ICT Call 7
ROBOHOW.COG
FP7-ICT-288533

Deliverable D6.3:

Description of the design and realization of the plan transformation system

January 31st, 2015

Project acronym:	ROBOHOW.COG
Project full title:	Web-enabled and Experience-based Cognitive Robots that Learn Complex Everyday Manipulation Tasks
Work Package:	WP 6
Document number:	D6.3
Document title:	Description of the design and realization of the plan transformation system
Version:	1.0
Delivery date:	January 31st, 2015
Nature:	Report
Dissemination level:	Restricted (RE)
Authors:	Jan Winkler (UNIHB) Michael Beetz (UNIHB)

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

Summary

WP6 is to design, implement, and empirically analyze plan transformation systems that enable a cognitive agent to change its behavior, based on prior experiences, forseen potential problems, and thus to forestall predicted behavior flaws. In order to detect and analyze action effects and their correlation to the overall robot performance, the complete state of an agent and its known environment are recorded. This is done via the generation of episodic memories, which consist of the complete internal state of an agent, and external events. The internal state consists of all performed plans and their parameters, triggered failures, joint states, and reasoning steps during decision making processes. Agent-external knowledge is the environment knowledge, camera streams, and signals triggered by external components. To effectively transform plans based on generic rules, generalized plans are designed and implemented. Their structure can yield a wide range of robot behavior, based on their parameters and their modular anatomy. Through change of contextual knowledge and parts of their modules, their purpose can be transformed to meet the requirements of specific situations. Using the results of such experience-backed plan transformation, an agent can predict how its performance will be affected in situations it encountered before. It develops an intuition about how certain plan parts behave in situations it experienced earlier, and can predict their outcome, further reducing the amount of undesired consequences. In the third year, we have focussed on three problem instances in which we employ episodic memory recording, and processing mechanisms:

- First, the recording and semantic encoding of episodic robot memories allows to retrace the internal state of an agent, its decision making processes, and the correlation between parameters and outcomes of different abstraction layers in its behavior system. Having this information available in a knowledge processing system acts as a base for plan transformation systems than can thus deduce rules from behavior and its outcome.

- Second, generalized plan structures and language constructs for failure detection and unwinding were developed. Based on a task-, object-, and location-description language, they can be parameterized and executed in a well-defined way. By differentiating static, dynamic, and contextual knowledge, robot behavior can be altered without having to change the plan structures themselves. By also introducing a modular structure in those plans that allow replacement of conceptual components, plan transformation rules can easily exchange behavior details altogether.

- Third, by building up memory data over differently composed generalized plans, a cognitive agent can collect experiences about a wide range of situations and plan-element approaches. This knowledge allows it to deduce correlations between plan parameterizations and outcomes, thus making predictions in live situations possible, triggering plan transformation and reparameterization on the fly.

Recording Comprehensive Episodic Robot Memories from Live Plan Execution When performing tasks, cognitive agents make a wide range of decisions that are processed internally and are insivible from the outside, making retracing of these processes very difficult. By recording comprehensive episodic memories of robot behavior, these processes can be analyzed, and the correlation between plan parameterization and their outcome can be used for review by a human developer, or by machine learning algorithms, aiming at higher plan performance. We have developed a set of tools and concepts that allow the recording of such aspects in robot systems in [Winkler et al., 2014], as attached to this deliverable.

Designing of Generalized Plans that allow context-specific behavior change To allow efficient restructuring and reparameterization of robot behavior models, modular language constructs in such a robotic system must be available that can be treated as parameterizable building blocks. Their behavior therefore must change based on the overall module composition in a robot plan, and their explicit parameterization. This allows a plan transformation system to interact with well-defined behavior models, and to leave house-keeping tasks such as task unwinding or implicit failure detection and recovery to the underlying plan execution system. We adressed the development and implementation of such generalized robot plans in [Winkler and Beetz, 2015a], as attached to this deliverable.

Predicting Task Outcome based on Robot Experiences Measuring the performance of robot plans allows a cognitive agent to judge whether one version of a plan is suited better for a given task than another version. By building complex models of task executions, connecting outcomes to parameterizations on different levels in a robot's behavior model, such an agent develop an intuition of how it will perform in a given situation. The agent improves itself over time by meeting more variants of the same, and of new tasks, being able to trigger reparameterization or plan transformation of the current behavior model if needed. We developed and implemented such a system in [Winkler and Beetz, 2015b], as attached to this deliverable.

Contributed Papers

Papers included in this deliverable are:

- Winkler, J., Tenorth, M., Bozcuoğlu, A., Beetz, M. CRAMm — Memories for Robots Performing Everyday Manipulation Activities. In Advances in Cognitive Systems, Volume 3, 2014
- Winkler, J., Beetz, M. Generalized Plan Design And Entity Description For Autonomous Mobile Manipulation in Open Environments. Currently under review for the 14th International Conference on Autonomous Agents and Multiagent Systems (AAMAS), 2015
- Winkler, J., Beetz, M. Robot Action Plans that Form and Maintain Expectations. Currently under review for the nth International Conference on Robotics and Automation (ICRA), 2015

Advances in Cognitive Systems 3 (2014) **–** Submitted 9/2013; published 5/2014

CRAM_m — Memories for Robots Performing Everyday Manipulation Activities

Jan Winkler WINKLER@CS.UNI-BREMEN.DE
Moritz Tenorth TENORTH@CS.UNI-BREMEN.DE
Asil Kaan Bozcuoğlu ASIL@CS.UNI-BREMEN.DE
Michael Beetz BEETZ@CS.UNI-BREMEN.DE
Institute for Artificial Intelligence, Universität Bremen, 28359 Bremen, Germany

Abstract

Agents that learn from experience can profit immensely from memorizing what they have done, why, how, and what happened. For autonomous robots performing complex manipulation tasks, these memories include low level data, such as perceptual snapshots of relevant scenes that influenced decision making, detailed complex motions the robot performed, and effects of these motions. They also include high level representations of the intended actions and the belief-dependent descisions that led to the chosen course of action. In this paper, we present CRAM_m, a memory management system that records very comprehensive and informative memories without slowing down the operation of the robot. CRAM_m offers a query interface that lets the robot retrieve the kinds of information stated above. This is done using a first-order logical language that provides predicates concerning the beliefs and intentions of the robot, its physical state, perceptual information, and action effects, as well as their relations at different levels of abstraction.

1. Introduction

Consider a robot that is supposed to prepare meals, set the table, clean up, load and unload the dishwasher, and so on. Such activities are commonly called "everyday activities". Anderson (1995) defines an everyday activity as "a) a complex task that is both common and mundane to the agent performing it; b) one about which an agent has a great deal of knowledge, which comes as a result of the activity being common, and is the primary contributor to its mundane nature; and c) one at which adequate or satisficing performance rather than expert or optimal performance is required." In this article, we investigate how robotic agents can be equipped with memories of previous activity episodes in order to build up the "great deal of knowledge" for competently performing everyday activities and to learn from their experience.

We present CRAM_m, a software infrastructure which equips robotic agents with a comprehensive memory of their experiences that allows a-posteriori reasoning, diagnosis and reconstruction of the believed world states at different points in time. The system is an extension of CRAM (Cognitive Robot Abstract Machine), a framework for the implementation of cognition-enabled robot control systems (Beetz, Mösenlechner, & Tenorth, 2010). In the context of CRAM_m, we consider memories to be the information gained from past experience, i.e., from everyday manipulation episodes,

that robotic agents can access and use to improve their future activities (Wood, Baxter, & Belpaeme, 2012). As such, the information content of the memories can be measured in terms of the queries that can be answered based on the information contained in the memory.

CRAM_m enables the robot to answer queries such as: *Which fetch tasks failed because the object could not be found? Which kinds of failures could the "place" sub plan not recover from? Which items in the refrigerator often stand at the same position? Did the robot block its view of the pot with its own arm when it put down the mug?* These questions require the robot to memorize its poses, the images it has taken, its beliefs and intentions, information about objects in the world when executing its plan, and the relations among these pieces of information. A robot capable of answering these questions is a robot that knows what it has done, how, and why, as well as the results of its activities (Brachman, 2002). The ability to answer such queries can aid robots in making better execution-time decisions and revising plans to improve their expected performance.

In cognitive psychology, memories are categorized into short-term (STM) and long-term memory (LTM), where the STM is a small-capacity store that provides the context for accomplishing the current task. The LTM is a high-volume memory that provides comprehensive information for all kinds of tasks. Wood et al. (2012) further categorize artificial memory types along other dimensions, such as procedural versus declarative memories, where the declarative memory is often considered as consciously accessible, and the procedural memory contains compiled or subconscious information. Episodic memories store experienced event information that is temporally and spatially organized and combined with context information. *In this paper, we focus on declarative, episodic, long-term memories for robot manipulation episodes.*

Functionally, memorization can be divided into three distinct processes: encoding, storage, and retrieval. The encoding is concerned with observing the state of plan execution and the data streams that are sent between the different components, and mapping this data into memory structures that allow answering of queries. The storage is concerned with accumulating the encoded data in a long-term memory while not degrading overall system performance. Retrieval is concerned with answering queries using the data stored in the memory.

