Towards Performing Everyday Manipulation Activities

Michael Beetz [a,*] Dominik Jain [a], Lorenz Mösenlechner [a], Moritz Tenorth [a]

[a] *Intelligent Autonomous Systems group, Technische Universität München*
Boltzmannstr. 3, 85748, Garching bei München, Germany

Abstract

This article investigates fundamental issues in scaling autonomous personal robots towards open-ended sets of everyday manipulation tasks which involve high complexity and vague job specifications. To achieve this, we propose a control architecture that synergetically integrates some of the most promising artificial intelligence (AI) methods that we consider as necessary for the performance of everyday manipulation tasks in human living environments: deep representations, probabilistic first-order learning and reasoning, and transformational planning of reactive behavior – all of which are integrated in a coherent high-level robot control system: COGITO. We demonstrate the strengths of this combination of methods by realizing, as a proof of concept, an autonomous personal robot capable of setting a table efficiently using instructions from the world wide web. To do so, the robot translates instructions into executable robot plans, debugs its plan to eliminate behavior flaws caused by missing pieces of information and ambiguities in the instructions, optimizes its plan by revising the course of activity, and infers the most likely job from vague job description using probabilistic reasoning.

Key words: reactive control, transformational planning, knowledge processing, statistical relational learning

1 Introduction

Enabling autonomous mobile manipulation robots to perform household chores exceeds, in terms of task and context complexity, anything that we have investigated in motion planning and autonomous robot control as well as in artificial intelligence so far. Household robots have to generate, debug and optimize a wide spectrum of

* Corresponding author.
Email addresses: beetz@cs.tum.edu (Michael Beetz), jain@cs.tum.edu (Dominik Jain), moesenle@cs.tum.edu (Lorenz Mösenlechner), tenorth@cs.tum.edu (Moritz Tenorth).

plans that must contain rich specifications of how actions are to be executed, what events to wait for, which additional behavior constraints to satisfy, and which problems to watch out for. Successful handling of such tasks calls for a sophisticated control scheme that must be based on more than one paradigm. In the following, we thus propose a hybrid control scheme that includes plan-based reactive control, probabilistic decision-making and plan parameterization, and automated plan acquisition from natural language sources.

Let us consider the task of table setting as an example. For people, "please set the table" suffices as an executable task. Since meals are everyday activities, people know what to put on the table and where, regardless of the context (be it a formal dinner or a regular breakfast with the family). People also know how to optimize the task by, for example, stacking plates, carrying cups in two hands, leaving doors open, etc., and they know how to deal with the open-endedness of the task domain. Being able to perform novel tasks both adequately and efficiently is certainly another key requirement.

The classical approach to solving novel tasks in novel contexts is action planning, which has been studied in artificial intelligence for almost half a century. To apply AI planning, we can state the task as a goal state, provide a library of atomic actions together with specifications of their preconditions and effects, and the AI planning methods will then determine a sequence of actions or a mapping of states into actions that transforms the current state into one satisfying the stated goal. Unfortunately, the way in which the computational problems are defined typically does not match the requirements for performing open-ended sets of household tasks for a number of reasons: We may not know precisely what the goal state is; Knowing *how* to perform an action may be much more important than knowing which actions to perform in which order; Large amounts of domain knowledge are required.

Luckily, much of the knowledge that we require to carry out novel tasks is readily available, as web pages such as `ehow.com` and `wikihow.com` provide step-by-step instructions for tasks such setting the table (Figure 10) or cooking spaghetti. Both web sites contain thousands of directives for everyday activities. The comprehensiveness of these task instructions goes well beyond the expressiveness of planning problem descriptions used in the area of AI action planning [8]. Even if the planning problems could be expressed, the number of objects and actions including their possible parameterizations would result in search spaces that are not tractable by the search algorithms of these planning systems. Thus, a promising alternative to generating plans from atomic actions as it is the standard approach in AI planning, which is still far away from generating plans with these types of complexities, is to look up the instructions for a new task from web pages and translate them into executable robot plans. Of course, such translations are not straightforward, because the instructions are not intended to be processed by machines and because tasks in general may be subject to uncertainty.

In this article, we investigate a novel computational model for task planning and execution for personal robots performing everyday manipulation tasks, in which

the execution of a task involves four stages:

(1) *Translation of the natural language instructions into an almost working but buggy robot plan.* Owing to the fact that web instructions are written to be executed by people with common sense knowledge, the instructions may contain ambiguities, missing parameter information and even missing plan steps.
(2) *Debugging of the plan.* In a second step, the above plan flaws are to be detected, diagnosed, and forestalled using transformational planning based on mental simulations of the plans in a simulated environment [5].
(3) *Plan parameterization.* Plans for everyday manipulation tasks depend on numerous parameters (e.g. the number of people taking part in a meal, specific preferences, etc.). We apply statistical relational learning techniques to infer this information based on previous observations.
(4) *Plan optimization.* Web instructions also fail to specify how tasks can be carried out efficiently. Thus, transformational planning is applied in order to find out, for example, that the table setting task can be carried out more efficiently if the robot stacks the plates before carrying them, if it carries cups in each hand, and if it leaves the cupboard doors open while setting the table [19].

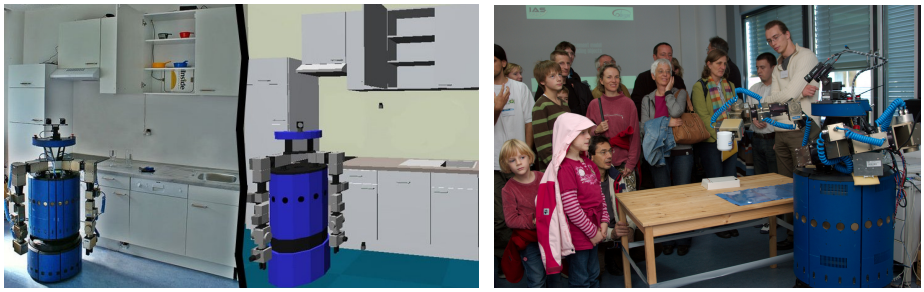
The key contribution of this article is the synergetic integration of some of the most promising AI methods, which we consider as necessary for the performance of everyday manipulation tasks in human living environments:

- We propose *deep representations* (i.e. representations that combine various levels of abstraction, ranging, for example, from the continuous limb motions required to perform an activity to atomic high-level actions, sub-activities and activities) and *knowledge processing* in order to enable the robot to translate natural language instructions into executable robot control programs. Abstract concepts from the instructions are mapped to geometric environment models and sensor data structures of the robot, enabling the robot to perform abstractly specified jobs.
- We apply *probabilistic first-order learning and reasoning* to enable the robot to successfully perform vague or underspecified jobs. For instance, a task such as "set the table" or "set the table but Anna will have cereals instead of fruits" can be performed by automatically inferring the most likely setup of the table.
- We realize powerful mechanisms for *transformational planning of reactive behavior* that is capable of eliminating flaws in plans generated from web instructions and that are caused by incomplete and ambiguous statements in the instructions. The transformational planning mechanisms are also capable of increasing the performance of table setting by 23-45% by making structural changes to the table setting plan.

1.1 Scenario

In this paper, we perform complex activities in a fairly realistic simulation environment shown in Figure 1a. In previous work [6], we investigated lower-level aspects of how to realize an action such as "putting an object on the table" in the context

of a table setting task, and we thus ignore these aspects in this article. Our robot is capable of recognizing the objects that are relevant to the task of table setting (i.e. plates, cups, glasses, etc), it can reliably pick them up and place them at the desired destinations, as continuously demonstrated during the open days at Technische Universität München (TUM) in October 2008 and 2009 (see Figure 1b). In this article, the focus is on the higher-level control framework of the system.



