum.edu

Abstract

As we are deploying planning mechanisms in real-world applications, such as the control of autonomous robots, it becomes apparent that the performance of plan-based controllers critically depends on the design and implementation of plan libraries. Despite its importance the investigation of designs of plan libraries and plans has been largely ignored. In this paper we describe parts of a plan library that we are currently developing and applying to the control of a simulated household robot. The salient features of our plans are that they are designed for reliable, flexible, and optimized execution, and are grounded into sensor data and action routines. We provide empirical evidence that design criteria that we are proposing have considerable impact on the performance level of robots.

Introduction

If we look at the fields of planning and plan-based control the issue of plan representation and plan and action design has been largely neglected for some time. In the last three ICAPS conferences none of the conference sessions and only one paper was explicitly addressing this issue. It seems that plan representation and design is considered either to be irrelevant or solved. Indeed, for the planning competitions the formats of valid plans are precisely specified.

Unfortunately, as we are increasingly investigating issues in planning and plan-based control in the real world, in particular in the context of autonomous robot control, it becomes evident that our plan languages abstract away from so much important information that existing planning mechanisms are only of little use for autonomous robot control.

It is clear why. Roboticists aiming at high performance robot behavior must address optimal parameterizations of control routines and the reliable execution of tasks. Consider an office delivery robot navigating through its environment. If we look at the state of the art, in particular when the robots use sonar sensors, then the robots are capable of navigating through the hallway quickly, but have problems with traversing doorways. Consequently, looking ahead and preparing for smooth and optimal doorway traversals has more impact on performance than tour optimization. However, these improvements cannot be achieved when considering door traversal as a blackbox independent of its parameterization and situational context.

Making plans tolerant of sensor error, execution failures, and changing environments requires them to specify how to respond to asynchronously arriving sensory data and other events. They must also prescribe concurrent and synchronized actions. Many control patterns others than those provided by common plan representations have been proven to be necessary for flexible and reliable robot control. Plans cannot abstract away from the fact that they generate concurrent, event-driven control processes without the robot losing the capability to predict and forestall many kinds of plan execution failures.

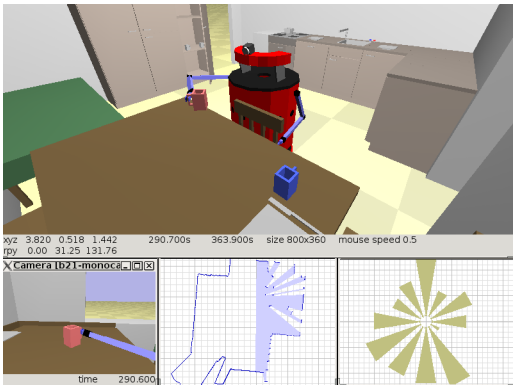

Figure 1: Simulated kitchen and household robot with sensory data (camera, laser, and sonar).

If we agree with these arguments, then plan representation and the design of plan libraries are the nuts and bolts of successful plan-based control of robotic agents. In this paper we report on our experiences with a preliminary design of a plan library for a simulated household robot that is depicted in figure 1. We will discuss specific control patterns that allow for failure monitoring and recovery, partly specified parameters, and annotations.

In the remainder of this paper we proceed as follows. The next section sketches the robot, its environment, and its tasks. The low-level platform for programming plans is described thereafter. We will then discuss design issues for plan libraries and discuss some implemented plans. We conclude with some experimental results, a discussion of related work, and a concluding discussion.

Environment and Robot Architecture

Before we can discuss plans and control mechanisms of an autonomous robot we must first have an understanding of the (simulated) world the robot is operating in. In this paper we use a household robot simulator for a kitchen environment based on the Gazebo simulator of the Player/Stage project. Our simulator is designed carefully, along those dimensions we are interested in, so that it challenges robot controllers in the same way as real environments do.

The Player/Stage project develops Free Software tools for robot and sensor applications. With Gazebo it is possible to simulate robots, sensors and objects in a three-dimensional environment. The pyhsical interaction between objects and the realistic sensor feedback is based on ODE, an accurate library for simulating rigid-body physics. Our robot controller does not communicate directly with Gazobo. For this we use Player, which provides us with a network interface and an abstraction from the used simulator/hardware. Using this abstraction we ensure that our controller can be used in an equivalent real environment supported by Player.

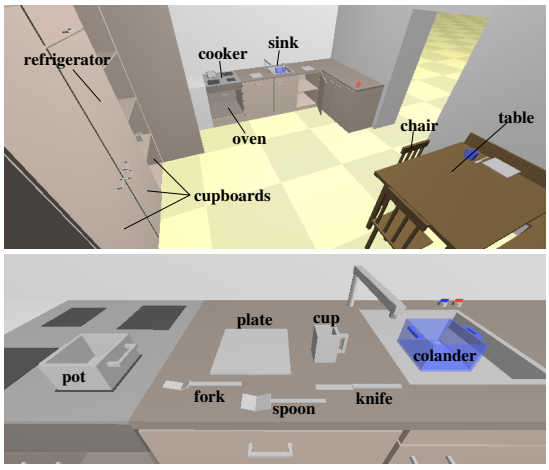

Figure 2: Kitchen Environment

The environment (depicted in figure 2) that the robot is to operate in is a kitchen (SIMUKITCHEN) that consists of the following pieces of furniture and appliances: a cupboard, an oven, a refrigerator, a sink, a cooking desk, and a table with chairs. The furniture pieces and appliances have doors, handle bars, knobs, faucets, and buttons that can be operated by the robot. The positions of the pieces of furniture are static and known, with those of the chairs being exceptions.

In addition, the environment contains flatware (such as knives, forks, and spoons), cookware (pots and pans), and dinnerware (including plates, cups, and bowls). These objects can be recognized and manipulated by the robot. Objects have properties, some of them are perceiveable. Usually these properties do not suffice to infer the object identity.

SIMUKITCHEN also simulates various physical processes and detailed manipulations at a very abstract level, but with realistic non-deterministic effects. For example, we have a process boiling the water that specifies the heat transfer from the oven to the liquid in the pot depending on the kind of liquid (water or milk) and the oven temperature. In particular this process non-deterministically generates overboiling events and spills the liquid onto the heating plates. Another example is to simulate how slippery a pot might get when filled with water and how likely the pot will slip out of the robot's gripper depending on the slipperiness and weight of the pot and the force applied through the grip.