The contributions of this paper are (1) expressive memory representations that combine symbolic plan events with subsymbolic sensor data, (2) methods for temporal, spatial, diagnostic and causal reasoning that operate on the symbolic and subsymbolic memory structures, and (3) efficient and scalable logging mechanisms that build up these structures during task execution without negatively affecting the robot's performance. We evaluate the system using log data collected on three different tasks (object perception, picking and placing an object, and continuous arm movements) that pose different challenges to the sensor and plan logs. To measure the information contained in the memories, we present a set of queries that cover a range of inference capabilities.

2. Cognition Enabled Robot Control and the CRAM System

Before introducing CRAM_m, we should briefly review CRAM and our robot control systems, along with the consequences and opportunities of artificial memory design. The Plan Language CPL (Beetz, Mösenlechner, & Tenorth, 2010; Beetz, 2000) is a concurrent reactive programming language that provides all the comfort of typical high level programming languages, including a rich

set of control structures that help to make the program robust and flexible as well as modular and transparent. The control program takes control decisions based on (possibly complex) inference processes. To this end, control decisions are often formulated as logical queries that evaluate to *true* or *false* or that compute values for parameterizing actions. To execute a task on the robot, the plans activate, parameterize, and deactivate modules of the robot's distributed control system that provide different kinds of functionality such as object perception, robot navigation and localization, etc.

An important aspect of the language are descriptions of entities such as objects, motion, grasps, or poses, that are called *designators* and that are first-class elements of the language. In the beginning of plan execution, these descriptions are often vague, such as "the cup on the table", leaving out situational context or detailed properties of what exactly is described. Plans are parameterized using such vague information and the system refines it when necessary. This way, plans can have qualitative parameters that allow much flexibility in how the task is executed, and that are only quantified when the information they provide is really necessary for execution. Designators are refined whenever more information becomes available, as when an object has been perceived. Since this information is not necessarily correct nor complete, designators can be revised with newer, more correct information as the result of reasoning processes or failures. When a designator is extended with further information, a new designator is created, holding the new, possibly more specific, description. Those two stages of description are then *equated*, i.e., linked, to track the change of parameterization over time. The robot's current *belief* about the world is described in terms of such designators. Especially the poses of objects in the environment and the robot's own position are described in this way.

The execution of a plan generates a tree of tasks, which are interpretation records of subplans very similar to stack frames in program execution. When robot agents are assigned goals to achieve during plan execution, they must perform one or more tasks in order to do so. An arc from task t to task t_{sub} roughly denotes that t called t_{sub} as a subplan. The data structures of the tasks include the local variables and their time-stamped value changes; McDermott (1993) provides a detailed description of this mechanism. Using these data structures, we can define what the robot "believes". For example, we can specify as logical rules that the robot intends to pick up a blue object if the plan parameter for the object acted on has an object description as its value where the color attribute has the value "blue".

3. An Overview of CRAM$_{\mathrm{m}}$

This plan language imposes requirements on the memory apparatus to be provided by CRAM$_{\mathrm{m}}$. This must remember the relationship between plans and their subplans, be able to reconstruct how a particular entity description looked like at a given state of plan execution, and be able to reconstruct why the robot made a particular decision during plan execution. To enable such functionality, robots using CRAM$_{\mathrm{m}}$ record

1. their symbolic beliefs and intentions, the intended course of action (i.e., the task tree of the plan execution);
2. events and data from the perception system, and lower-level information like their position in the environment and their poses;

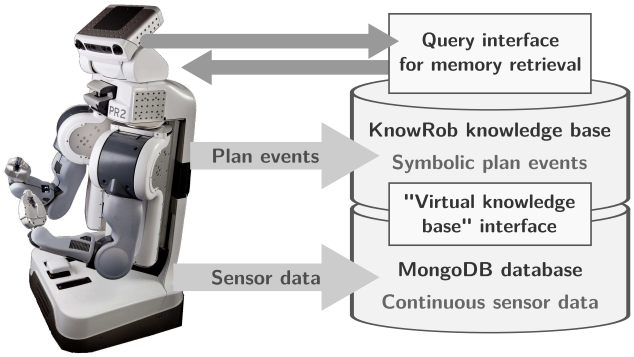


Figure 1: The system architecture for recording live robot data includes logging mechanisms for continuous sensor measures into a MongoDB database and symbolic plan events into a KNOWROB knowledge base. A virtual knowledge base interface integrates continuous data with the symbolic knowledge base. A specialized query interface reasons on the stored information.

3. the relations between them, including the temporal synchronization using global time stamps, how sensor data and changes in data structures cause changes in the beliefs and intentions of the robot, and how the interpretation of plans causes changes in the world.

This information, coming from different sources, is combined to a timeline of events in the robot's knowledge base that allows ontological, teleological, causal and temporal reasoning. Figure 1 depicts the main components of the system which will be explained in more detail in the following sections. During task execution, the robot records comprehensive action logs. Symbolic plan events are directly asserted to the knowledge base (Section 5.1), including the task tree of the robot's control program, described as instances of the respective action classes, information about start and end times, references to manipulated objects, and success and failure states. Continuous measures like sensor data or the frequently updated robot pose are stored in an efficient and scalable database that supports high-volume and high-frequency data recording without slowing down the robot (Section 5.2). Both data sources are described using the same representation language as explained in Section 4. The sensor data is integrated as a "virtual knowledge base" that provides an abstract query interface similar to the rest of the knowledge base. The representation forms the basis for sophisticated inference methods that can help the robot to take control decisions or diagnose plan failures, as described in Section 6.

The CRAM_m system builds upon the functionality of existing components like the CRAM executive, the KNOWROB knowledge base, the robot self model in the SRDL language (Kunze, Roehm, & Beetz, 2011a), and a tool for logging sensor data into a database. In this work, we have integrated these components and have added new modules for logging high level plan events and designators from the CRAM executive, for accessing the logged sensor data from the KNOWROB knowledge base, for computing spatial transformations based on the logged data, and for reasoning about the combination of all this information.

4. Formal Representation of Experiences

The representation of logged actions builds upon the action ontology of the KNOWROB robot knowledge processing system (Tenorth & Beetz, 2013) which provides structures to represent tasks as well as their spatial and temporal context, including events, objects, environment maps, and robot components. KNOWROB is implemented in PROLOG and represents knowledge using the Web Ontology Language OWL (W3C, 2009). As mentioned earlier, our memory consists of a knowledge base of logged plan events (stored in terms of OWL statements in the KNOWROB knowledge base) and a large-volume database with continuously-valued sensor data. To integrate them in a coherent representation that the robot can reason about, we use a special feature of the KNOWROB system that allows the definition of "virtual knowledge bases" on top of sub-symbolic data. Conceptually and from a query point of view, they appear like any other information stored in the knowledge base. However, instead of storing the information in preprocessed symbolic form, it is extracted on demand at query time from the stored data. This has several advantages: The same data can be used to compute different relations that do not have to be selected at recording time, the extracted symbols are inherently grounded, and the large-volume data can be recorded and stored using optimized databases, processing only what is needed to answer a query.

The KNOWROB ontology provides a conceptualization of the robotics domain, as well as formalized background knowledge about the relation between actions, agents, and goals. For example, the "action" branch of the ontology contains about 130 action classes that form the building blocks for describing robot tasks. In addition to existing classes in the ontology that focus on the robot's behavior in the outer world, we have added classes for describing control structures during task execution in order to be able to also reason about these aspects. The memory consists of assertions about occurrences of actions, represented as instances of these action classes, and assertions about the task context. The transitive *subAction* predicate links actions in the task hierarchy; references to objects and locations can be described using properties from KNOWROB such as *objectActedOn*, *fromLocation*, and *toLocation*. Due to the class–instance relationship between the robot's plans and its logged experiences, it is very easy to retrieve examples of previous executions of an action from the memory.

Actions are represented as special kinds of events initiated by agents to achieve a desired effect. This makes it possible to describe these endogenous events using the same structures as exogenous events like sensor readings or utterances of a dialog partner. Figure 2 visualizes the representation of actions and external events using a pick-up task as example. The overall task *PickingUpAnObject* had the goal to bring object *Cup93* into the robot's gripper. This task started at time point T_1 and ended at time point T_{12}. Intermediate subtasks for perceiving, reaching, grasping, and lifting the object are described as *subAction* within the task tree and are therefore directly associated with the overall goal. Each event is characterized by its *startTime* and, if its duration is finite, its *endTime*. The KNOWROB system provides methods for reasoning about the timelines using Allen's interval algebra (Allen, 1983). Events that are produced by other components of the robot's distributed control system are usually not synchronized (e.g., the exogenous events in the lower part of Figure 2), but can be associated with the logged actions using temporal reasoning on the time stamps.