(a) (b)

Fig. 1. B21 robot: (a) real environment vs. simulation; (b) public demonstration at TUM

Let us assume that the robot is given the command "Set the table." Initially, the robot may have only basic knowledge about activities, e.g. that the goal of 'table setting' is the placement of objects on the table, where the items to be used by the individuals are located in front of the respective individual's seat. From this knowledge, the robot infers that table setting can be parameterized with the set of people expected to participate and the locations at which they sit. If the robot finds a plan for a given command in its plan library, it tries to infer the optimal parameters using learned probabilistic models. If there is no such plan, it queries the web to retrieve instructions (see Figure 7 (left) and Figure 10). These instructions are interpreted by the robot in terms of the concepts it knows about. Thus, the abstract concept "table" is linked to the model of the particular table in its environment, including its position, dimensions and information to parametrize navigation, visual search, and motion tasks (see Figure 2).

Once the instructions are interpreted, the robot roughly knows about the sequence of actions to be performed. For making the plan flexible and reliable, however, it needs to add code pieces for execution monitoring, failure detection, failure analysis, diagnosis and recovery. The resulting concurrent reactive plan comprises more than 2000 lines of code that have been generated automatically from the initial 12-lines of instructions.

Most instructions are missing essential information that is required for executing them, like the exact locations at which objects are to be placed, or additional actions that are necessary to make

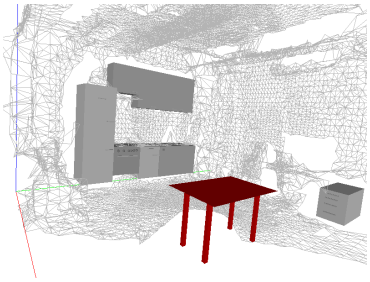


Fig. 2: Results of the query *owl_query(?O, type, "TablePieceOfFurniture")*, retrieving objects of the type "TablePieceOfFurniture" from the knowledge base.

an object reachable. In order to identify and debug these flaws, the robot "mentally" executes the plans using the robot's simulation capabilities. The result of such a simulation – the projected execution scenario depicted in Figure 3 – includes continuous data such as trajectories, discrete actions and events like the placing of objects, the intentions of the robot, and its belief state.

To recognize and diagnose execution failures, the robot reasons abstractly about the predicted execution scenarios. It queries whether the robot has overlooked particular objects, if unexpected collisions occurred, and so on. For the table setting instructions, one of the flaws that are detected this way is a collision with the chair while navigating to a location from to put down the place mat. To eliminate the flaw, the robot adds a supportive goal to temporarily put the chair out of the way and putting it back immediately after the table has been set.

After having debugged the plan, the robot further revises it in order to optimize its performance, e.g. by transporting objects using containers, using both grippers, or skipping repeated actions that collectively have no effect, such as opening and closing doors. The robot performs these optimizations by applying transformations to the plan and assessing the efficiency of the resulting plans.

In order to parametrize its plans, for instance to scale them to the correct number of people or adapt to their respective preferences, the robot uses probabilistic relational models that have been learned from observations of humans, e.g. of the participation of family members in various meals and the utensils and foodstuffs used. The respective queries are issued automatically by the planning system when an under-specified plan, for which a statistical model is available, needs to be parameterized.

1.2 System Overview

The core component of our system, as shown in Figure 4, is the COGITO plan-based controller [4,3,2]. Whenever COGITO receives a command, it first checks in its plan library whether it already has a tailored plan schema for this sort of command, the required parameterization and the respective situation context. If so, the appropriate plan schema is instantiated with the command parameters and consequently executed. COGITO logs any execution data as plans are carried out for later analysis. Plans that were classified as flawed are then put on the agenda for

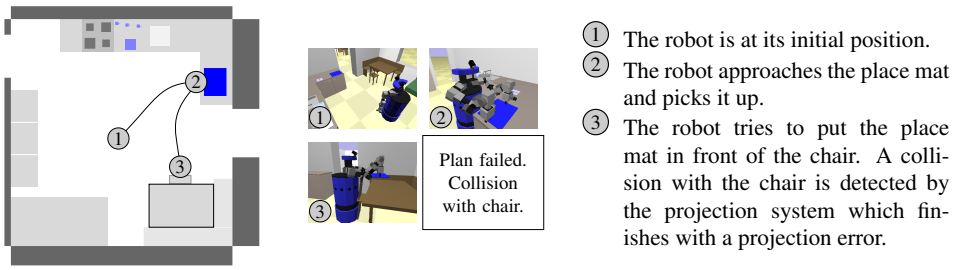


Fig. 3. The first projection of an execution scenario where the robot tries to put the place mat in front of the table.

future plan improvement.

When the robot is given the command to execute a task for which no plan can be found in its library, it imports web instructions, and transforms them into formally specified instructions in its knowledge base — by interacting with the PLANIMPORTER component of the system. A (usually buggy) plan is generated from the formal instructions, which is consequently debugged by repeatedly projecting the plan and generating an execution trace, criticizing it using this execution trace and revising it. We attempt to optimize the resulting plans by applying plan transformations like reordering actions, using containers for transporting objects or optimizing the usage of the robot's resources. The optimizations yield an optimized plan that is added to the plan library.

Another task that can be performed in idle situations is to learn, by interacting with the PROBCOG first-order probabilistic reasoning component of the system, statistical relational models of plan parameterizations, provided that the robot has collected observations describing the respective parameterizations. These models enable the robot to infer, for any given situation, a suitable set of parameters, allowing it to adapt its plans appropriately prior to execution [12].

2 Knowledge Processing for Autonomous Mobile Manipulation

COGITO employs various kinds of knowledge in order to accomplish its manipulation jobs successfully. It uses symbolic interfaces to a 3D object model of its operating environment, to observations of human manipulation actions for reasoning about objects and actions and to log data from robot activities. Furthermore, COGITO has access to general encyclopedic and common sense knowledge.

The concepts in the knowledge base are partly taken from the researchCyc ontology [16], partly extended to meet the special needs of mobile robotics (Figure 5). We furthermore make use of existing links between concepts in Cyc and sets of synonymous words in the WordNet lexical database [7] when resolving the meaning of words in web instructions.

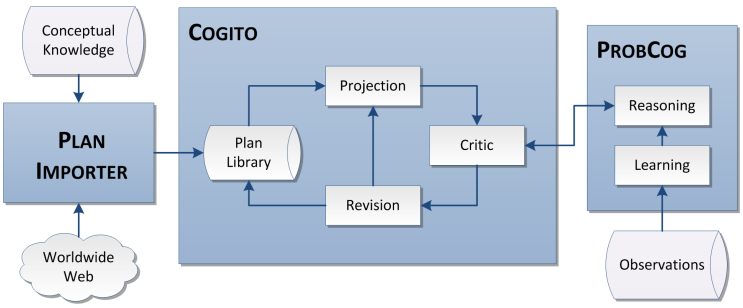


Fig. 4. System components: The PLAN IMPORTER allows to import plans from the Web, which can subsequently be debugged and revised by COGITO, the plan-based controller at the core of our system. PROBCOG provides the planning system with statistical learning and reasoning methods.