Since exogenous events may be generated randomly, the robot can have at best probabilistic models of them. From the robot point of view the world is nondeterministic. Besides making planning more difficult, exogenous events require the robot to react to unplanned events.

Commands and Jobs

The properties of the commands and jobs given to the robot also determine desiderata for the robot controllers and plans to be used. Our delivery robot is taskable: it is supposed to carry out multiple jobs, which can be changed by adding, deleting, or modifying jobs at any time. The jobs are instances or variations of more general jobs that the robot has done often before. The robot has routines for carrying out the general jobs that can handle many but not all variations of possible job instances.

A typical set of jobs for a day includes laying the table, cooking, serving lunch, and keeping the kitchen clean and tidy. The robot's individual jobs are diverse: achieving a state of affairs, like having lunch served, maintaining a state, like keeping the kitchen clean, or getting information. More specifically, the robot can be asked to cook and serve pasta (see listing 1), where the pasta is specified by a digital recipe found in the World Wide Web (e.g. on www.ehow.com)

```
(add-job
  (cook-and-serve
    (some meal
      :serving-time '(monday 12:00)
      :servings 4
      :meal-items '( (some meal-starter
                       :type 'soup
                       :recipe url-soup)
                     (some main-dish
                       :items ( (some pasta :type 'spaghetti
                                            :recipe url-pasta)
                                (some sauce :type 'bolognese
                                            :recipe url-sauce) )) ))))
```

Listing 1: Adding a new job for cooking and serving a meal

Sometimes a job cannot be accomplished without taking into account the remaining jobs of the robot. Jobs might be incompatible and require the robot controller to infer possible interferences between them in order to accomplish all of them reliably. If the robot is cleaning another room while the soup is sitting on the oven, there is a danger that the soup will overboil without the robot noticing it. Interferences are rarely so obvious and more often are buried deep within the routines.

The objective of the robot controller is to carry out the given jobs reliably and cost-effectively. The jobs specify conditions on the effects that the robot should have on its environment. If the robot is to provide lunch, then it should serve all components of a course at the same time, every component should be hot and nothing overcooked. An experimenter can assign the robot a score for a given period based on the number of jobs it has performed, the degree to which it has accomplished them, and the time it has spent.

The Robot

Let us turn to the physics-based model of the simulated robot, which is depicted in figure 3, and its basic control routines. The robot is modelled after an RWI B21 robot but equipped with two arms with grippers.

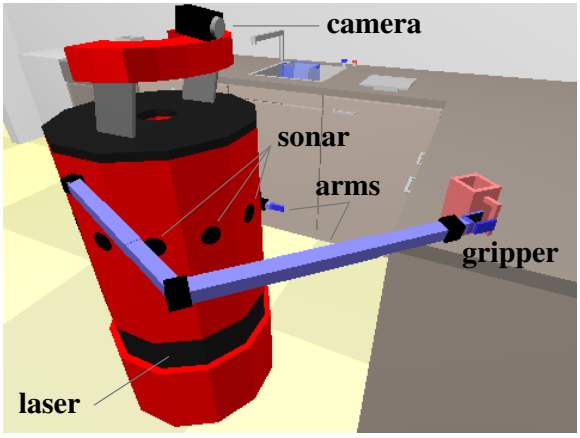

Figure 3: Simulated B21 Robot

The main sensor is a simulated camera that allows the robot to look for objects at its current location, track them, and examine them. Vision is limited to the camera's field of view. Since even this area is complex the robot does not try to perceive whole scenes but actively looks for the information needed for executing its routines. Thus, when activating a vision routine, the robot controller parameterizes the routine with the information it wants to extract from an image. For instance, the routine look−for can be called with "flat plate" as parameter. This routine returns a possibly empty set of designators, data structures that partially describe the "flat plates" and their positions in the image. Since the robot's camera is imperfect, designators will be inaccurate and the robot may overlook or imagine objects.

Our model of robot vision makes two important assumptions: that the vision routines can recognize objects satisfying a given symbolic description; and that the robot only sees what it is looking for. The vision model is defined using the following four functions. (1) *Objects within sensor range*: returns all objects in the sensor range of the camera matching the given features based on ground truth information. Currently the sensor range is the robot's current location. (2) *Objects seen*: predicate that returns true if an object is seen. The predicate can be implemented probabilistically and depending on the distance to the robot and the properties of the surroundings. (3) *Imagined objects*: returns a list of objects that are imagined by the robot sensors. (4) *Perceived properties*: given an object and a list of properties, the function returns the list of perceived property values. The perception can thus be noisified, e.g. in dim light a red cup is often perceived as violet.

The logical camera first calls the function *Objects within sensor range*. It then applies the predicate function *Objects seen* on every returned object, and removes overseen objects. In the next step the function *Imagined objects* adds probabilistically new objects to the list of seen objects. Finally the object properties are noisified with *Perceived properties*.

To grasp or manipulate an object, the robot must visually track the object, that is it must keep the camera pointed at the object to get a visual feedback for controlling the robot's hands. In addition to recognizing objects with given features and tracking, the cameras can examine objects, i.e., extract additional visual features of a tracked object. It goes without saying that only one process can control and use the camera at a time and that threads of control might have to wait for access to the camera.

Robot localization and navigation are considered to be solved. The robot is equipped with two 180° laser range finders and 12 sonars. In a kitchen environment with known positions of the pieces of furniture accurate laser-based localization has been demonstrated to work reliably. As perceptual data the robot controller gets a two-dimensional global pose (x, y, and φ) and the translation and rotation velocity of the robot. The controller can set either the translation and rotation velocity directly or can use the navigtion routines to change the pose of the robot.

For every arm a three-dimensional pose (x, y, z, φ_x, φ_y, and φ_z) of the gripper relative to the robot is perceived. The opening distance and the velocity of the grippers is also known to the controller. In addition, the grippers are equipped with sensors that measure the applied gripper force. If the gripper is holding an object this force measure is high, otherwise it is low. The gripper pose, its velocity, and the distance between the fingers are directly controllable.