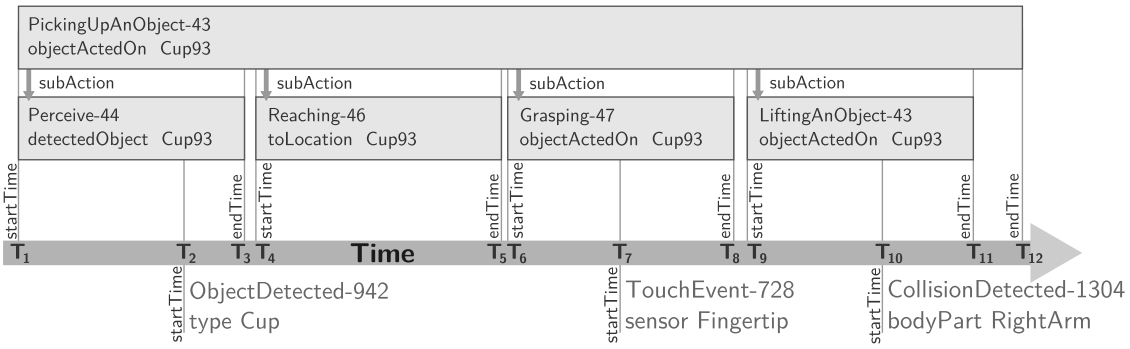


Figure 2: Example timeline of events for a pick-up, task including its subtasks and a few external, instantaneous events. Temporal relations can be computed based on the start and end times of the actions.

5. Encoding and Storage of Memory Contents

We distinguish between symbolic plan events and continuous-valued sensor signals, which are logged using different mechanisms. Section 6 explains how information from both kinds of storage structures can be retrieved and combined in queries.

5.1 Logging Plan Events

As a modern robot plan language, the CPL supports splitting complex goals into subgoals and plan primitives. CRAM provides mechanisms for defining goals, implementing the reasoning processes necessary for their parameterization, and ultimately performing these parameterized tasks. High level goals correspond to the *intentions* of the current plan execution while the subactions executed to achieve these goal reflect the progress and the dynamically inferred parameterization of the task at hand. This approach allows the distinction between different contexts in which each component is executed, for example which goals are currently active at different levels of the hierarchy.

CRAM_m records the task tree including the task parameterizations, failures that arose during execution, the start and end times, success states of single subgoals, and the reported progress feedback from intermediate tasks. In addition, it stores when a designator is created (e.g., an object's occurrence in the world is first mentioned) and when its information is updated, resulting in a change of belief about the world. This information is stored in the OWL representation language in the knowledge base.

Figure 3 shows a simplified example of a *perceive and pick* action, depicting several hierarchically connected tasks. The original task tree comprises roughly 250 actions and events, and 34 designators at 44 different time points, which we have pruned to improve readability. The top-level task to achieve the `object-in-hand` goal is decomposed into a task for perceiving the object and another one for actually grasping it. Task parameters are described by designators, as well as the perception results. The white box in the upper left visualizes the contents of the designator describing the detection of an object of type `CONTAINER`. This symbolic log is linked to sensor data recorded during the task, for example camera images, which are stored at important times during the task execution, or the robot's pose.

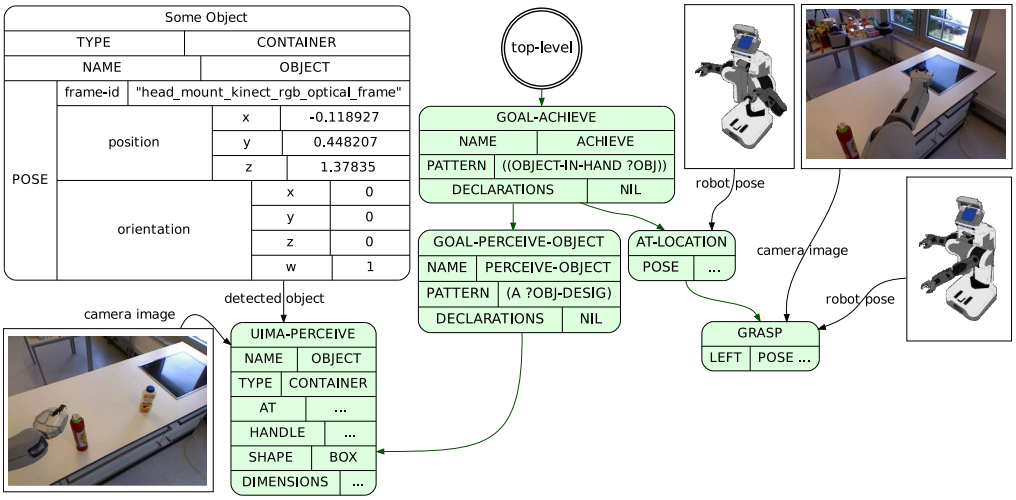


Figure 3: Simplified plan event log for an `object-in-hand` goal achievement. A perception algorithm is employed to find the correct pose for the object in question, the robot agent navigates towards that pose, and grasps the object.

5.2 Logging Sensor and Robot Pose Data

The abstract information from the plan logs is complemented with recorded data from sensors, information about the robot's position in the environment, its pose, etc., in order to be able to reconstruct the world from the viewpoint of the robot as accurately as possible at a later point in time. This can lead to quite a significant amount of data that needs to be recorded without slowing down the task execution.

Our robots are running the ROS communication middle ware (Quigley et al., 2009) in which sensor data and robot pose information are broadcast on so-called "topics" – an asynchronous communication channel that other components (such as the logger) can listen to. This gives the logger access to virtually all pieces of information that are sent around in the robot's system. For recording this information, we use a modified version of the *mongodb_log* software (Niemueller, Lakemeyer, & Srinivasa, 2012) that stores the data in a MongoDB database. While this "NoSQL" database does not support sophisticated SQL queries, it is a fast and scalable storage solution that allows recording robot data with little overhead.

The extensions developed for this logging software include an interface for logging designator communication between different components, as well as methods for limiting the amount of data that is recorded. The former enables exact reconstruction of the high level communication between the plan execution system and for example the perception system (requests, results), storing the designators as nested key-value lists in the MongoDB database. The second kind of extensions is necessary to keep the log databases in a manageable size and consist of different methods. First, sensor data like images are only stored for particular points in time, such as the beginning and end of a grasping action. Point clouds are stored as depth images, which contain the same information but consume much less memory. In addition, the *tf* transformations, which represent positions of objects in the world and especially the position and orientation of every joint of the robot, are only

Table 1: Predicates for reasoning about the memorized experiences.

Meta-Predicates (belief state or ground truth)	
$holds(occ, T_i)$	Occasions in the real world
$belief\_at(event, T_i)$	Occasions in the belief state
$occurs(event, T_i)$	Events in the belief state
Reasoning about the logged task tree	
$task(Task)$	Tasks on interpretation stack
$task\_goal(Task, Goal)$	Goal of task
$task\_start(task, T)$	Start time of task
$task\_end(Task, T)$	End time of task
$task\_status(Task, Status)$	Status of task (not started, ongoing or finalized)
$subtask(Task, Subtask)$	Task is a parent of Subtask
$subtask^{+}(Task, Subtask)$	Task is an ancestor of Subtask
$returned\_value(Task, Result)$	Result of task (success or fail)
$failure\_task(Error, Class)$	Failure of task
$failure\_class(Error, Class)$	Class of failures
$failure\_attrib(Err, Name, Val)$	Attribute of failure
Reasoning about events	
$loc\_change(Obj)$	Object changed its location
$object\_perceived(Obj)$	Object has been perceived

Reasoning about occasions	
$loc(obj, Loc)$	Location of an object
$object\_visible(Obj)$	Object is visible to the robot
$object\_placed\_at(Obj, loc)$	Object was placed at location
Reasoning about logged poses and designators	
$desig\_type(Desig, Type)$	Type of designator
$desig\_prop(Desig, Prop, Val)$	Property values of designator
$obj\_pose\_by\_desig(Obj, Pose)$	Object pose from perceived designator
$lookup\_transform(F_s, F_t, T, Tr)$	Logged transform *Tr* from F_s to F_t at time T
$transform\_pose(P_i, F_s, F_t, T, P_o)$	Transform P_i from frame F_s to frame F_t at time T
$visible\_in\_cam(Obj, Cam, T)$	At time *T*, *Obj* was in the field of view of *Cam*
$blocked\_by\_in\_cam(O, B, C, T)$	At time *T*, *B* was blocking the view of *C* on *O*
$robot\_pose\_at\_time(R, Fr, T, P)$	At time *T*, robot *R* had pose *P* in coordinate frame *Fr*
Visualization	
$add\_object(Obj)$	Visualizes an object in 3D
$add\_object\_with\_children(Obj)$	Also visualizes all parts of Obj
$highlight\_object(Obj)$	Highlight an object in the scene
$add\_trajectory(Link, St, End)$	Show trajectory of $Link$ between times St and End
$add\_diagram(Type, [DataRanges])$	Add data ranges to a diagram

logged when the data has changed. These transformations are updated very frequently (at around 30-40 Hz), which is needed for motion control, but not necessarily to reconstruct the approximate motions from the log files. We therefore introduce a threshold and only store transformations which have changed more than this value with respect to the previously logged version. This reduces the resulting *tf* file size from around 200 MB to around 30 MB for a regular pick and place task since only actual movement data is recorded. The thresholds have been chosen as $0.005m$ Euclidean and $0.005rad$ angular distance. In addition, we log each transformation at least once a second to facilitate the retrieval of the last transformation before a given time point from the database.