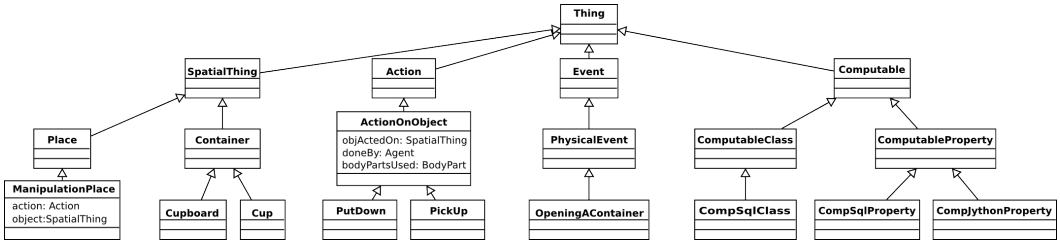


Fig. 5. Excerpt of the taxonomy in the knowledge base.

To integrate perceived information into the knowledge base, to infer abstract information from sensor data, and to translate symbolic specifications into parameterizations of control programs, the symbolic knowledge must be *grounded* in the basic data structures of the control system. In COGITO, the knowledge base is tightly coupled with perception and action modules (Figure 6) like a semantic environment map created from 3D laser scans [22], a visual object recognition system [14] and a markerless human motion tracker that helps the system learn from humans [1]. In addition, the knowledge representation cooperates very closely with the planning system, as described in the sections that follow. The knowledge processing system itself is described in more detail in [23].

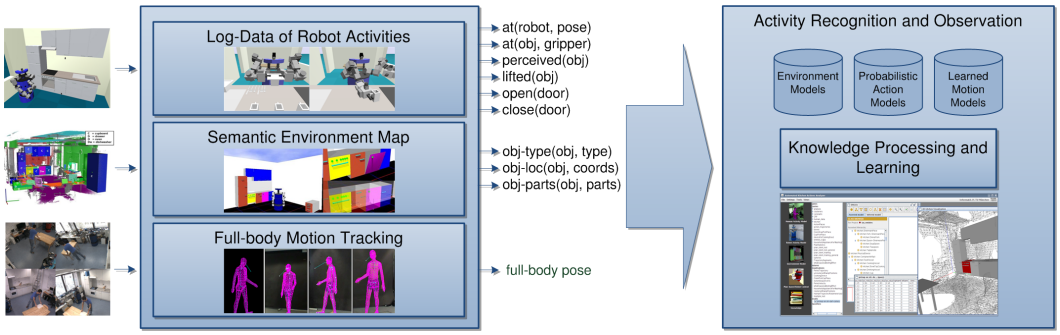


Fig. 6. Exemplary multi-modal sensor data that are integrated into the knowledge base.

3 Translating Instructions into Plans

In this section, we will present the steps involved in tranforming natural language instructions into an executable plan, based on the example sentence "Place a cup on the table". Figure 7 gives an overview of the structure of our system. (A more detailed description of the import procedure can be found in [24].)

3.1 Semantic Parsing

Starting from the syntax tree generated by the Stanford parser, a probabilistic context-free grammar (PCFG) parser, increasingly complex semantic concepts are generated in a bottom-up fashion using transformation rules.

Every leaf of the parse tree represents a word, *Word(label, pos, synsets)*, annotated with a label, a part-of-speech (POS) tag and the synsets the word belongs to (see Section 3.2). Examples of POS tags are *NN* for a noun, *JJ* for an adjective or *CD* for a cardinal number. In the following, an underscore denotes a wildcard slot that

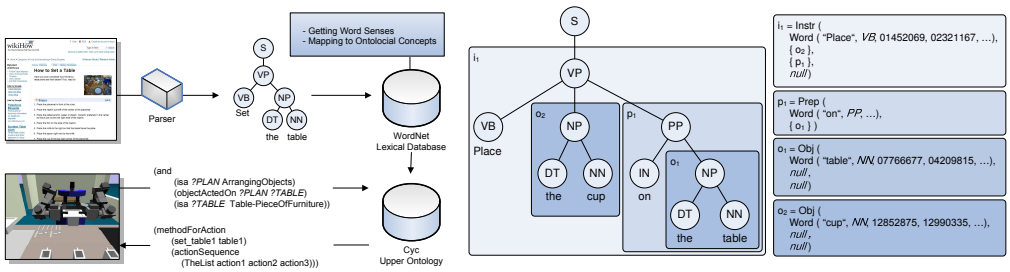


Fig. 7. Left: Overview of the import procedure. Center: Parse tree for the sentence "Place the cup on the table". Right: The resulting data structures representing the instruction created as an intermediate representation by our algorithm.

can be filled with an arbitrary value.

Words can be accumulated to a quantifier *Quant(Word(_,CD,_),Word(_,NN,_))* consisting of a cardinal number and a unit, or an object *Obj(Word(_,NN,_),Word(_,JJ,_), Prep, Quant)* that is described by a noun, an adjective, prepositional statements and quantifiers. A prepositional phrase contains a preposition word and an object instance *Prep(Word(_,IN,_),Obj)*, and an instruction is described as *Instr(Word(_,VB,_), Obj,Prep,Word(_,CD,_))* with a verb, objects, prepositional postconditions and time constraints. Since some of the fields are optional, and since the descriptions can be nested due to the recursive definitions, this method allows for representing complex relations like "to the left of the top left corner of the place mat". Figure 7 (center/right) exemplarily shows how the parse tree is translated into two *Obj* instances, one *Prep* and one *Instr*.

3.2 Word Sense Retrieval and Disambiguation

Once the structure of instructions has been identified, the system resolves the meaning of the words using the WordNet lexical database [7] and the Cyc ontology [16]. In WordNet, each word can have multiple senses, i.e. it is contained in multiple "synsets". There exist mappings from the synsets in WordNet to ontological concepts in Cyc via the *synonymousExternalConcept* predicate. "Cup" as a noun, for instance, is part of the synsets *N03033513* and *N12852875*, which are mapped to the concepts *DrinkingMug* and *Cup-UnitOfVolume* respectively.

Most queries return several synsets for each word, so a word sense disambiguation method has to select one of them. The algorithm we chose is based on the observation that the word sense of the action verb is strongly related to the prepositions (e.g. "taking something from" as *TakingSomething* up vs. "taking something to" as *PuttingSomethingSomewhere*). It is further explained in [24].

3.3 Formal Instruction Representation

With the ontological concepts resolved, a how-to can be formally represented as a sequence of actions in the knowledge base:

```
(methodForAction
  (COMPLEX_TASK ARG1 ARG2 ...)
  (actionSequence (TheList action1 action2  ...)))
```

Each step *action1*, *action2*, etc. is an instance of an action concept like *PuttingSomethingSomewhere*. Since the knowledge base contains information about required parameters for each concept, the system can detect if the specification is complete. For instance, the action *PuttingSomethingSomewhere* needs to have information about the object to be manipulated and the location where this object is to be placed.

Action parameters are created as instances of objects or spatial concepts, and are linked to the action with special predicates. In the example below, the *objectActedOn* relation specifies which object the action *put1* of type *PuttingSomethingSomewhere* is to be executed on. *purposeOf-Generic* is used to describe post-conditions; in this case, the outcome of the action *put1* shall be that the object *cup1* is related to *table1* by the *on-UnderspecifiedSurface* relation.

```
(isa put1 PuttingSomethingSomewhere)
(isa table1 Table-PieceOfFurniture)
(isa cup1 DrinkingMug)
(objectActedOn put1 cup1)
(purposeOf-Generic
      put1 (on-UnderpecifiedSurface cup1 table1))
```

3.4 Robot Plan Generation

In order to execute the formal instruction representation, it has to be transformed into a valid robot plan. Actions in the ontology can be directly mapped to high level plans of the plan library in COGITO. For instance, the action *put1*, acting on an object and a goal location, can be directly mapped to the goal $Achieve(Loc(Cup_1, Table_1))$. Action parameters, like object instances and locations, are linked to object references in a plan using designators. Designators are partial, symbolic descriptions of entities such as locations, objects or actions. During plan execution, solutions that fit the description are inferred and used to parametrize perception, navigation and the manipulation planner. Designators and goals are discussed in more detail in the next section.