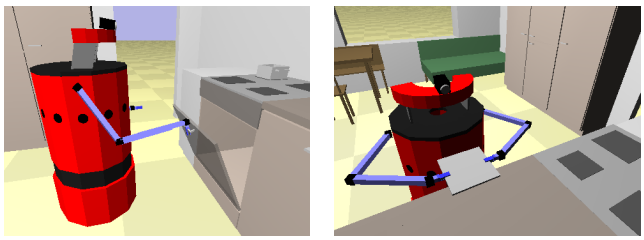

Figure 4: B21 routine plans in action. Open oven door (left). Put down plate (right).

We have now described the basic percepts and actions of the robot. In a later section we will show how we use these actions in routine plans for tasks such as reaching out a hand to the positions of tracked objects, grasping them and picking them up (see figure 4). The robot can also move around and manipulate food (which is greatly simplified). Besides receiving the pure percepts described here, we develop plans for actively perceiving and tracking objects and their properties. All this is performed in an uncertain environment, where perception can be erroneous or objects might slip out of the robot's hand while grasping them. The degree of perceptual uncertainty can be controlled by defining logical sensors, especially cameras that overlook or hallucinate objects, or noisify object descriptions.

Abstract Machine

We consider our control system as a "Robotic Agent" abstract machine. As in the notion of programming languages, our machine contains data structures, primitives and control structures, which are used to build higher-level programs. For our purposes, the main data structures are *fluents*,

variables that change over time and signal their changes to blocked control threads of sensor-driven robot plans. The most important primitive statements are continuous perception and control processes, such as localization and navigation processes, which are encapsulated in *process modules*. Finally, the means for specifying sophisticated behavior by combining and synchronizing are provided in the form of control structures including conditionals, loops, program variables, processes, and subroutines as well as high-level constructs (interrupts, monitors) for synchronizing parallel actions. We will specify the robot plans in RPL (Reactive Plan Language) (McDermott 1991).

Using the "Robotic Agent" abstract machine we will build a hierarchy of plans. Low level plans provide basic actions for the planner and are annotated with additional information for the planner. High-level plans have an explicit structure, so that the planner can "understand" these plans without additional annotations.

In the following we describe the parts of the abstract machine in more detail. Low- and high-level plans are explained in more detail later.

Fluents — Processing Dynamic Data. Successful interaction with the environment requires robots to respond to events and to asynchronously process sensor data and feedback arriving from the control processes. RPL provides *fluents*, registers or program variables that signal changes of their values. To trigger control processes fluents are often set to the truth value "true" in one interpreter cycle and reset in the following one. We call this setting and resetting of fluents "pulsing". Fluents are also used to store events, sensor reports and feedback generated by control processes. Moreover, since fluents can be set by sensing processes, physical control routines or by assignment statements, they are also used to trigger and guard the execution of plans.

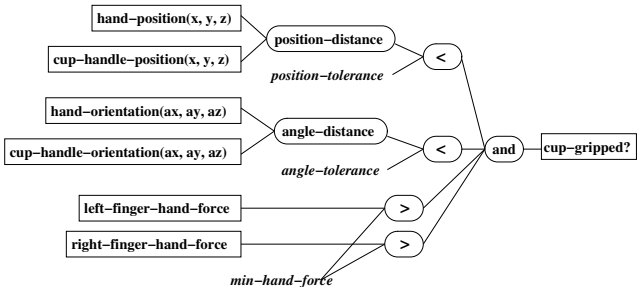

Figure 5: Fluent network computing the fluent cup−gripped?. The fluent network receives the hand and cup's handle poses (positions and orientations) and the hand-forces of the fingers as its input fluents. Parameters are written in italics, functions are denoted by ovals.

For example, the robot plans use fluents to store the robot's estimated position and the confidence in its position estimation. Fluents can also be combined into digital circuits (fluent networks) that compute derived events or states. For example, figure 5 shows a fluent network, which has the output fluent cup−gripped? that is true if and only if the distance between the hand pose (position and orientation) and the cup's handle pose is smaller than the allowed tolerances (position-distance and angle-distance) and the hand-force sensors of both fingers are above a certain minimum.

Fluents are best understood in conjunction with the RPL statements that respond to changes of fluent values. The RPL statement whenever f b is an endless loop that executes b whenever the fluent f gets the value "true." Besides whenever, wait−for f is another control abstraction that makes use of fluents. It blocks a thread of control until f becomes true.

Control Processes and Process Modules. Almost all modern control systems are implemented as distributed and asynchronously communicating modules (control processes) providing the basic means for moving the robot, sensing the surroundings of the robot, interacting with humans, etc. The modules can be distinguished into *server modules* and *robot skill modules*. Server modules are associated with a particular device. The camera server enables the robot to request an image of a particular type and obtain the image. The pantilt server allows the robot to point the camera into a specified direction. Other server modules include the laser, sonar, and the robot base server.

The robot skill modules provide the robot with specific skills, such as guarded motion, navigation planning, image processing, map building, and position estimation. The software modules communicate using an asynchronous message-passing library.

To facilitate the interaction between plans and continuous control processes, the abstract machine provides *process module*s. Process modules are elementary program units that constitute a uniform interface between plans and the continuous control processes (such as image processing routines or navigation routines) and can be used to monitor and control these processes.

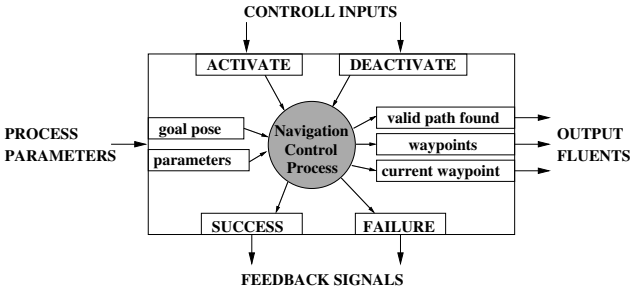

Figure 6: Process module encapsulating a navigation control process. Input parameters include the goal pose and the maximum acceleration and velocity. The output fluents provide information about the status of the current navigation task. The process module can be activated and deactivated and provides success and failure feedback.