6. Retrieval of Information from Stored Memory Data

The vaguely structured plan event logs recorded in the memory can hold substantial amounts of information about the tasks and the events that happened during their execution. Taking a seemingly simple *pick and place* task as example, questions such as *"How long did the pick and place task take?"* and *"How many tries did the agent need to find a suitable pose to stand at when grasping?"* become answerable.

These queries can be formulated using the predicates listed in Table 1. The first set of *meta-predicates* is used to ask for information at a given time and to distinguish between the robot's uncertain belief and ground truth data about the state of the world. While all information in the memory originates from sensor data, some is much more reliable than others. The proprioceptive sensors measuring the robot's joint angles and thus producing information about its pose, for example, are very accurate and reliable and are thus considered as ground truth. Visual object recognition

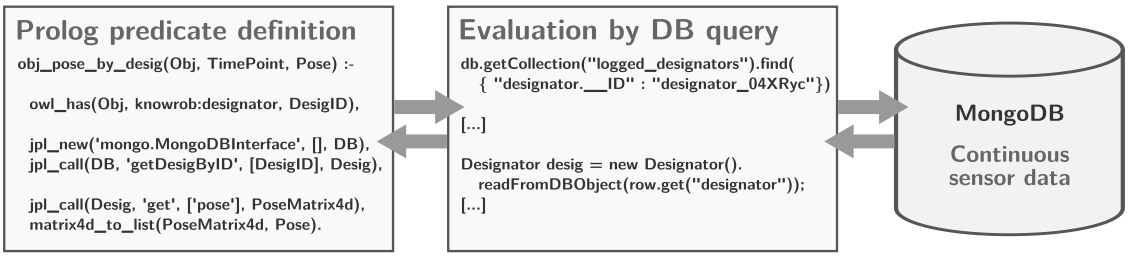


Figure 4: Simplified illustration of the implementation of reasoning predicates that are evaluated based on logged perception data. The set of these predicates spans a "virtual knowledge base" over the recorded memory data.

and pose estimation, in contrast, is a comparatively brittle and unreliable source of information that only updates the robot's belief about the world. The other predicates can be used as arguments to the meta-predicates to reason about the logged task tree, recorded events, occasions (similar to situations), designator values and robot poses over time.

The task tree is logged directly to the knowledge base, i.e., the respective predicates can be implemented by normal PROLOG queries. In contrast, the predicates for reasoning about events, occasions, designators and robot pose information are evaluated on the data logged in the MongoDB database. To the user, they span a kind of "virtual knowledge base" that is computed on demand at query time. Figure 4 depicts how the PROLOG predicate *obj_pose_by_desig* computes the pose of an object at a given time based on detections of that object described as designators. The PROLOG implementation reads the designator attached to the object at hand, and calls a Java method using the Java Prolog Interface to read its pose information. This Java method translates the call into a query to the database and returns the results to the PROLOG predicate.

7. Experiments

We have applied these techniques to a *pick-and-place* scenario, featuring a PR2 robot that transports an object from one arbitrary position on a counter to another. The experimental setup involves a cylindrical object being placed on a kitchen counter. The robotic agent is equipped with only the information that the object is somewhere on this counter and that it should pick it up and transport it to a random new position on the counter. The top-level plan structure

```
(let* ((loc-desig (a location '((on Cupboard) (name kitchen_island)))))
       (obj-desig (an object '((type container) (at ,loc-desig))))
  (achieve '(loc ,obj-desig ,loc-desig))))
```

supplies information about the object itself, but leaves out situational data. The `(achieve '(loc ,obj-desig ,loc-desig))` call ultimately starts plan execution, ordering the robotic agent to move the object `obj-desig` from its current location (on the counter) to the new location `loc-desig`, also on the kitchen counter. The designator `loc-desig` used herein has a twofold use. It generally describes all locations on the counter, without stating an explicit pose. It is applied to the current object location, which is somewhere on the kitchen counter, making all explicit poses on the counter valid search regions for this object. Also, it is used as the target location for putting down the object, which in turn makes all (free) poses on the table valid putdown poses during object

placement. At no point, the high level structure `log-desig` is replaced in the high level plan, but rather resolved to actual 6D poses in the lower level modules.

Figure 5 shows images automatically taken during plan execution as part of the plan log. Three situations are depicted – detecting, approaching, and grasping the object in question. Several more situations were encountered in which the agent failed to perceive the object, and had to try out several positions to stand at before being able to grasp or place the object. We elaborate on this scenario using different queries we developed in order to gain knowledge from recorded experimental data. The information acquired this way spans over all kinds of data the robot is recording: plan events, motion and pose data, communication with other components, and images taken. The knowledge resulting from inquiries, such as why certain tasks mostly fail or whether certain standing positions are bad for grasping nearby objects, are key information for enabling cognitive abilities like reflection about how a task is performed.

The set of queries developed serve the purpose of extracting specific kinds of information from the recorded experience data. Besides finding individual and average time requirements for executed tasks, their success state and potential failure reasons can be acquired. Using the mechanisms presented, statistics can be generated about which tasks fail with what probability due to which reason. Also, a more data driven approach for failure interpretation is described, such as the agent not looking at the object to detect, or blocking the view on the object by one of its own body parts.

7.1 Recorded Experimental Data

The experimental evaluation of the presented techniques covers the examination of three distinctly different logging subjects. In a low level sense, we record the motion data of the robot. For this purpose, we let it perform a mundane movement sequence over a long period of time that shows the efficiency of low level data logging and storage. Another type of sensor event to record during many experiments is the image stream from the robot's cameras. To comprehend each respective situation throughout an experimental trial and to be able to post-process imagery from such an experiment, visual evidence from key moments (grasping, navigation, perception) is collected. In order to find a feasible mechanism for this, we conducted a multitude of experiments for table top inspection (i.e. finding all objects on a table) to validate that the data recording mechanism can handle such data streams. The most complex type of data stream to collect is the actual task and parameter description of any high level plan the robot is performing. In order to enable the proposed system to reliably assemble this information, we ran several large pick and place experiments.

The resulting data that is recorded during the execution of a robot plan includes information about performed high level tasks, low level motion, and images taken in key moments. Such key moments are triggered before and after travelling to a new position, before and after grasping an

Figure 5: Camera images taken during execution of a pick and place task.

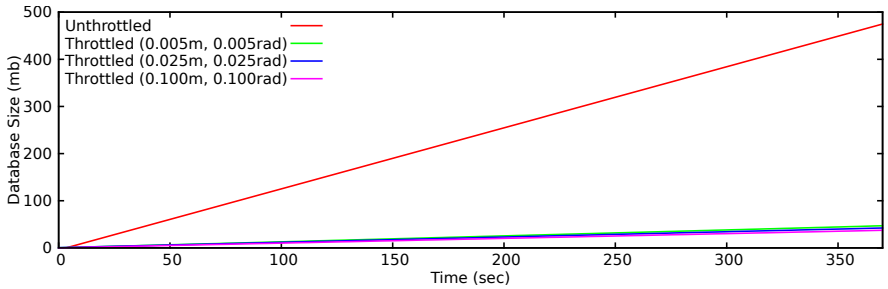


Figure 6: Stored transformation data over time. The different lines represent different throttling thresholds, as shown in the Figure. The experiments were conducted over a timespan of 370s.

object, and when running perception attempts. The images taken during this consist of single `JPG` encoded, compressed image files, which take up around 45kb per file. This keeps the amount of drive space used for visual evidence during the plan logging in reasonable ranges. The symbolic reasoning data recorded during a complete pick and place task as examined in this work, covering symbolic high level reasoning data such as a task description, as well as object, action, and location definitions, sums to about 200–250kb.