4 Plan-based Control of Robotic Agents

In this section, we will first introduce the key concept of the computational model our plan based control is based on: the robotic abstract machine. Then we will give an overview of the transformational planning process that is used to debug and improve the plans imported from the WWW.

Our plans are implemented in extended RPL [17], a reactive plan language based on Common Lisp. A detailed discussion of transformational planning for plan optimization, plan projection and reasoning about plan execution can be found in [19] and [18].

4.1 The Robotic Agent Abstract Machine

We implement our control system using a "Robotic Agent" abstract machine. Our machine contains data structures, primitives and control structures for specifying complex behavior. In addition, the abstract machine provides means for combination and synchronization of primitive behavior in the form of control structures like conditionals, loops, program variables, processes, and monitors. Based on the "Robotic Agent" abstract machine, we implemented a plan library for high level actions, such as putting objects at locations. Plans form a hierarchy, ranging from the high-level actions necessary for executing WWW plans down to low-level actions such as moving the arm and grasping objects.

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*, program variables that signal changes of their values to blocked control threads. RPL supports fluents with two language constructs, namely *whenever* to execute code whenever a fluent becomes true, and *waitfor* to wait for a fluent to become true. By applying operators such as *and*, *or*, $>$, $<$ etc., fluents can be combined to fluent networks.

Designators. Designators are partial symbolic descriptions of objects, locations and actions that are described by conjunctions of properties. For example,

(object (**type** cup) (**color** blue) (on table))

describes a blue cup that is standing on the table. Designators are resolved at run time, based on the current belief state and knowledge about the environment. The properties of a designator are used to select and parametrize reasoning mechanisms that infer valid solutions for the designator. This includes spatial reasoning, 3D object models used by the vision system [14] and probabilistic inference [12].

Control Processes and Process Modules. To facilitate the interaction between plans and continuous control processes, the abstract machine provides *process modules*. Process modules are elementary program units that constitute a uniform interface between plans and continuous control processes (such as manipulation, navigation or perception) and that can be used to monitor and control these processes. More importantly, process modules allow to directly relate a physical effect with a program component. Grasping errors, for instance, always originate in actions that are controlled by the manipulation process module, and collisions between the robot torso and pieces of furniture originate in navigation activity controlled by the navigation process module.

A schematic view of a process module for navigation is shown in Figure 8. Control processes can be activated and deactivated and return, upon their termination, success and failure signals. They are parameterized by designators and allow the plans to monitor the progress of their execution by updating fluents that can be read by the plan (e.g. the output fluent *current-waypoint*).

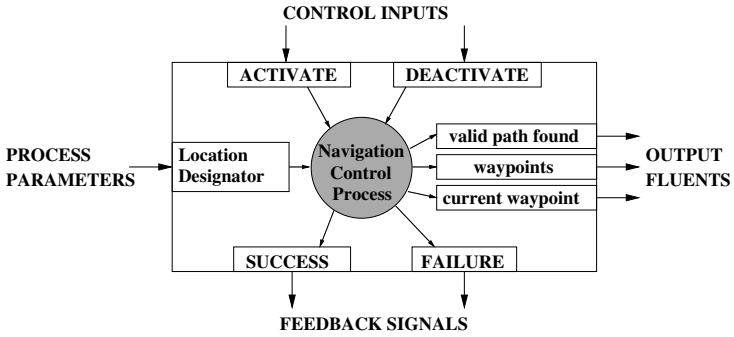


Fig. 8. Process module encapsulating a navigation control process. The input is a location designator, symbolically describing the goal pose and the context this code is executed in (e.g. *(location (to see) (object cup))*). The output fluents provide information about the status of the current navigation task.

4.2 Plan Representation

Debugging and optimizing plans for everyday manipulation tasks requires computational processes to infer the purpose of sub-plans, to automatically generate a plan to achieve a goal, detect behavior flaws, and estimate the utility of a behavior. These computational problems are, in their most general form, unsolvable, or at the very least computationally intractable.

To deal with this complexity, robot control programs must be implemented as declarative plans with symbolic annotations about the semantics (i.e. the intention) of the corresponding set of instructions. In this context, we define *occasions* and *goal statements*. Occasions are states of the world that hold over time intervals and are achieved (if not already true) by goals. A goal statement to express that the intention of the corresponding code is to achieve that the cup is on the table is written as follows: $Achieve(Loc(Cup, Table))$. A list of the most important occasion statements used in the current system can be found in Table 1a.

Goals are organized in a hierarchy, i.e. goals are achieved via sub-goals and, at the lowest level, by controlling process modules. This hierarchy is important in order to infer the intention of goal statements. When a goal statement is used within another goal statement, its intention is to help to achieve its parent goal.

If plans are implemented using such declarative statements, many reasoning problems like determining whether the robot has tried to achieve a certain state, why it failed, or what it believed, which are unsolvable for arbitrary robot control programs, essentially become a matter of pattern-directed retrieval.

4.3 Plan Projection and Transformational Planning

Figure 9 explains how a plan is debugged and transformed. After generation, the plan is added as a new candidate for the criticize-revise cycle. The search space of candidate plans is generated by criticizing the plan and thereby producing *analyzed plans* that are annotated with the behavior flaws they caused, their diagnosis, and their estimated severity. Criticizing is done by projecting a plan to generate an execution scenario, and by matching behavior flaw specifications against the scenario

to detect and diagnose flaws. The analyzed plans are added to the plan queue. In the *revise* step, the most promising analyzed plan is taken and all transformation rules applicable to the diagnosed flaws are applied to produce improved candidate plans. Each of these candidate plans is criticized and added as an analyzed plan to the plan queue. This search process continues until a plan is found that causes no behavior flaws that could be fixed.

To predict the effects of a plan, the plan is projected by executing it in a realistic ODE-based physical simulation and for every time instant data about plan execution, the internal data structures, the robot's belief state, the values of fluents, and the simulated world state including the exact locations of objects and the robot and exogenous events are logged. This includes continuous data, such as trajectories, as well as discrete data instances, such as task status changes.

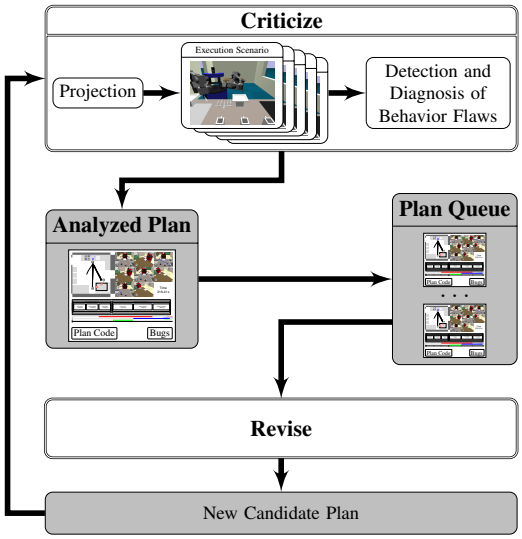


Fig. 9: The "criticize-revise" cycle. After having been generated from WWW knowledge, the plan is criticized by projecting it and querying it for bugs. Then a new plan revision is created and projected once more.