A schematic view of process modules is shown in the example in figure 6. The navigation control process is encapsulated into a process module. Control processes can be activated and deactivated and return upon their termination success and failure signals. They can be parameterized and allow the plans to monitor the progress of their execution by updating fluents that can be read from the plan (e.g. the output fluent *current-waypoint*).

Control Process Composition. The abstract machine provides RPL's control structures for reacting to asynchronous events, coordinating concurrent control processes, and using feedback from control processes to make the behavior robust and efficient (McDermott 1991).

par $p_1 ... p_n$	(**par** (**achieve** (hand-at-pose 'left ...)) (**achieve** (hand-at-pose 'right ...)))
try-all $p_1 ... p_n$	(**try-all** (**achieve** (estimate-pos ... 'camera)) (**achieve** (estimate-pos ... 'laser)))
try-in-order $p_1 ... p_n$	(**try-in-order** (**achieve** (entity-in-hand ... 'left)) (**achieve** (enitity-in-hand ... 'right)))
with-policy $p\ b$	(**with-policy** (**maintain** (entity-in-hand ...)) (**achieve** (robot-at-pose ...)))

Figure 7: Some RPL control structures and their usage.

Figure 7 lists several control structures that are provided by RPL and can be used to specify the interactions between concurrent subplans. The control structures differ in how they synchronize subplans and how they deal with failures.

The par-construct executes a set of subplans in parallel. The subplans are activated immediately after the par plan is started. The par plan succeeds when all of his subplans have succeeded. It fails after one of its subplans has failed. The second construct, try−all, can be used to run alternative methods in parallel. The compound statement succeeds if one of the processes succeeds. Upon success, the remaining processes are terminated. Similarly try−in−order executes the alternatives in the given order. It succeeds when one process terminates successfully, it fails when all alternatives fail. with−policy p b means "execute the primary activity b such that the execution satisfies the policy p." Policies are concurrent processes that run while the primary activity is active and interrupt the primary if necessary. Additional concepts for the synchronization of concurrent processes include semaphores and priorities.

Design Issues and Programming Tools

Using the abstract machine described in the previous section and novel extensions to RPL that we will describe in this section we have designed, implemented, and experimented with variations of plan libraries for the household domain.

In our view, critical objectives for the design and implementation of plan libraries that are to be applied to the control of realistic robotic agents include the following ones:

1. *Performance.* First and foremost we are convinced that plans in a plan library that is to be employed by realistic robotic agents must be capable of producing robot behavior at a level of performance comparable to handcoded routines. This requirement seems to be non-controversial. Yet, accepting it has serious consequences: planning mechanisms cannot abstract away from behavior specifications for robots being concurrent programs that specify how the robot is to react to sensory input without losing the capability to understand and forestall many possible behavior flaws caused by its routines. For example, to explain why the robot might have grasped the wrong object, it might be necessary to look at a data structure stored in a local program variable that describes which object was used to parameterize the grasping routine. Also, to avoid many behavior flaws the planner has to modify the control structures in the concurrent plans.

2. *Cognitive capabilities.* We believe that the definition of plans being sequences of actions produced by planning systems and interpreted by execution systems in order to produce robot behavior is too narrow. Rather, we should, as McDermott (1992) does, consider robot plans to be any robot control program that cannot only be executed but also reasoned about and manipulated. This view has recently got additional support from work on more general cognitive capabilities of artificial systems. As Brachman argues: *"... systems that can reason from represented knowledge, can learn from their experience so that they perform better tomorrow than they did today, can explain themselves and be told what to do, can be aware of their own capabilities and reflect on their own behavior, and can respond robustly to surprise."* (Brachman 2002). It is obvious that this range of reasoning tasks in artificial cognitive systems cannot be accomplished without the representation of plans allowing for their successful realization.

3. *Failure monitoring and recovery.* If we aim at achieving high performance robot behavior two aspects of plans appear to be vital. First, the capabilities of monitoring the plan execution and recovering from execution failures. Second, mechanisms of execution time planning that can be used to specialize unconstrained execution parameters in order to optimize the robot behavior. For example, in order to produce optimal behavior for picking up a cup the robot has to wait until it sees the cup in order to commit to a particular position from where it intends to pick up the cup and how it intends to pick up the cup.

4. *Grounding and automatic model acquisition.* The models of plans used for planning should be grounded and the plans should acquire and maintain their models automatically. This requires that abstract states such as having a cup in its hand must be grounded into the sensor data and the experiences of the robot. For example, the robot believes that it has a particular object in its hand if it has successfully executed a subplan with the respective object description as an argument and the hand force has never dropped to zero since then. The robot must also have routines for achieving, maintaining and perceiving the respective abstract state.

In the remainder of this section we introduce several extensions to RPL that we have made in order to meet the design objectives listed above.

Behavior Monitoring and Failure Recovery

We achieve an effective integration of execution monitoring and failure recovery into plan-based control by introducing with−failure−handling as an additional control structure of the plan language RPL:

```
(with-failure-handling failure
  (recover <recovery-code>)
  (constraints <execution-constraints>)
  (perform <body>))
```

This control structure executes <body> such that <execution−constraints> are met. In addition, the code piece executes <recovery−code> in order to recover from failures of the <body>. Failures of <body> are bound to the loacal variable failure. The control structure coordinates the execution of goal achieving steps (<body>) and code pieces that aim at the reliable execution. with−failure−handling also makes the purpose of code pieces transparent for the planning mechanisms. If the code pieces produce goal achieving behavior they are to located in <body>, if they are responsible for creating and maintaining context conditions that enable reliable execution they are part of <execution−constraints>, and if they are to recover from signalled failures they must be specified as <recovery−code>.

Abstraction Mechanisms for the Plan Library

A key capability of a cognitive robot is that it can abstract away from details of its activities.