The most drive space intense part of the logged experiment data is actual low level motion information (*tf* link transformation data). Figure 6 shows the amount of data recorded for a reference motion. During this motion, the robot moves one arm from one position to another 25 times with different inverse kinematics solutions. This way, differences in inverse kinematic solutions can be neglected and different filter settings for the *tf* throttling can be compared. The figure shows that throttling greatly decreases the data to store, at the cost of accuracy. During the pick and place tasks, a threshold of $0.005m$ and $0.005rad$ was used. These thresholds still allow for qualitative reasoning, as noise of perception systems and the robot base localization introduce similar uncertainties. Therefore, the slightly lower accuracy can be regarded as negligible in favor of a smaller storage size.

7.2 Queries on the Recorded Data

We assess the performance of the system by the range and diversity of queries it is able to answer based on the memorized information. The following queries are exemplary for different kinds of reasoning problems that occur when reasoning about logged execution data: Durations of tasks, types and probabilities of failures to occur, spatial reasoning to compute relations between objects and between objects and the robot's pose at different points in time, as well as the use of these inferences for diagnostic purposes. For being able to answer these queries, the system has to combine information from the high level task tree, low level data like the robot's pose over time, detected objects, and background knowledge like the robot's self model. While some of these queries will directly be integrated into the robot's decision making procedures, their main purpose is to retrieve training data and annotations for learning statistical models of the robot's plans and its performance in different situations.

7.2.1 How Long Does a Perception Task Take on Average?

The average duration of certain tasks can be important when analyzing time requirements of plans. For example, the following query returns the average time needed for a perception action by counting how many tasks with the goal OBJECT-IN-HAND ?OBJ have existed and how many seconds each of these tasks took:

```
?- bagof(Dur, (task_goal(Tsk, 'GOAL-PERCEIVE-OBJECT'),
                task_start(Tsk, StT),
                task_end(Tsk, EndT),
                Dur is EndT - StT), Durs),
    sumlist(Durs, Sum),
    length(Durs, Num),
    Avg is Sum / Num.

Durs = [7,7,9,7,9,7], Sum = 46, Num = 6, Avg = 7.6667.
```

Based on the logged experience data, a perception call takes about *Avg = 7.7* seconds on average.

7.2.2 Which Tasks Failed Due to an Undetected Object?

The robot can investigate which tasks have failed due to *ObjectNotFound* failures using the query:

```
?- task(Task),
   failure_class(Error, kr:'ObjectNotFound'),
   failure_task(Error, Task).

Task = log:'node_E3dONaOC',
Error = log:'node_E3dONaOC_failure_0'.
```

which can be answered based on the recorded task tree. CRAM_m can also return all images captured by the robot in the context of perception tasks that did not detect matching objects (return value `nil`). These images serve programmers as diagnostic material or, in an autonomous learning context, can be used by a robot to test alternative perception methods offline.

7.2.3 How Likely does a Task Class Fail in given ways?

Using the logged memories, robots can compute success probabilities for their tasks. The probability that a task fails due to a given reason can be computed by the number of failed tasks divided by the total number of tasks of this kind. This probability can help to model the expected behavior of the plans and to determine whether refinements are necessary. For example the query:

```
?- bagof(Err, (task_class(Task, kr:'ResolveActionDesignator'),
                failure_class(Err, kr:'ManipulationPoseUnreachable'),
                failure_task(Err, Task)), Errors),
    length(Errors, NumErr),

    bagof(Task, task_class(Task, kr:'ResolveActionDesignator'), Tasks),
    length(Tasks, NumT),

    Probability_of_failure is NumErr/NumT.

NumErr = 2, NumT = 36, Probability_of_failure = 0.0556.
```

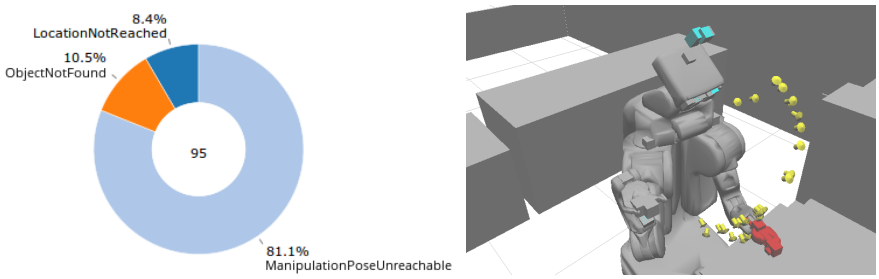


Figure 7: Left: Distribution of common failure types that occurred during task execution. Right: Pose of the robot during a grasping action that has been reconstructed from the recorded log files. The yellow arrows show the trajectory for reaching towards the object, the components highlighted in red denote the robot part used for this action, those in blue are the robot's cameras as read from the SRDL robot model.

computes the percentage of *ResolveActionDesignator* tasks that failed due to an unreachable manipulation pose. Dividing the number of failed tasks (two) by the overall number of tasks of this kind (36), this leads to a probability of `Probability_of_failure = 0.0556`.

7.2.4 What are the Probabilities of Common Failure Types?

By considering log files of several activities, we can compute statistics of common error types that occur during task execution. Consider a query that reads all possible failure classes (which are subclasses of *CRAMFailure* and computes how many of these occurred:

```
?- findall(Type-Num, (owl_subclass_of(Type,'CRAMFailure'),
                      findall(F, failure_class(F,Type),Fs),
                      length(Fs,Num)), Distrib),
   pairs_keys_values(Distrib,Types,Nums),
   add_diagram('Failure distribution','piechart',[[Types,Nums]]).
```

The result of this query is depicted in Figure 7 (left). As can be seen, the *ManipulationPoseUnreachable* failure dominates, which in turn means that improving the motion planning and navigation modules promises the biggest impact on the overall performance.

7.2.5 What was the Pose and the Gripper Trajectory for Grasping an Object?

By combining the symbolic plan logs with the logged geometric information, it is possible to reconstruct the three-dimensional environment including the environment map, the robot pose and trajectories of motions during the task. The following query reads this information and displays the results as in Figure 7 (right):

```
?- task_goal(T,'GRASP'),
   task_start(T,St),
   task_end(T,End),

   robot_pose_at_time('PR2','/map',St,Pose),
   add_object_with_children('PR2'),

   arm_used_for_manipulation(T, Link),
   highlight_object(Link),
   add_trajectory(Link, St, End).
```

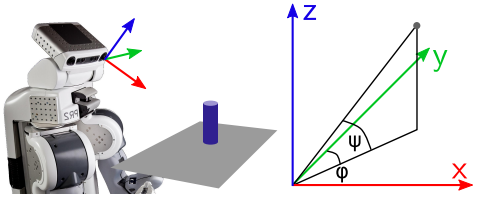


Figure 8: Left: Outside view of the scene with robot camera coordinate frame. Right: In camera-local coordinates, the computation of the bearing towards the objects can be decomposed into two two-dimensional problems.

By visually reconstructing a situation, humans can often analyze problems quickly and estimate why the tasks performed as they did. In the following queries, we will show how the logged 3D geometrical information can also be used for computations.

7.2.6 Which Objects were on the Countertop at a given Time?

Remembering which objects were at a certain location in the past can save robots from carrying out additional perception tasks. An example is a query about what objects the robot believed to be on the table before it tried to grasp an object:

```
?- task_goal(T, achieve('(OBJECT-IN-HAND ?OBJ)')),
   task_start(T, S),
   belief_at(loc(O,L), S),
   on_Physical(O, kr:'CounterTop208').
T = log:'CRAMAchieve_4aOJNJBZ', L = kr:'RotationMatrix3D_vUXiHMJy',
O = log:'VisualPerception_WbrSG11j_object_0', S = 1378119171.
```

The result of such a query is obtained from which objects were in the belief state at that time instance. Moreover, the last predicate checks whether this object is on top of the counter by checking the location of the island with the semantic map of the environment. This query integrates the recorded designators (perception results), the symbolic task tree, and prior knowledge from the robot's environment model.

7.2.7 Did the Camera Face the Object to be Detected?

If an object cannot be detected, it may be that the robot did not look at the right location. If a subsequent detection succeeds, we can analyze if this was the problem by computing whether the position of the object was in the robot camera's field of view before. To compute what the camera was looking at during some point in time, we need to know where the camera is positioned and how large its field of view is.

The former information can be obtained from the recorded robot pose data. In the context of the ROS robot software system, the *tf* library facilitates the management of 3D coordinates by offering methods for transforming any pose into any coordinate frame at a given time. While the original *tf* only keeps data from the past ten seconds, we have extended the system to operate on the full memory of poses such that it allows arbitrary transformations between all coordinate frames at all times for which data is available. The latter information can be obtained from the robot model in the Semantic Robot Description Language (Kunze, Roehm, & Beetz, 2011b), which describes the

geometry of robot parts, their kinematic structure and, for special components like sensors, semantic properties like their resolution or field of view of a camera.