The plan projection generates a continuous data stream that is transformed into a first-order representation to reason about the plan execution and possibly unwanted side effects. The representation is based on *occasions*, *events*, *intentions* and *causal relations*. As already mentioned, occasions are states that hold over time

$Contact(obj_1, obj_2)$	Two objects are currently colliding
$Supporting(obj_1, obj_2)$	obj_t is standing on obj_b
$Attached(obj_1, obj_2)$	obj_1 and obj_2 are attached to each other.
$Loc(obj, loc)$	The location of an object
$Loc(Robot, loc)$	The location of the robot
$ObjectVisible(obj)$	The object is visible to the robot
$ObjectInHand(obj)$	The object is carried by the robot
$Moving(obj)$	The object is moving

(a) Occasion statements

$LocChange(obj)$	An object changed its location
$LocChange(Robot)$	The robot changed its location
$Collision(obj_1, obj_2)$	obj_1 and obj_2 started colliding
$CollisionEnd(obj_1, obj_2)$	obj_1 and obj_2 stopped colliding
$PickUp(obj)$	obj has been picked up
$PutDown(obj)$	obj has been put down
$ObjectPerceived(obj)$	The object has been perceived

(b) Event statements

Table 1
Occasion and event statements. Occasions are states that hold over time intervals and events indicate changes in the currently holding occasions.

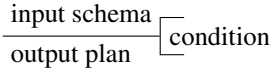
intervals. The term $\mathit{Holds}(occ, t_i)$ states that the occasion *occ* holds at time specification t_i. Time is specified either by the term $\mathit{During}(t_1, t_2)$ to state that the occasion holds during a sub-interval of $[t_1, t_2]$ or by the expression $\mathit{Throughout}(t_1, t_2)$ to state that the occasion holds throughout the complete time interval. Events indicate state transitions. In most cases, robot actions are the only cause of events. Table 1b gives an overview of the most important events. We assert the occurrence of an event *ev* at time t_i with $\mathit{Occurs}(ev, t_i)$. Occasions and events can be specified over two domains: the world and the belief state of the robot, indicated by an index of *W* and B for the predicates *Holds* and *Occurs* respectively. Thus, $\mathit{Holds}_W(o, t_i)$ states that o holds at t_i in the world and $\mathit{Holds}_B(o, t_i)$ states that the robot believes at time t_i that the occasion o holds at t_i. Syntactically, occasions are represented as terms or fluents. By giving the same name o to an occasion in the world as well as to a belief, the programmer asserts that both refer to the same state of the world. The meaning of the belief and the world states is their grounding in the log data of the task network and the simulator data respectively. Finally, we provide two predicates $\mathit{Causes}_{B\to W}(\mathit{task}, \mathit{event}, t_i)$ and $\mathit{Causes}_{W\to B}(o_W, o_B, t_i)$ to represent the relations between the world and beliefs. The former asserts that a task causes an event whereas the latter relates two occasion terms, one in the world state, one in the belief state, to each other. In other words, it allows to infer that a specific belief was caused by a specific world state.

Behavior flaws are defined in the first-order representation of the execution trace introduced above. As an example, we can state the occurrence of an unexpected event as follows:

$$\begin{aligned}&\mathit{UnexpectedEvent}(\mathit{event}, t) \Leftrightarrow\\ &\quad \mathit{Occurs}(\mathit{event}, t) \wedge\\ &\quad \neg\mathit{Member}(\mathit{event}, \mathit{ExpectedEvents}(t))\end{aligned}$$

Note that the plan interpreter is aware of the set of expected events at any point in time because every active process module generates a well-defined sequence of events. Unexpected events always indicate problems in the plan and are therefore a starting point for plan debugging. To debug a complex plan, we define a hierarchy of behavior flaws that describes errors such as unexpected events (e.g. unwanted collisions), unachieved goals (e.g. objects that were placed at wrong locations) and flaws concerning resource usage and performance (e.g. did the robot carry two objects at once).

The internal representation of behavior flaws allows the programmer to optionally define a transformation rule to fix it. Transformation rules have the form:



If the condition of a rule is satisfied by the projected execution scenario, then the parts matching the input schema are replaced by the instantiated schemata of the output plan. Note that transformation rules change the semantic structure of plans, e.g. by inserting plan steps, leaving out some steps, parallelizing them, or enforcing

a partial order on plan steps.

4.4 Plan Debugging in the Scenario

Using the machinery introduced above, we can revisit the application scenario and explain the debugging step in more detail. We consider a plan that was generated from the instructions in Figure 10.

The first detected behavior flaw is the collision of the robot with the chair. It is diagnosed as a *collision-caused-by-navigation* flaw, and one of the transformation rules that is applicable is the rule *remove-collision-threat-temporarily*. The rule produces a plan to move the chair in order to better reach to the table. This new plan is further investigated and produces another flaw — a *detected-flaw*, because the robot's grippers are already in use for carrying the place mat. The fix adds code to put down the place mat and pick it up later around the commands for moving the chair. While the robot further projects the plan, it detects several other flaws: For each object, there is a collision with the chair since the robot, per default, moves it back to its original position after each object manipulation. The *detected-flaw*, indicating that the grippers are not empty, is also detected and fixed for each object. This, of course, causes a highly sub-optimal plan that is optimized later after all flaws leading to substantial failures have been fixed.

All behavior flaw definitions and transformation rules used in this example were designed to be as flexible and general as possible. This means they work not only in this specific example but generalize to a wide range of different plans, for instance cooking pasta. In Section 6.2 (and, in more detail, in [19]), we show how plan transformation can not only fix bugs, but also speed up plans by up to 45%.

(1) Place the placemat in front of the chair.
(2) Place the napkin just left of the center of the placemat.
(3) Place the plate(ceramic, paper or plastic, Ceramic prefered) in the center so that it just covers the right side of the napkin.
(4) Place the fork on the side of the napkin.
(5) Place the knife to the right so that the blade faces the plate.
(6) Place the spoon right next to the knife.
(7) Place the cup to the top right corner of the placemat.

Fig. 10. Instructions to set a table from http://www.wikihow.com/Set-a-Table

5 Inferring Command Parameterizations from Vague Specifications

In the following, we consider the additional benefit of using probabilistic reasoning techniques to facilitate decision-making and to parameterize under-specified plans. The rigidity of the plans we obtain using the techniques outlined beforehand renders them specific to particular situations and requirements. Given the many circumstances under which a plan could be carried out, and the idiosyncrasies of the users for whom it is carried out, it seems natural to adapt the generated plans according to the needs at hand. Having identified the parameters of a plan from a logical description of the respective task, a robot can use statistical knowledge

on previously observed instances of the task to infer reasonable parameterizations, adding a new dimension to its control scheme.

Statistical models allow us to represent the uncertainty that is inherent in the robot's environment. Our models should thereby mostly represent *general principles* which are to be applicable to arbitrary instantiations of a domain, i.e. arbitrary situations involving varying numbers of relevant objects. Therefore, first-order languages, which allow universal quantification and thus abstract away from concrete objects, are a suitable basis for our models. In recent years, numerous approaches to combine first-order representations with the semantics of probabilistic graphical models have been proposed [9].

In our table setting example, we would want a probabilistic model to accurately represent the complex interactions between the participation of people in a meal, the attributes of the meal, the utensils used by the participants, and the food that is consumed. For any situation — involving any number of people, utensils and meals — the model should indicate a reasonable probability distribution over the set of possible worlds induced by the relevant predicates such as *takesPartIn*, *usesAnyIn* and *consumesAnyIn*.