Defining Abstract States. One important form of abstraction is that the robot can form and reason about abstract states. For our household robot the abstract state of having an entity in one of its hands is an important abstract state. We will represent this state as the term (entity−in−hand ?entity ?hand) that maps a given entity ?entity and a hand ?hand into the set of all time instances in which ?entity is in the respective ?hand. We then use the predicate holds to assert that an entity is in a hand at a particular instance of time t using the fact (holds (entity−in−hand ?entity ?hand), t). The predicate holds is true if t is a member of the set of time instances denoted by (entity−in−hand ?entity ?hand).

A cognitive robot must know what abstract states mean. We take this to mean that the robot is capable of computing a belief with respect to this class of world states from the sensor data, the prior knowledge, and its experience. To do that we define that the robot has a belief class (entity−in−hand ?entity ?hand) and specify how the belief class is grounded in the perceptions of the robot. This is done in the following declaration:

```
(<=> (holds (entity-in-hand ?entity ?hand) ?t)
     ( (exists t')
       (AND (<= t' ?t)
            (succeeds (achieve (entity-in-hand ?entity ?hand) t'))
            (holds-tt (t' ?t) (force hand high))) ))
```

The declaration says that the robot believes in (holds (entity−in−hand ?entity ?hand), now) if it has successfully executed a plan with the name (achieve (entity−in−hand ?entity ?hand)) and the hand force sensor has never dropped to zero after completing this plan until now. Note, that such declarations enable the robot to exhibit certain capabilities of self-explainability, diagnostic reasoning, and advice taking, and make it more cognizant about itself and its operation. In particular, it can explain why it believes that it has the entity in its hand: because it had successfully picked it up and never let it go since then.

```
(def-abstract-state (entity-in-hand ?entity ?hand)
  :belief (<=> (holds (entity-in-hand ?entity ?hand) ?t)
               ( (exists t')
                 (and (<= t' ?t)
                      (succeeds
                        (achieve (entity-in-hand ?entity ?hand) t'))
                      (holds-tt (t' ?t) (force hand high)))) )
  :to-achieve ( (grip-entity (universal-routine) ...)
                (grip-cup (goal-operator)
                  :constraint (class-of ?entity 'cup)
                  :precondition (and (within-arm-range ?entity ?hand)
                                     (empty ?hand)))
                ... )
  :to-maintain ...
  :to-perceive ...)
```

Listing 2: Abstract State definition for (entity–in–hand ?entity ?hand)

To make a robot "aware" of abstract states takes more than only defining its beliefs with respect to this state. The robot should also know how to make this state true – if it is possible, how to maintain this state, how to prevent it, how to perceive it, etc. The respective declarations are shown in listing 2.

Besides specifying how the belief is computed with respect to the given abstract state, the declaration also specifies a set of routines for the different goal types (achieve, maintain, perceive, etc) that can be executed in order to achieve the goal. We distinguish two forms of goal achieving plans: *universal goal achieving routines* and *goal operators*. *Universal goal achieving routines* can be called in standard situations and will achieve their goal with high probability. *Goal operators* only work under specified preconditions and if the specified context conditions are satisfied. Because they assume certain assumptions to hold they can typically apply more specialized goal achieving mechanisms and therefore have higher expected performance if the execution context satisfies their requirements. Thus with each goal we have a set of routines for achieving it and possibly an arbitration mechanism using models for performance prediction in order to select the appropriate routine.

Abstract states are also asserted using a class specialization heirarchy. For example, we might have an abstract state class (entity−in−hand ?entity ?hand) and a specialization of it that restricts the entities to be small objects with handles and a further specialization for cups. In the future the specialization hierarchy will enable us to learn routines for novel classes of objects by retrieving the routines for similar objects and adapting them.

Partially Specified Parameters. Another aspect of behavior specification that has great impact on the performance of complex tasks in challenging environments is the use and dealing with partially specified parameters of plans. Consider a plan that is to specify how the robot is to pick up a cup. Clearly, which hand to use, which kind of grip to apply, how to position itself to grasp the cup reliably, which gripper and arm trajectory to take, and where to hold the cup after picking it up cannot be determined before seeing the cup. The optimal choice also requires often to take the situational context, expectations, and foresight into account. In our kitchen scenario, the hand to be used, the pose of the

hand during transportation, and the pose used to grip the cup decide on the reliability and efficiency with which the tasks of the subplans are executed.

To enable robots to reason about parameter values that are not known in advance we introduce specific language constructs that allow for their explicit and transparent representation. With these constructs programmers can specify the parameters that are not fully constrained by the plans calling them. They can be chosen at execution time in order to maximize the reliability and the efficiency of the generated behavior. Our plan interpretation mechanisms then call specific arbitration mechanisms that decide on the appropriate parameterization of the subplan calls.

```
(some pose
      :within-arm-range T :for-entity entity :with-hand hand)

(some trajectory
      :for-entity entity :with-hand hand
      :collision-free T :within-arm-range T)

(argmax
  #'(lambda (hand')
      (expected-performance (current-goal)
                            :bind ( (?hand . hand') )))
  (set-of h (and (class-of h 'hand) (empty h))))
```

Listing 3: Partly specified parameters.

Listing 3 shows three typical applications of partially specified parameters in our plans. The first one returns a robot pose such that the cup is within reach and the second one returns a collision free arm/gripper trajectory to the grip pose. The last is the most interesting one. It returns the hand to be used for grasping in order to achieve the highest expected utility. Determining this hand requires prediction, that is to mentally simulate the execution of the current goal and is therefore a decision based on planning.

An important property of these partially specified parameters is that they automatically collect their instantiations and the resulting behavior/performance. These data can then be used to learn optimal parameter settings from experience. Once learned, the partially specified parameters automatically use the learned values to optimize robot performance.

Plan Library

Having explained several aspects of plan-building we now show some example low-level and high-level plans from our plan library.

Low-level Plans

At some level of detail the planning and reasoning mechanisms abstract away from the implementation details of control routines and consider the respective routines to be black boxes with an associated behavior model. The associated model includes information necessary to predict what will happen when a black box routine gets executed and other annotated information for enabling various reasoning mechanisms. We call these black box routines *low-level plans*.

In this section we will start with presenting and explaining the behavior specification implemented by a low-level plan. Then we will sketch a prediction model for the behavior generated by the low-level plan and discuss the kinds of information that must be provided by this model. Finally, we present what additional pieces of information the low-level plans must provide to make the robot "aware" of them.