Being able to transform poses into other coordinate frames at arbitrary times makes the problem of computing the camera's view very simple. Using the logged pose data, we can transform the object pose, which is stored with respect to the robot's environment map, into the local camera coordinates (Figure 8). Instead of having to solve a three-dimensional problem, we can now decompose the problem into the computation of the bearing towards the object in horizontal and vertical direction and compare the angle to the camera's field of view:

$$\phi = atan(\frac{y_{obj}}{x_{obj}}) < HFOV \quad , \quad \psi = atan(\frac{z_{obj}}{x_{obj}}) < VFOV \quad .$$

This computation is implemented in the *obj_visible_in_camera* predicate that can be used to ask whether an object was visible for some specific camera at a given time (e.g., the beginning of a perception action), or in which cameras it has been visible.

```
?- task_start(log:'CRAMPerceive_uocvmivw', _St),
   obj_visible_in_camera(log:'VisualPerception_Z9fXhEae_object_0',
                         pr2:pr2_head_mount_kinect_rgb_link, _St).
true.

?- task_start(log:'CRAMPerceive_uocvmivw', _St),
   owl_individual_of(Cam, srdl2comp:'Camera'),
   obj_visible_in_camera(log:'VisualPerception_Z9fXhEae_object_0',
                         Cam, _St).
Cam = pr2:pr2_high_def_frame; Cam = pr2:pr2_head_mount_kinect_ir_link;
Cam = pr2:pr2_head_mount_kinect_rgb_link; [...]
```

The first query computes if the object was in the field of view of the PR2's head-mounted Kinect camera, the second one backtracks over all cameras in the robot model and, for each of them, computes whether the object has been visible in this camera.

7.2.8 Was the View of an Object Blocked by a Robot Part?

A common problem in object manipulation tasks is that the robot cannot see an object because one of its arms is blocking the view. This problem could be avoided by retracting both arms out of the scene, but this is very inefficient. To analyze if a perception failed because a robot part was in the view, we can again use the logged pose and object position data, but instead of computing whether the bearing towards the object is smaller than the camera's field of view, we compute whether the bearings to the object and some robot part are close enough together. This exploits the hierarchical nature of the model by backtracking over all *sub_component*s of the robot's arm and checking each of them to determine if they block the view.

```
?- task_start(log:'CRAMPerceive_uocvmivw', _St),
   sub_component(pr2:pr2_right_arm, Part),
   obj_blocked_by_in_camera(log:'VisualPerception_Z9fXhEae_object_0',
                            Part,
                            pr2:pr2_head_mount_kinect_rgb_link, _St).
Part = pr2:pr2_r_wrist_roll_link; Part = pr2:pr2_r_gripper_palm_link;
Part = pr2:pr2_r_forearm_cam_optical_frame; [...]
```

This query is an approximation of the object's visibility since it neither takes the volume of the robot's arm nor the object into account. We are working on methods for geometrically reconstructing the recorded scenes so that we can apply more sophisticated techniques like off-screen rendering of the scene (Mösenlechner & Beetz, 2013).

These queries demonstrate which kinds of information can be acquired from logged episodic memories. Failures during object perception tasks may be explained by robot parts blocking the view. Objects may be found more easily when past experiences about common places to find this kind of object are taken into account. Common problems during execution of specific plans can be anticipated, and potentially avoided altogether, such as not trying to enter a certain region of a known room as it is difficult to navigate and causes plan performance to drop. By recording this information in the episodic memories and making it available to the control system by the described queries, this experience knowledge can be used for writing robot plans that improve over time.

8. Related Work

Episodic memories similar to the ones recorded by CRAM_m have been investigated in the area of cognitive architectures, though often with a focus on modeling human cognitive processes rather than implementing a scalable architecture for robots systems. One of the earlier cognitive architecture that mimics humans' working memory is Soar (Laird, Newell, & Rosenbloom, 1987). Soar's working memory contains procedural, declarative and episodic knowledge. Namely, it contextualizes a *context stack*, which specifies active goals, problem spaces, states and operators of the embodied agent, *objects*, which are denoted with attributes called *values*, and preferences, which give the procedural search-control knowledge.

ACT-R (Anderson et al., 2004) is another cognitive framework that is built upon a memory concept. In contrast to Soar, it has two different memories for declarative and procedural knowledge, which contain facts and things, respectively. It was adapted for different cognitive applications such as choosing among the competing associations of a concept (Anderson & Reder, 1999), *list memory paradigm* (Anderson et al., 1998), and creating a memory based on the theory of *serial memory* in psychology (Anderson & Matessa, 1997).

ICARUS (Langley, Choi, & Rogers, 2009), an integrated cognitive architecture for physical agents, has two different memory hierarchies. On the one hand, the conceptual memory contains knowledge about general features of things and their relationships. On the other hand, the skill memory stores knowledge about how to accomplish goals. Each of these hierarchies has a long-term memory and a short-term memory.

In the context of robotics, a memory system needs to consider the properties of physical robotic agents, scalability issues due to storage constraints, and processing speeds of the memorized data. Prior work in this field was done by Beetz (2000) using a simulated robot in a simpler environment performing navigational tasks. Our work contributes by extending the domain of application to *mobile manipulation*, which covers much more complex manipulative and perceptive tasks, and by applying the principles to an actual, real robot.

The mechanisms shown are deeply anchored in the robot's control system and can handle high volume, low level data without disturbing the plan execution. For such low level logging,

Niemueller et al. (2012) have presented a comprehensive, dynamic logging system for low level sensor signals into a MongoDB database. We build upon this work, having extended it with methods for logging plan events and designators, and have integrated it with our knowledge processing system to allow semantic reasoning about the data. To keep the amount of logged data to a manageable size, we implemented techniques for throttling high-frequency data like the stream of robot pose information before recording.

Hilbert and Redmiles (2000) describe the benefits of event logging and event stream transformation into streams of interest by selecting, abstracting, and storing them according to current requirements. They make use of this technique to summarize sequences of actions into tasks and to characterize sequences based on probability matrices.

As Coad (1992) points out, such an *"Event Logging Pattern"* consists of a *"device"* triggering an event remembering message which adds a certain sensor event to a database of events with historical values when surpassing a given threshold value. In our case, these messages might be generated when a plan event *starts* and when it *ends*, putting everything in between into its context. Our assumption is that a certain context is *active* as long as it is not revoked by an active trigger or by the absense of a previously active trigger signal. In the case of plan logging, a task context is started at the beginning of its subroutine and ends when the subroutine is left again. Subtasks of this task may show the same behavior, making them hierarchical children.

On the basis of such high level information, Brachman (2002) describes the necessity of systems that can reflect on their current task and their own performance. Benefits gained from more reflective systems would be the ability to take a step back from the current situation and getting out of a mental box, but also to be able to explain why a certain task is being performed in the way it is done. Our proposed approach aims at gaining this kind of knowledge from observing the (internal) state of the agent and the surrounding environment, and thus is able to reconstruct any situation during the performance, as well as build up a causal connection between events and their consequences.

With such knowledge at hand, Kaelbling and Lozano-Pérez (2013) elaborated on having a complete belief state available for replanning and reasoning purposes. They intend to harness this information as basis for dynamic decision making. Also taking a robot's possible courses of actions into account, their system plans ahead based on the current situation in order to get an impression about the nature of future situations. The process they perform in a live scenario relies on quick processing times and on inexpensive computation mechanisms to not slow down the ongoing task. We build upon this principle by making a complete belief state available ex post. This enables our approach to use more computationally expensive algorithms on memorized episodes and generate more insights about the respective situation, performing a-posteriori reasoning about the characteristics of the robot behavior.

9. Discussion and Conclusions

In this paper, we presented a comprehensive memory system for cognitive agents acting in the real world. Within this context, we made efforts to clearly distinguish between the agent's current *belief* about the state of the world and the *actual* state, as well as the robot's *intentions* that led to this state. When performing an action, the agent expects a certain outcome – this makes up its *believed*

world state. Sensor readings, such as camera images, may provide information that contradicts its beliefs, and can yield knowledge about how well a task was performed and even what went wrong when comparing the expected and the actual outcomes. Taking the intentions of the current task into account, the agent can answer questions about *why* it performed a certain task in a certain way, and it can store information about possible failures and common pitfalls during this type of action, making improvement of future executions of the same or similar tasks feasible.

The memory of the robotic agent is filled from two streams of events. The first, being of low volume, consists of the symbolic hierarchical task structure of executed plans. This data also includes qualitative parameterizations of tasks described by designators, allowing for logical reasoning. These designators can be extended over time and made more precise when new information becomes available or old information gets retracted or replaced. The second stream holds quantitative data from the robot's sensors, including camera images and the robot pose. Both streams are synchronized using time stamps of the start and end times of plan events. This approach allows to reason about what information was gained through which measures. Perception tasks that have limited *a priori* information about the objects they are looking for yield their requesting and resulting designators. Comparing both may conclude that an object on a table is at a specific 3D coordinate, changing vague information into more specific details.