5.1 System Integration

The top-level architecture to link our probabilistic reasoning engine to the overall system is shown in Figure 11. Whenever the robot control program is faced with a situation in which probabilistic inference is necessary, e.g. an under-specified task, it queries the probabilistic reasoning system by issuing a request consisting of the name of the model to use as well as a list of evidence variables and a list of query variables, where the variables are simply logical ground atoms. The PROBCOG reasoner, which manages a pool of probabilistic models, then processes the request by instantiating the selected model for the given set of objects, running the inference method, and finally returning the inference results in a reply. The robot controller processes the returned probabilities and uses them to parameterize its plans or to modify its control program in general.

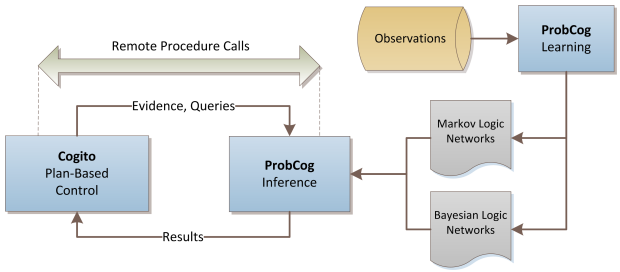


Fig. 11: Coupling of the plan-based control module (COGITO) and the probabilistic reasoning module (PROBCOG)

As a simple example, consider again our example problem of setting the table. Assume that we have been told that exactly three people will participate in breakfast, namely Anna, Bert and Dorothy — members of the family that are known to our model. To set the table, we need to know what utensils will be required at which seat; our problem thus translates to a probabilistic query as follows

$$\mathbf{P}(sitsAtIn(?p, ?pl, M), usesAnyIn(?p, ?u, M) \mid mealT(M, Breakfast) \wedge takesPartIn(P1, M) \wedge name(P1, Anna) \wedge takesPartIn(P2, M) \wedge name(P2, Bert) \wedge takesPartIn(P3, M) \wedge name(P3, Dorothy)) \quad \text{(Q1)}$$

The query will return, for each person and place, the probability of the corresponding *sitsAtIn* atom, and, for each person and utensil type, the probability of the corresponding *usesAnyIn* atom.

5.2 Representation Formalisms

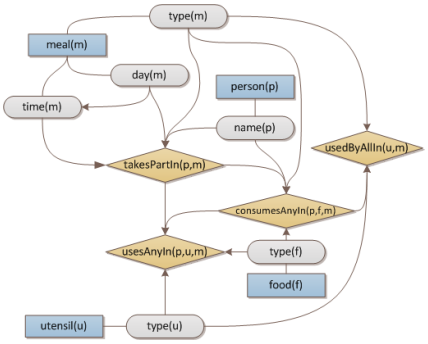
Many representation formalisms that combine first-order logic or a subset thereof with (undirected or directed) probabilistic graphical models have been proposed. *Markov logic networks (MLNs)* [21] are based on the former and are among the most expressive, for they indeed support the full power of first-order logic. The expressiveness of MLNs does come at a price, however, for not only is learning generally more problematic [11], inference also becomes more expensive and is therefore less well-suited to near-real-time applications. *Bayesian logic networks (BLNs)* [13], based on directed graphical models, are a sensible compromise between expressiveness and tractability. A BLN is essentially a collection of generalized Bayesian network fragments which are applicable to a random variable (ground atom) under certain circumstances, and which collectively define a template for the construction of a Bayesian network for any given set of objects/constants. In addition, a BLN may define arbitrary logical constraints on the probability distribution in first-order logic, such that global dependencies between variables may be adequately formulated. Combining these constraints with the ground Bayesian network yields the full ground model, which is thus a mixed network [15] with probabilistic and deterministic dependencies. Typically, if there are few hard global constraints, inference in BLNs is much more efficient than inference in an equivalent MLN.

5.3 Learning and Inference

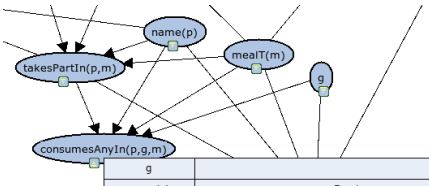
For learning the models, we assume that the structure, i.e. a specification of possible dependencies, is given by a knowledge engineer.A simplified structure of the table setting model is shown in Figure 12a. We can translate such a structure into either conditional dependencies (as in a BLN) or logical formulas (features of MLNs).

The PROBCOG learning stage then uses a training database containing a list of ground atoms (corresponding to sensory observations) in order to learn the model parameters that best explain the observations. The training database is obtained by collecting data from sensors like RFID sensors (in cupboards, on tables and in gloves worn by kitchen users), laser range scanners, and cameras, and translating it into the required logical format. For the table setting model, the configurations in which the table has been set can, for instance, be observed by an overhead camera and RFID sensors. The generation of logical ground atoms is then straightforward.

The learning algorithms are based on either maximum likelihood or MAP estimation. In MLNs, even learning needs to be done approximately; pseudo-likelihood



(a) Dependency structure



g	Bread							
name(p)	Frank							
mealT(m)	Dinner		Lunch		Breakfast		Dinner	
takesPartIn(p,m)	True	False	True	False	True	False	True	False
True	16.7%	0.0%	0.0%	0.0%	96.7%	0.0%	87.5%	0.0%
False	83.3%	100.0%	100.0%	100.0%	3.3%	100.0%	12.5%	100.0%

(b) Fragment parameters

Fig. 12. Bayesian logic network for the table setting model (excerpt): (a) Edges indicate dependencies; rectangular nodes, rounded rectangular nodes and diamond-shaped nodes correspond to object classes, attributes thereof and relations respectively. (b) Conditional distribution attached to a fragment.

methods are usually used. In BLNs, which make the causal structure of the model explicit, exact maximum likelihood learning is particularly simple, as it essentially reduces to counting occurrences of parent-child configurations in the data. Figure 12b shows an exemplary part of a fragment of the table setting model indicating the conditional distribution of the predicate *consumesAnyIn(person, food, meal)*.

Since the models are to be queried by a robot controller, they require that results are obtained in near real-time. Given the NP-hardness of probabilistic inference, we usually resort to approximate inference techniques. For BLNs, the PROBCOG inference module supports various sampling algorithms like SAT-based importance sampling or the SampleSearch algorithm [10]. For MLNs, the only inference algorithm that has proved to produce accurate results in real-world situations is MC-SAT [20], a Markov chain Monte Carlo algorithm.

As an example inference task, consider the query (Q1). In our model, it produced the results listed in Figure 13a, which imply the configuration shown in Figure 13b when assuming for each person the most likely seating location and assuming that *usesAnyIn* atoms with a probability over 0.05 should be considered as likely. Notice that the results change considerably if we remove from the evidence the identities of the three people (Figure 13c).