The decision of which routines we want to realize as low-level plans is a pragmatic one. By defining a routine as a low-level plan we assert that the behavior specification of this routine does not need to be reasoned about by the planning mechanisms. Thus, if the abstraction level of the low-level plans is too high, then the designers give away opportunities of plan-based behavior optimization, if the level is too low, then the search space for plan optimizations and the computational cost for optimization steps will be drastically increased without getting the proper pay-off in terms of behavior improvement.

Behavior Specifications for Low-level Plans. Listing 4 depicts a typical low-level plan of our household robot: grip−cup serves as a method to achieve the abstract state (entity−in−hand ?entity ?hand).

```
1  (define-plan grip-cup (?entity &optional ?hand)
2    (let ( (hand (if (bound-p hand)
3                     hand
4                     (argmax
5                       #'(lambda (hand')
6                         (expected-performance (current-goal)
7                                               :bind ( (?hand . hand') )))
8                       (set-of h (and (class-of h 'hand) (empty h)))))) )
9      (let ( (trajectory-poses (some trajectory :for-entity entity
10                                                :with-hand hand
11                                                :collision-free T
12                                                :within-arm-range T)) )
13       (par
14         (achieve (hand-at-pose hand
15                                (last-one trajectory-poses)
16                                trajectory-poses))
17         (achieve (fingers-open hand)))
18       (achieve (gripping hand)))))
```

Listing 4: Low-level plan definition of grip-cup

This plan consists of two parts: the plan steps needed to achieve the desired state and the instantiation of certain parameters that were left open by higher-level plans and have to be filled in now.

Lines 13–18 show the actual plan steps for gripping a cup. First the hand has to be navigated to a position reachable by the gripper. While the arm navigation is active, the gripper's fingers are opened in parallel. When the goal position of the hand is achieved, the gripper only has to be closed ((achieve (gripping hand))).

The main characteristic of a low-level plan is that execution parameters that are abstracted from on higher levels, are substantiated. In this case it is the choice which hand to use (lines 2–8) and the determination of the trajectory the hand is to navigate on (lines 9–12). The difference in these two parameters is that the hand trajectory can only be determined by the plan and cannot be prescribed by a super-plan. In contrast, the hand parameter is more abstract and might have been determined by a higher-level plan. Therefore the parameter ?hand can be passed to the grip−cup routine.

If the hand has not been specified when the routine is called, it is determined according to the situation. With models about the routine's performance we can determine the expected performance for each hand and choose the one with the highest value.

Predictive Models of Low-level Plans. Since grip−cup is a low-level plan, the planning and reasoning mechanisms of

the robot consider it to be a black box. Considered as black boxes, low-level plans can be activated and deactivated; they either signal their successful or unsuccessful completion. While running they change the computational state by setting fluents or plan variables and change the environment through manipulation and the physical state of the robot, for example by moving the arm.

We represent this black box abstraction as *projection rules* that describe the behavior of low-level plans, as models of what the plans do. The rules have the form (Beetz & Grosskreutz 2005):

```
(project (name <args>)
  condition
  ( d_1 ev_1 ... d_n ev_n )
  outcome)
```

If the effects of a low-level plan name are to be predicted the projector fires the projection rule. If a projection rule is fired it first checks whether condition is satisfied in the last timeinstant ti of the timeline. If so it appends the events ev_j at the time $ti + \sum_{1 \leq j \leq n} d_j$ to the timeline unless they have the form (computational−event <exp>). Every duration d_j is relative to the previous duration. If events have the form (computational−event <exp>) they describe changes in the computational state of the controller rather than changes in the world, i.e., these events set global variables and fluents of the controller. The outcome is either of the form (success) or (fail <failure−class> <failure−description>). This statement specifies the signal that is returned by the low-level plan when the plan is completed. This signal is caught and handled by the interpreter in projection mode.

Typically, to describe a low-level plan we need a set of projection rules that probabilistically describe the possible event sequences and outcomes when activating a plan. In each situation exactly one projection rule is applied, although the decision about which one it is might be probabilistic.

```
(project (grip−cup ?cup−desig ?hand)
  (and (pose ?hand ?hand−pose)
       (pose (handle ?cup−desig) ?handle−pose)
       (grip−succeeds ?hand−pose ?handle−pose)
       (approach−time ?hand−pose ?handle−pose ?dt))
  ( 0          (begin (grip−cup ?cup−desig ?hand))
    0          (computational−event (set−fluent arm−moving? true))
    3          (computational−event (set−fluent fingers−open? true))
    (− ?dt 3)  (computational−event (set−fluent arm−moving? false))
    2          (computational−event (set−fluent fingers−open? false))
    0          (end (grip−cup ?cup−desig ?hand))
    0          (computational−event
                 (update−desig ?cup−desig :in−hand ?hand)) )
  (success))
```

Listing 5: A predictive model of the low-level plan grip-cup.

Listing 5 shows a predictive rule that models the normal behavior the robot exhibits when running the low-level plan grip−cup. The condition of the projection rule determines where the hand and the cup's handle are and determines probabilistically whether the grip will succeed. The last condition estimates the time ?dt that the robot needs to approach the cup with its arm. Upon completion of the grip−cup plan the projected interpretation process sends a success signal. The projected behavior consists of seven events: two that change the world (begin (grip−cup ?cup−desig ?hand)) and (end (grip−cup ?cup−desig ?hand)), which occurs ?dt+2 later. The other events cause changes of the computational state of the plan-based controller: the update of the fluents and the designators used for controlling the robot.

Besides asserting the events that take place during the execution of a plan we have to specify how these events change the world. This is done by using *effect rules*. Effect rules have the form (e−>p−rule ev cond prob effects) and specify that an event ev under condition cond has with probability prob the effects effects. effects are either propositions which specify that ev causes the corresponding proposition to be true or have the form (clip props) which specifies that ev causes prop not to hold any longer.

```
(e−>p−rule
  (end (grip−cup ?cup−desig ?hand))
  (and (applied−gripper−force ?hand ?gf)
       (physical−track ?cup−desig ?real−cup)
       (rigidity ?real−cup ?rig)
       (≥ ?gf ?rig))
  0.92
  (broken ?real−cup))
```

Listing 6: An e−>p−rule describing the effects of the event (end (grip-cup ?cup-desig ?hand)).