The proposed representation forms the basis for the definition of higher-level concepts like which action *causes* which effects and what the current *beliefs* are. For example, taking an arm movement of the robot into account, this action might be signaled by a plan event *VoluntaryBodyMovement* as it is part of a grasping action. On the basis of this high level concept, low level data about the robot's pose and the actual reaching motion of the arm can now be connected and reasoned about. Assuming all movements to be connected to the current plan event is sufficient here as the plan triggering the motion takes exclusive control over the arm through a semaphore mechanism. Based on the agent's belief, information about real-world entities is available to internal reasoning processes through the designator representation as well. Taking two designators denoting objects in the real world, their positions might be concluded to be near each other as the designators indicate physical proximity – on a quantitative level, e.g., being near to each other, or on a qualitative level, residing on the same supporting surface.

We presented a comprehensive robot memory system, featuring an extensive encoding scheme for symbolic and subsymbolic experience data collected from real-world robot plan executions. An integrated storage approach for both kinds of data was introduced and logical queries for information retrieval from this memory were developed to make use of the resulting knowledge possible for plan improvement mechanisms. The presented queries up to now picture the conceptual setup of the system, forming the base for more elaborate reasoning mechanisms and query types. Taking more information into account and running exhaustive analysis algorithms on the collected experience data offers much potential when it comes to a-posteriori reasoning and analysis, especially in terms of life long learning concepts. Collecting large amounts of data over many trials can form the basis for substantial improvements during planning and plan execution. Possible results of such learning processes include regions that reflect the utility of robot and object poses for certain tasks, and appropriate failure handling for failed tasks under a specific situational context. Experiences collected by robotic agents in different situations can not only hold information about single task

types currently performed, but open up possibilities to reason about general knowledge that applies to many situations and tasks. We are developing an open source software framework[1] around the presented techniques that implement plan logging capabilities. For sensor data storage, we rely on the *mongodb_log* ROS package, which is available as open source, and the reasoning capabilities shown in our example queries for knowledge acquisition are implemented into the KNOWROB knowledge base system, which is being developed as open source as well.

Based on the presented results, we are working on a number of extensions of, and applications for, the plan logging system. The developed system acts as a framework for supplying robot plan designers with a memory collection and interpretation mechanism to enhance robot behavior. In its current state, there is no comprehensive feedback mechanism to transparently incorporate collected data. We are expanding its capabilities and develop specialized plan structures that transparently base plan decisions on former experience data. To make the collected memories more accessible for humans, we are developing a web interface based on the Robot Web Tools library (Alexander et al., 2012) that provides visualization tools for 3D scenes and for statistical data for past and live episodic memories. To enhance plan performance through log data analysis, we also work on abstracting from plan control flow paths to generate finite automata from executed plans, and therefore allow anticipation of live plan performance based on earlier episodes. In the long term, we want to incorporate more data not only from the CRAM plan system, which currently is the only symbolic high-level data source, but also control decision details from external components such as the perception system and the knowledge base, and extend the developed query library accordingly.

Acknowledgements

This work was supported in part by the DFG Project BayCogRob within the DFG Priority Programme 1527 for Autonomous Learning and the EU FP7 Projects *RoboHow* (Grant Number 288533) and *SAPHARI* (Grant Number 287513).

References

Alexander, B., Hsiao, K., Jenkins, C., Suay, B., & Toris, R. (2012). Robot web tools. *Robotics & Automation Magazine*, *19*, 20–23.

Allen, J. (1983). Maintaining knowledge about temporal intervals. *Communications of the ACM*, *26*, 832–843.

Anderson, J. E. (1995). *Constraint-directed improvisation for everyday activities*. Doctoral dissertation, Department of Computer Science, University of Manitoba, Winnipeg, Manitoba, Canada.

Anderson, J. R., Bothell, D., Byrne, M. D., Douglass, S., Lebiere, C., & Qin, Y. (2004). An integrated theory of the mind. *Psychological Review*, *111*, 1036–1060.

Anderson, J. R., Bothell, D., Lebiere, C., & Matessa, M. (1998). An integrated theory of list memory. *Journal of Memory and Language*, *38*, 341–380.

Anderson, J. R., & Matessa, M. (1997). A production system theory of serial memory. *Psychological Review*, *104*, 728–748.

1. `http://www.github.com/code-iai/planlogging`

Anderson, J. R., & Reder, L. M. (1999). The fan effect: New results and new theories. *Journal of Experimental Psychology: General*, *128*, 186–197.

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

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

Brachman, R. (2002). Systems that know what they're doing. *IEEE Intelligent Systems*, *17*, 67–71.

Coad, P. (1992). Object-oriented patterns. *Communications of the ACM*, *35*, 152–159.

Hilbert, D. M., & Redmiles, D. F. (2000). Extracting usability information from user interface events. *ACM Computing Surveys*, *32*, 384–421.

Kaelbling, L. P., & Lozano-Pérez, T. (2013). Integrated task and motion planning in belief space. *The International Journal of Robotics Research*, *32*, 1194–1227.

Kunze, L., Roehm, T., & Beetz, M. (2011a). Towards semantic robot description languages. *ICRA* (pp. 5589–5595). IEEE.

Kunze, L., Roehm, T., & Beetz, M. (2011b). Towards semantic robot description languages. *Proceedings of the IEEE International Conference on Robotics and Automation, 2011* (pp. 5589–5595). Shanghai, China.

Laird, J. E., Newell, A., & Rosenbloom, P. S. (1987). Soar: An architecture for general intelligence. *Artificial Intelligence*, *33*, 1–64.

Langley, P., Choi, D., & Rogers, S. (2009). Acquisition of hierarchical reactive skills in a unified cognitive architecture. *Cognitive Systems Research*, *10*, 316–332.

McDermott, D. (1993). *A reactive plan language* (Technical Report). Computer Science Dept., Yale University, CT, USA.

Mösenlechner, L., & Beetz, M. (2013). Fast temporal projection using accurate physics-based geometric reasoning. *Proceedings of the IEEE International Conference on Robotics and Automation, 2013*. Karlsruhe, Germany.

Niemueller, T., Lakemeyer, G., & Srinivasa, S. S. (2012). A Generic Robot Database and its Application in Fault Analysis and Performance Evaluation. *Proceedings of the IEEE/RSJ International Conference on Intelligent Robots and Systems, 2012*. Vilamoura, Algarve, Portugal: IEEE/RAS.

Quigley, M., Conley, K., Gerkey, B., Faust, J., Foote, T., Leibs, J., Berger, E., Wheeler, R., & Ng, A. (2009). ROS: an open-source Robot Operating System. *IEEE International Conference on Robotics and Automation, 2009*. Kobe, Japan.

Tenorth, M., & Beetz, M. (2013). KnowRob – A Knowledge Processing Infrastructure for Cognition-enabled Robots. *International Journal of Robotics Research*, *32*, 566–590.

W3C (2009). *OWL 2 Web Ontology Language: Structural specification and functional-style syntax*. World Wide Web Consortium. http://www.w3.org/TR/2009/REC-owl2-syntax-20091027.

Wood, R., Baxter, P., & Belpaeme, T. (2012). A review of long-term memory in natural and synthetic systems. *Adaptive Behavior*, *20*, 81–103.

Generalized Plan Design And Entity Description For Autonomous Mobile Manipulation in Open Environments

Jan Winkler
Institute for Artificial Intelligence
Universität Bremen
Am Fallturm 1, 28359 Bremen, Germany
winkler@cs.uni-bremen.de

Michael Beetz
Institute for Artificial Intelligence
Universität Bremen
Am Fallturm 1, 28359 Bremen, Germany
beetz@cs.uni-bremen.de

ABSTRACT

Autonomous robotic agents acting in open environments have to master situations and action effects they did not anticipate. To deal with these issues we propose an information processing framework for plan interpretation that is based on three concepts:

- Description of actions, tasks, objects, locations, that are vague, incomplete, and ambiguous, and can be refined when needed;
- context-specific instantiation of descriptions using symbolic reasoning; and
- control structures of plans that can monitor plan execution for unexpected events and respond appropriately.

We present design patterns for cognition-enabled reactive robot plans that allow for monitoring specific aspects of plan execution and provide mechanisms for detecting and recovering from unwanted effects. Plans realized according to these design patterns are applied to a reasoning intense human-scale manipulation task in a kitchen environment.

Categories and Subject Descriptors

I.2.9 [**Robotics**]: Autonomous vehicles; I.2.8 [**Problem Solving, Control Methods, and Search**]: Plan execution, formation, and generation; D.3.3 [**Language Constructs and Features**]: Concurrent programming structures

General Terms

Languages, Reliability, Algorithms

Keywords

Robot planning and plan execution; Robotic agent languages and middleware for robot systems; Reasoning in agent-based systems; Mobile agents; Programming languages for agents and multi-agent systems

1. INTRODUCTION

Performing complex everyday activities becomes increasingly comprehensive and popular for robotic agents nowadays. These agents have one thing in common: Their task is well defined and specifically tailored to the scenario they are deployed in. The design of their strategies is mostly purpose driven [1], while more challenging and complex situations [2, 3] would overburden the system with unexpected failures, situationally necessary recovery actions, or simply with ambiguous sensor data [4].