5.4 Integration with the Control Program

In order to integrate probabilistic inference into the plan-based controller, the plan language was extended with a new language construct, *likely-let*. It establishes a binding of variables to tuples of atoms and the corresponding probabilities within the current lexical context, based on a set of queries

```
(likely-let
  ((places
     :query
      '(sitsAtIn ?person ?seating-location M)
     :argmax ?person)
   (utensils
     :query '(usesAnyIn ?person ?utensil M)
     :threshold 0.05)
   :evidence
     '((takesPartIn P1 M) (name P1 "Anna")
       (takesPartIn P2 M) (name P2 "Bert")
       (takesPartIn P3 M) (name P3 "Dorothy")
       (mealT M "Breakfast")))
  (...))
```

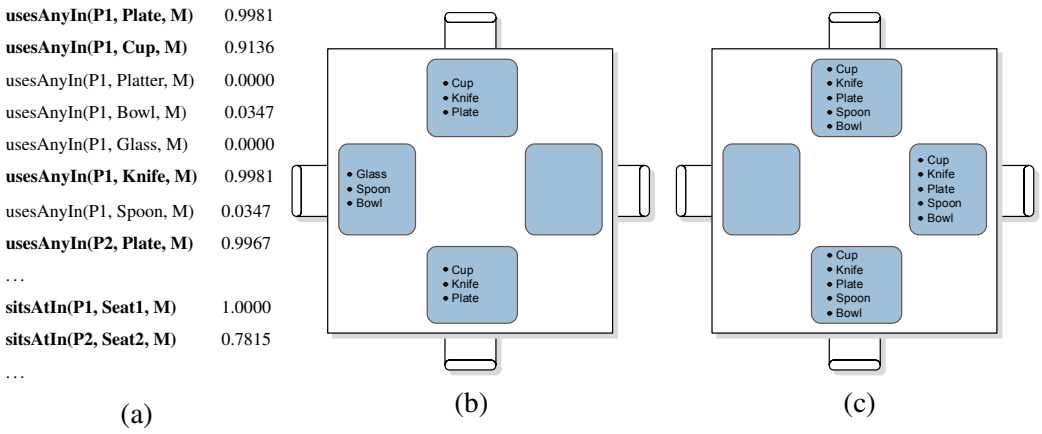


Fig. 13. Inference results: (a) Results for query (Q1) and (b) a corresponding abstracted interpretation; (c) interpretation of results for a second query where the identities of participants are unknown.

and a set of evidences. *likely-let* also provides support for post-processing returned probability distributions, e.g. an argmax operator or a threshold operator.

As an example, consider once again (Q1). We query the seating locations and the objects used by the participants as shown to the right, applying suitable post-processing operators (*argmax* and *threshold*) to the results. The plan is then carried out simply by iterating over the matching elements of the set generated by combining the elements of *places* and *utensils*. As this example shows, probabilistic inference can provide a sound way of parameterizing under-specified plans and can furthmore provide the control framework with general decision-making capabilities.

6 Experimental Results

We now provide a brief evaluation of the parts of our hybrid control architecture for which an experimental, statistical evaluation is appropriate. To a large degree, however, our architecture simply enables a robot system to perform tasks it was previously unable to handle; thus a quantitive analysis is infeasible.

6.1 Evaluation of the Import Procedure

The performance of the import procedure depends largely on the correctness and completeness of different, partly external software components. In order to give a more detailed evaluation, we not only examined the complete system but also looked at individual modules.

A primary source of error is the syntax parser. Although the Stanford parser that is used in this system is known to be one of the best of its kind, it still runs into problems as sentences get longer. To better show the influence of the parser on the recognition rate, we evaluated the system both with automatically parsed syntax trees and manually created ones. Another main issue affecting the recognition rate

are missing mappings from synsets in WordNet to the corresponding ontological concepts in Cyc. In the training set, we manually added 72 mappings for actions, objects and adjectives. Finally, we analyzed how many instructions are correctly transformed into the internal data structures before being added to the knowledge base. In the following, "instruction" refers to one step of a "how-to", i.e. one specific command.

Our training and test sets were made up of 88 and 64 instructions respectively, taken from `ehow.com` and `wikihow.com` how-tos pertaining to household activities. First, we trained the disambiguator on the training set with manually created parse trees. Afterwards, we ran the system including the syntax parser on the same set of how-tos. The results are shown in Table 2. With correct parse trees, the system achieves a recognition rate of 82% on the training set and even 91% on the test set before the ontology mapping and the transformation of the instructions into the formal representation. The remaining 18% resp. 9% have either been recognized incorrectly (missing object or preposition in the instruction) or not at all. The latter group also comprises instructions that are not expressed as imperative statements and, as such, are not supported by the current implementation. In both test runs, errors caused by the syntax parser result in a significant decrease in the recognition rate when switching from manually parsed to automatically parsed sentences (15 percentage points in the training set, 22 in the test set).

Training Set:	**aut. parsed**		**man. parsed**	
Actual Instructions	**88**	**100%**	**88**	**100%**
Correctly Recognized	59	67%	72	82%
False Negative	29	33%	16	18%
False Positive	4	5%	2	2%

Test Set:	**aut. parsed**		**man. parsed**	
Actual Instructions	**64**	**100%**	**64**	**100%**
Correctly Recognized	44	69%	58	91%
False Negative	20	31%	6	9%
False Positive	3	5%	6	9%

Table 2
Summary of the evaluation on instruction level; recognition rates before mapping the instructions to concepts in the knowledge base.

Table 3 shows the results of the translation into the formal instruction representation. In the training set, 70 of the 72 instructions which have been recognized in the previous step could successfully be transformed. The two errors were caused by mappings of word senses to concepts that cannot be instantiated as objects in Cyc: the concept *PhysicalAmountSlot* in the commands "Use the amount that..." and the relation *half* in "Slice in half".

The results of the translation of the test set show that two external components are the main sources of error: 40% of the import failures are caused by the syntax parser, since a decrease from 61% to 21% of failures in the initial recognition step can be observed when switching

Test set of how-tos	**Instr. Level**	**KB Level**	**KB+maps**
How to Set a Table	100%	100%	100%
How to Wash Dishes	92%	46%	62%
How to Make a Pancake	93%	73%	81%
How to Make Ice Coffee	88%	63%	88%
How to Boil an Egg	78%	33%	57%

Table 4: Evaluation of the import procedure per how-to.

Training Set:	**aut. parsed**		**man. parsed**	
Actual Instructions	**88**	**100%**	**88**	**100%**
Import Failures	**31**	**35%**	**18**	**20%**
Incorrectly/Not recognized	29	94%	16	89%
Missing WordNet entries	0		0	
caused Import Failures	0	0%	0	0%
Missing Cyc Mappings	0			
caused Import Failures	0	0%	0	0%
Misc. Import Errors	2	6%	2	11%
Disambiguation Errors	0		0	
Correctly imported into KB	**57**	**65%**	**70**	**80%**

Test Set:	**aut. parsed**		**man. parsed**	
Actual Instructions	**64**	**100%**	**64**	**100%**
Import Failures	**33**	**52%**	**28**	**44%**
Incorrectly/not recognized	20	61%	6	21%
Missing WordNet entries	3		3	
caused Import Failures	2	6%	2	7%
Missing Cyc Mappings	14		23	
caused Import Failures	11	33%	20	71%
Misc. Import Errors	0	0%	0	0%
Disambiguation Errors	2		3	
Correctly imported into KB	**31**	**48%**	**36**	**56%**

Table 3
Summary of the evaluation on knowledge base level. Recognition rates after mapping the words to concepts in the knowledge base.

from automatic parsing to manually
created syntax trees. In this case, missing Cyc mappings and WordNet entries are the main problem, causing about 78% of the remaining errors. An evaluation per how-to (Table 4) shows that a reasonably large number of the instructions can be recognized correctly. The last column contains the results after having added in total eight mappings, including very common ones like *Saucepan* or *Carafe*, which will also be useful for many other instructions. The generation of a robot plan from the formally represented instruction is a rather simple translation from Cyc concepts to RPL statements which did not produce any further errors.

6.2 Plan Optimization

As mentioned in Section 4.4, plans are optimized for performance after debugging. The set of transformation rules that is needed for optimization is, for pick-and-place tasks, very limited because most of the flaws result from increased resource usage (e.g. unnecessary repetitions). Note that errors such as dropping a knife do happen, but are normally fixed locally. That means when the robot drops an object, it picks it up again and continues with its original plan. The following list informally shows some of the most important transformation rules used in the optimization step:

- **if** the robot is to carry multiple objects from place P_1 to place P_2 **then** carry a subset of the objects by stacking them or using a container;
- **if** the robot moves objects repeatedly to a temporary location, performs some actions and moves the object back **then** move it once, perform all actions in between and move the object back after completing the actions.
- **if** the robot has to place objects at positions $P_1, \ldots, P_n$ and $P_1, \ldots, P_n$ are within reach when standing at location L **then** perform the place tasks standing at location L.