How the event (end (grip−cup ?cup−desig ?hand)) changes the environment is projected by e−>p−rule rules. One of them is shown in listing 6. The rule specifies that if at a time instant at which the event occurs the applied gripper force ?gf is higher than the rigidity ?rig of the real cup ?real−cup (described by the designator ?cup−desig), then the real cup is broken afterwards with a probability of 0.92.

The projector also allows for modeling exogenous events. Exogenous event models are represented by rules of the form (p−>e−rule cond dur ev), where dur is the average time that ev occurs under condition cond. Thus each time before the projector asserts a new event δ timesteps later than the last event it asks whether cond holds and if so randomly decides whether an event of type ev has occurred within δ. If so it randomly generates a time δ' between 0 and δ at which ev is projected to occur.

Annotations for Low-level Plans. We specify additional annotations for low-level plans in order to enable the robot to learn and maintain grounded and accurate models of the low-level plans.

One annotation (updated−fluents) asserts which fluents are updated by the low-level plan and its subroutines. Knowing these fluents is important for the specification of complete predictive models. If the predictive model fails to update a fluent properly higher level plans that are triggered by these fluents can not be projected accurately.

Another annotation specifies the context conditions that determine whether a low-level plan will succeed or fail. For example, the success of the low-level plan grip−cup might depend on the orientation of the handle, the position of the cup and the robot, the free space, the slipperiness of the cup and the gripper, etc. The assertion of this causal relation and the implied independence from other state variables enables the robot to selectively collect data from experience and use the collected data to learn predictive models for plan success and failure using decision tree learning (Beetz, Kirchlechner, & Lames 2005). The learning of predictive models is also briefly sketched in the section "Results".

Other assertions about low-level plans that are used for more accurate plan models are the specifications for signalled plan failures. For instance, we can assert that the low-level plan grip−cup returns a failure grip−failure if the gripper is completely closed and the measured gripper force still low. The annotations of low-level plans are logical statements that assert plan properties needed by learning, planning, and other reasoning mechanisms for low-level plans.

High-level Plans

High-level plans are the plans that the planning mechanisms have to reason about. Their structure is explicit and transparent for the planner. So are the purposes of the subplans.

Reliable robot control routines often share a common structure. They have an inner part that specifies the steps for producing the normal behavior. Then additional execution constraints are wrapped around these goal achieving steps that make the behavior more robust and flexible and allow for situation specific behavior adaptations.

```
1  (define-plan pick-up-cup (?entity &optional ?hand)
2    (let ( (hand (if (bound-p hand)
3                     hand
4                     (argmax
5                       #'(lambda (hand')
6                         (expected-performance (current-goal)
7                                               :bind ( (?hand . hand') )))
8                       (set-of h (and (class-of h 'hand) (empty h))))))
9           (grip-pose (some pose :within-arm-range T
10                                :for-entity entity
11                                :with-hand hand))
12          (transportation-pose (some pose :safe-transportation-possible T
13                                          :for-entity entity
14                                          :with-hand hand)) )
15     (with-failure-handling failure
16       (recover ( (typep failure 'grip-failure)
17                  (retry-or-fail failure :max-tries 4) ))
18       (constraints
19         (scope (end-task go2pose)
20                (end-task carry)
21           (maintain (robot-at-pose grip-pose)
22             :severity :hard
23             :failures ( (:class pose-failure
24                          :fluent-trigger (robot-pose-changed)
25                          :recover-with pose-failure-during-pickup-handler
26                          :max-tries 2) )))
27         (scope (end-task grip)
28                (end-task carry)
29           (maintain (entity-in-hand entity hand)
30             :severity :hard
31             :failures ( (:class entity-lost-failure
32                          :fluent-trigger (not (inner-contacts hand))
33                          :recover-with entity-lost-during-pickup-handler
34                          :max-tries 3) ))))
35       (perform
36         (:tag go2pose
37           (achieve (robot-at-pose grip-pose)))
38         (:tag grip
39           (achieve (entity-in-hand entity hand)
40             :asserted-precondition (within-arm-range entity hand)))
41         (:tag carry
42           (achieve (entity-at-pose entity transportation-pose hand)
43             :asserted-precondition (in-hand hand entity)))))))
```

Listing 7: High-level plan definition of pick-up-cup

To accommodate this common structure we will describe the high-level plan in listing 7 from the inside to the out. The goal achieving steps are listed in lines 35–43. These steps specify that in order to achieve the state of having an entity of the type cup in its hand the robot has to (1) position itself at a pose such that the cup is in the working space of one of its arms (achieve (robot−at−pose grip−pose)); (2) get the cup in its gripper, but under the asserted condition that the cup is already within reach (achieve (entity−in−hand entity hand); and (3) to lift the cup up into a position that is safe for carrying (achieve (entity−at−pose entity transportation−pose hand).

Thus from the syntactic structure of the high-level plan the reasoning mechanisms can infer that the robot aims at achieving to have a cup in its hand by first achieving to be at a position from which it can pick up the cup, then achieving the cup to be in its hand, and finally achieving that the cup and the hand will be in the transporation position (lines 35–43). In addition, it is stated that the second and third goal-achieving steps assume the cup to be within arm range and the cup to be in the gripper when lifting the gripper to the transportation pose. The mechanisms are also capable of inferring that the maintenance goals (not moving away while picking the cup up and keeping the object in the hand (lines 19–34)) maintain a situational context for picking up the cup more reliably using the semantics of with−failure−handling.

The high-level plan is made robust by wrapping it into a subplan that monitors the generated behavior, detects flawed behavior and corrects it or recovers from local failure. The code pieces needed for reliable execution are specified in the lines 15–34. It is typical for reliable routines that the fraction of code needed for failure detection and recovery outweighs the fragments specifying the action steps by far.