To test the performance improvement of transformational plan optimization, we hand-coded a default table-setting plan which places a cup, a plate and cutlery for an arbitrary number of people on the table. The plan was carefully designed to be

highly general and robust in order to work in most possible kitchen environments, but this leads to lower performance. We compared the default plan with eleven alternative plans generated by applying transformation rules.

The hand-coded plan was applied to situations in which the table was to be set for a varying number of people in two different dining rooms. The corresponding experiments examine between 168 and 336 plan executions. Depending on the experimental setting, an average run took between 5.4 min and 6.5 min. The total plan execution time for the experiments was approximately five days. Table 5 shows the performance gains (in percent) of the best plan compared to the default plan of the experiment. The gain is calculated by using the duration as performance measure and ranges from 23.9% to 45.3%.

persons	kitchen table	living-room table
A,T	23.9 %	30.1 %
T,D	39.4 %	45.3 %
T,S	30.2 %	36.9 %
A,S	31.5 %	33.4 %
A,T,S	24.5 %	34.8 %
A,T,D	29.5 %	39.5 %
T,S,D	34.6 %	42.4 %
A,T,S,D	32.0 %	42.7 %

Table 5: Summary of best plans showing the performance gain achieved by plan transformation in various scenarios, where the performance measure was time consumption.

7 Conclusion

We have presented a high-level control architecture that synergetically integrates a number of highly promising AI techniques on top of a reactive planning system in order to enable an autonomous robot system to learn novel tasks, execute them reliably and efficiently, and to deal with the uncertainty that governs the task domain. As indicated by our experiments, the transformation of web instructions into executable robot plans is an elegant way of acquiring initial knowledge about tasks. The transformational planning system is capable of correcting flaws in imported plans by means of projection and reasoning, and it can furthermore optimize plans for increased performance. The incorporation of probabilistic reasoning capabilities enables the system to deal with underspecified tasks, and provides general decision-making tools, adding another dimension to the control architecture.

We firmly believe that the combination of such methodologies is fundamental in achieving efficient, reliable and adaptive behavior in robotic agents. So far, the capabilities of our system, like those of our robot hardware and low-level control, are limited to pick-and-place tasks for objects of daily use. The procedures for importing and debugging plans largely scale to this domain, though some adaptations to the ontology may be required if the instructions contain unknown objects or actions. The debugging rules already cover many common failures and are formulated in a general way, but for new kinds of failures, new rules would have to be written as well. Scaling the system for a broader range of everyday manipulation tasks beyond pick and place actions will require us to consider tool usage, to reason about actions that modify objects and the state of the environment in general, and to consider (naive) physics. These are subjects of our ongoing investigations.

Acknowledgments

This work is supported in part within the DFG excellence initiative research cluster *Cognition for Technical Systems – CoTeSys*, see also `www.cotesys.org`.

References

[1] Jan Bandouch, Florian Engstler, and Michael Beetz. Accurate human motion capture using an ergonomics-based anthropometric human model. In *Fifth International Conference on Articulated Motion and Deformable Objects (AMDO)*, 2008.

[2] M. Beetz and D. McDermott. Declarative goals in reactive plans. In J. Hendler, editor, *First International Conference on AI Planning Systems*, pages 3–12, Morgan Kaufmann, 1992.

[3] Michael Beetz. *Concurrent Reactive Plans: Anticipating and Forestalling Execution Failures*, volume LNAI 1772 of *Lecture Notes in Artificial Intelligence*. Springer Publishers, 2000.

[4] Michael Beetz. Structured Reactive Controllers. *Journal of Autonomous Agents and Multi-Agent Systems. Special Issue: Best Papers of the International Conference on Autonomous Agents '99*, 4:25–55, March/June 2001.

[5] Michael Beetz. *Plan-based Control of Robotic Agents*, volume LNAI 2554 of *Lecture Notes in Artificial Intelligence*. Springer Publishers, 2002.

[6] Michael Beetz et al. Generality and legibility in mobile manipulation. *Autonomous Robots Journal (Special Issue on Mobile Manipulation)*, 28(1):21–44, 2010.

[7] C. Fellbaum. *WordNet: an electronic lexical database*. MIT Press USA, 1998.

[8] Maria Fox and Derek Long. Modelling mixed discrete-continuous domains for planning. *Journal of Artificial Intelligence Research*, 27:235–297, 2006.

[9] Lise Getoor and Ben Taskar. *Introduction to Statistical Relational Learning (Adaptive Computation and Machine Learning)*. The MIT Press, 2007.

[10] Vibhav Gogate and Rina Dechter. SampleSearch: A Scheme that Searches for Consistent Samples. In *AISTATS*, 2007.

[11] Dominik Jain, Bernhard Kirchlechner, and Michael Beetz. Extending Markov Logic to Model Probability Distributions in Relational Domains. In *30th German Conference on Artificial Intelligence (KI-2007)*, pages 129–143, 2007.

[12] Dominik Jain, Lorenz Mösenlechner, and Michael Beetz. Equipping Robot Control Programs with First-Order Probabilistic Reasoning Capabilities. In *International Conference on Robotics and Automation (ICRA)*, 2009.

[13] Dominik Jain, Stefan Waldherr, and Michael Beetz. Bayesian Logic Networks. Technical report, IAS Group, Fakultät für Informatik, Technische Universität München, 2009.

[14] Ulrich Klank, Muhammad Zeeshan Zia, and Michael Beetz. 3D Model Selection from an Internet Database for Robotic Vision. In *International Conference on Robotics and Automation (ICRA)*, 2009.

[15] Robert Mateescu and Rina Dechter. Mixed deterministic and probabilistic networks. *Annals of Mathematics and Artificial Intelligence*, 2008.

[16] C. Matuszek, J. Cabral, M. Witbrock, and J. DeOliveira. An introduction to the syntax and content of Cyc. *2006 AAAI Spring Symposium*, pages 44–49, 2006.

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

[18] Lorenz Mösenlechner and Michael Beetz. Using physics- and sensor-based simulation for high-fidelity temporal projection of realistic robot behavior. In *19th International Conference on Automated Planning and Scheduling (ICAPS'09)*, 2009.

[19] Armin Müller, Alexandra Kirsch, and Michael Beetz. Transformational planning for everyday activity. In *17th International Conference on Automated Planning and Scheduling (ICAPS'07)*, pages 248–255, 2007.

[20] Hoifung Poon and Pedro Domingos. Sound and Efficient Inference with Probabilistic and Deterministic Dependencies. In *AAAI*. AAAI Press, 2006.

[21] Matthew Richardson and Pedro Domingos. Markov Logic Networks. *Mach. Learn.*, 62(1-2):107–136, 2006.

[22] Radu Bogdan Rusu, Zoltan Csaba Marton, Nico Blodow, Mihai Dolha, and Michael Beetz. Towards 3D Point Cloud Based Object Maps for Household Environments. *Robotics and Autonomous Systems Journal*, 2008.

[23] Moritz Tenorth and Michael Beetz. KnowRob — Knowledge Processing for Autonomous Personal Robots. In *IEEE/RSJ International Conference on Intelligent RObots and Systems.*, 2009.

[24] M. Tenorth, D. Nyga, and M. Beetz. Understanding and executing instructions for everyday manipulation tasks from the world wide web. In *IEEE International Conference on Robotics and Automation (ICRA)*, 2010.