In listing 7 the *execution constraints* comprise two maintenance goals. First, they ensure that the robot is not moving away while trying to grasp the cup ((robot−at−pose grip−pose), lines 19–26). Second, they ensure that the cup stays in the gripper after it has first been grasped successfully ((entity−in−hand entity hand), lines 27-34). The surrounding scope statements ensure that the maintenance goals have the appropriate temporal scope: that the robot does not move from the begin of the grasp until the successful completion.

Besides the state to be maintained, the maintenance goals have additional parameters that specify their behavior. Thus violations of the maintained states can be of three different levels of severity. If the severity level is rigid then any violation of the protected state will cause the subplan to fail immediately. For example, if an object to be transported breaks, there is no use in the completion of the transportation. If the severity is hard, then the maintenance subplan must restore the state to be maintained before the execution of the body is continued, because the state is necessary for the reliable execution of the body. Finally, if the severity is soft, then the state needs to be restored after the body has been completed.

The failures parameter specifies the failure handling and is more expressive. class specifies the class of failure that is generated when the maintenance fails. fluent−trigger specifies the fluent that signals failures of this type. recover−with defines the recovery method that is to be invoked upon the detection of failures of this type. Finally, max−tries specifies the frustration level of the subplan. That is, after how many failures the plan gives up in executing this subplan.

The recover part of the plan specifies recovery methods for failures thrown by subroutine calls that can possibly be handled successfully in the current execution context.

Results

In the last sections we have shown how robust, universal plans can be created. In order to keep plans simple and understandable for the planner, we abstract away from parameters on higher levels, which are filled in at run-time. In the plans this process is indicated by the some construct. Up to now we haven't stated, how to fill in these parameters. Our approach is to employ automatic learning techniques in order to make appropriate decisions in every situation.

For instance, we want to have a function telling the robot which hand to use in order to grab a cup. Therefore we learned a decision tree that maps the cup's relative position to the robot and a hand to the probability of being successful.

To learn this function, we put the robot on a fixed position in front of the work space. In 1000 runs we put a cup in different positions within the robot's reach. The robot then tried to grab the cup with each hand (in two different runs) and recorded if the trial was successful. This data was processed by the C4.5 decision-tree learner. Below is an example of a learned decision rule that predicts failures of the class *:grip-failure* if the robot attempts to pick up a cup in a distance of more than 1.12m and facing away from the cup ((> rel−angle−towards−cup 12.29°)). The rule has an expected predictive accuracy of 84.1% .

```
(and (> distance 1.12m)
     (> rel-angle-towards-cup 12.29°))
  84.1%
  ⟶  (fails :grip-failure ...)
```

One can see that the form of the decision rules is similar to the prediction models for the low-level plans and also provides a measure of the accuracy and therefore, decision tree learning is a suitable means for automatically acquiring and maintaining these rules. Also, the rule itself is meaningful in the sense that it can be used by reasoning mechanisms for explaining and diagnosing why a plan failed.

We have also run experiments in which we validated the impact of the failure handling on the reliability of the plans. If failures occur non-deterministically and when failed actions tend to change the state and thereby also change the situational context, the successful recovery after multiple occurrences of failures works as expected and improves the performance.

Related Work

The work presented in this paper is most closely related to the work of Firby and his colleagues (1996) on the design of general plan libraries and their application to trash collection tasks. In this paper we address tasks of a broader scope and higher sophistication. Also, Firby's plan libraries were not designed to be automatically reasoned about and optimized through learning. Plan language designs for autonomous robots were also proposed by Beetz (2002). We have taken some of these mechanisms and extended them in substantional ways. Another thread of research that aims at the introduction of abstract states into robot control are Williams' ideas on model-based programming (Williams *et al.* 2003). Again, plans that are transparent and explicit to the planner in the way that our representations are, is not a focus of Williams' research. The use of predictive models for optimizing partially specified parameters is also demonstrated by Stulp and Beetz (2005). We will apply their learning and execution mechanisms proposed to our parameter optimization problems.

Conclusion

In this paper we have presented a preliminary design of a plan library for a very challenging control task to be performed by a simulated autonomous robot. The novel properties of this plan library are that it is designed to achieve optimizing and reliable behavior and that it enables cognitive reasoning mechanisms such as behavior and plan explanation, advice taking, etc. at the same time. Prominent functionalities of the library include the use of modular and transparent failure monitoring and recovery handling, smart and prediction-based instantiation of partly bound parameters, and the use of annotations to automatically learn and maintain informative models of low-level plans. We consider these extensions of plan representations both to be necessary for as well as capable of generating high-performance robot behavior. This has been shown in a limited demonstration in the results section. The ultimate goal of our design is to realize a competent robotic agent that can optimize its behavior by adapting it to the particular environment and tasks at hand. The robot is to show planning and prediction capabilities, as well as explain and diagnose its behavior, take advice, and improve itself.

References

Beetz, M., and Grosskreutz, H. 2005. Probabilistic hybrid action models for predicting concurrent percept-driven robot behavior. *Journal of Artificial Intelligence Research*. accepted for publication.

Beetz, M.; Kirchlechner, B.; and Lames, M. 2005. Computerized real-time analysis of football games. *IEEE Pervasive Computing* 4(3):33–39.

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

Brachman, R. 2002. Systems that know what they're doing. *IEEE Intelligent Systems* 67 – 71.

Firby, R.; Prokopowicz, P.; Swain, M.; Kahn, R.; and Franklin, D. 1996. Programming CHIP for the IJCAI-95 robot competition. *AI Magazine* 17(1):71–81.

McDermott, D. 1991. A Reactive Plan Language. Research Report YALEU/DCS/RR-864, Yale University.

McDermott, D. 1992. Robot planning. *AI Magazine* 13(2):55–79.

Stulp, F., and Beetz, M. 2005. Optimized execution of action chains using learned performance models of abstract actions. In *Proceedings of the Nineteenth International Joint Conference on Artificial Intelligence (IJCAI)*.

Williams, B. C.; Ingham, M.; Chung, S. H.; and Elliott, P. H. 2003. Model-based programming of intelligent embedded systems and robotic space explorers. *Proceedings of the IEEE: Special Issue on Modeling and Design of Embedded Software* 9(1):212–237.