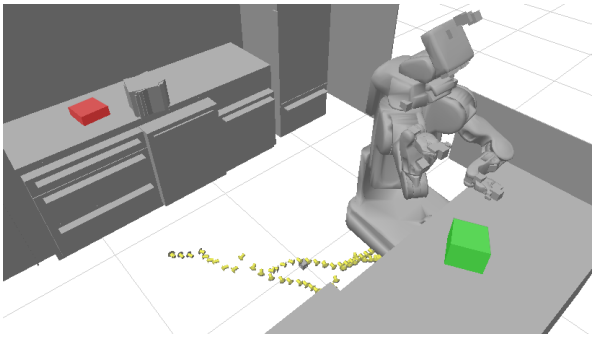


Figure 1: Web Inspection Interface for Log Data Analysis of Live Experiments. The PR2 robot moved an object from one location (red) to a different one (green). The robot's navigation path and its current pose as memorized from the experiment are shown.

Since task necessities change based on the current situation's context, we present plan control structures that take into account contextual information to disambiguate uncertain situations, allowing an agent to make informed rather than heuristical decisions. We define a task description language that can express tasks, objects, and locations at different levels of vagueness, allowing symbolic reasoning to instantiate and ground vague task descriptions specific to the current context.

Therefore, we see a need for making autonomous agents understand for what reason (i.e. in what context) they are to perform a task, and what the nature of its subtasks are. While this should cover knowledge about anticipated outcomes of such tasks, the ability to undo one's doings in order to rewind an action that had unanticipated, unwanted effects plays an important role – and for that matter, perceiving that seemingly innocent effects in a given context are undesired, or even malicious, is key. To enable an agent to notice differences in the expected and actual course of action, we developed plan language structures that concurrently monitor specific aspects of robot plans, and can alter or interrupt the performance of a task when the situational context changes.

Consider a robot agent in a human environment, performing tasks in lieu of humans. Major parts of tasks that such an agent is ordered to do are mobile manipulation tasks, e.g. bringing objects from one place to another, opening and closing containers on the way, and operating its environment to reach its goals. Examples of such tasks range from table setting scenarios and room service applications [5] to supporting elderly people [6], which cannot completely be parameterized a priori, leaving all task action parameters vague

Appears in: *Proceedings of the 14th International Conference on Autonomous Agents and Multiagent Systems (AAMAS 2015), Bordini, Elkind, Weiss, Yolum (eds.), May, 4–8, 2015, Istanbul, Turkey.*

during plan design time.

To competently specify a wide range of task features, action descriptions, and object entities in a vague manner, a potent task description language is required, which can at the same time represent complex behaviour for an autonomous robot agent, and be general enough to leave space for contextual interpretation.

In this work, we address all three of these challenges. By employing new plan language structures, we equip an agent with the ability to performs vaguely described tasks in a context aware manner, while at the same time monitoring its environment for situationally important changes that might interfer with its own goals. Through outcome anticipation during task performance, this agent is able to react to unexpected results, unwinding its current action, and further pursueing its current task. To evaluate the performance of our approach, we use an open source robot experience analysis toolkit [7]. Figure 1 depicts such an agent's experience log, featuring a PR2 robot having transported an object.

In the remainder of this document we proceed as follows. The following section depicts an overview of all conceptual plan design patterns and reasoning schemes underlying our approach, and describes our overall architecture. We describe the role and definition of contextual knowledge in Section 3, discuss concurrent task monitoring in Section 4, and depict a symbolic language for task description in Section 5. Vital plan structures are discussed in Section 6. The presented concepts are applied to human-scale mobile manipulation tasks in Section 7. After evaluating actual robot experiments, we discuss current and future challenges in Section 8. We conclude with related work.

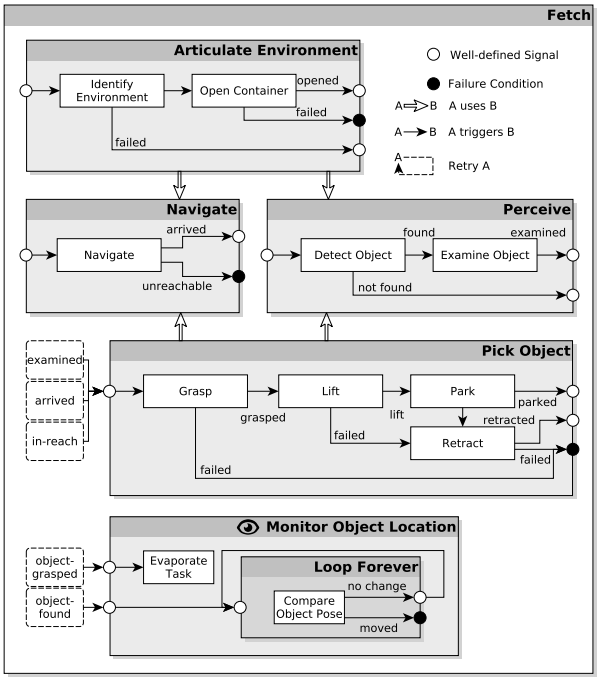


Figure 2: High Level Plan Schematics for a *Fetch* task. It consists of primarily picking objects, and additionally perceiving, navigating, and articulating the environment. Task preconditions for picking objects and for monitoring an object's pose are shown.

2. OVERVIEW

The key to competence in autonomous agents is contextual knowledge. We show examples of applicable situations and develop plan structure elements that allow specification of contextual constraints in robot plans. Contextual knowledge can originate from two sources: *a)* an external knowledge base, specifying static constraints, and *b)* dynamically infered constraints, resulting from the current situation. Using specialized plan structures, we therefore supply three functional elements: *knowledge retrieval*, *knowledge inference*, and *context specification*. These details are addressed in Section 3.

To successfully cope with dynamic environments and unpredicted context changes, we actively pursue the development of concurrent, reactive monitoring and support processes. When an unanticipated event changes the odds of success in the current task, an agent has to adjust its strategy, and possibly undo formerly performed actions. By tracking the changes done, and equipping an agent with knowledge about how to revert them, dynamic environment changes can be addressed in a transparent manner. In Section 4 we introduce plan structure elements allowing reactive interruption of task performance based on *concurrently monitored* task features, and successfully *reacting* to context changes.

When specifying general robot plans that are meant to work in multiple different contexts without changing their task description, a descriptive language for action specification is required. Given the generality of tasks such as *"Get a glass from the kitchen"*, a lot of contextual information is missing – e.g., where the kitchen is, what a glass is, and how to pick up a glass. A task description language that can depict this kind of conceptual idea allows an agent to contextually interprete the meaning of such an order. Section 5 deals with the definition, interpretation, and disambiguation of these vague action descriptions.

3. CONTEXTUAL KNOWLEDGE IN AUTONOMOUS ROBOT AGENTS

Common robot plans are structured in a hierarchical tree, allowing to encapsulate and reuse functionality. Historical robot programming languages, such as STRIPS [8] and Universal Plans [9] are using primitives that can be concatenated based on their predecessors outcome. More recent work, such as RPL (Reactive Programming Language [10]) and HTN (Hierarchical Task Networks [11]) further enhance this approach and allow more complex interaction between tasks. A missing aspect in all these approaches, though, is implicit context awareness. Tasks ought to perform differently not only based on their *explicit parameterization*, but also based on *implicit circumstances*. These circumstances can depend on external factors, but also on knowledge available to plans. We therefore propose structure elements to enhance a plan's context awareness without encoding all possible situations into the plans themselves. By introducing the `with-context` environment

```
with-context
  context-parameters <c1, ..., cn>
  body <body-code>
```

nested plans can define a contextual setting for other *black box* plan building blocks that they are using. Any layer of a hierarchical robot plan can therefore either add new contextual parameters, or alter and overwrite old ones. The behaviour of single plan blocks can thus be influenced by a semantically higher hierarchy layer and be made contextually aware. Internally, contextual information is encoded as sets of PROLOG rules and facts, allowing dynamic addition of new complex constraints. When new contextual information becomes available, and task knowledge is available to the PROLOG reasoning system, arbitrary plans can request specific constraint information such as

```
(context ?context)
(task pick-and-place ?task)
(constraint max-object-tilt ?constraint)
(contextual-constraint
 ?context ?task ?constraint ?angle)
```

resulting in specific, context-dependent information about the requested type of constraint. In the above example, the maximum tilting angle of objects transported during a pick and place task is resolved. A context that describes the transport of liquid filled mugs results in different maximum tilting angles than when transporting empty ones, given the assumption that nothing should be spilled.

4. CONCURRENT MONITORING FOR CONTEXTUAL CHANGES

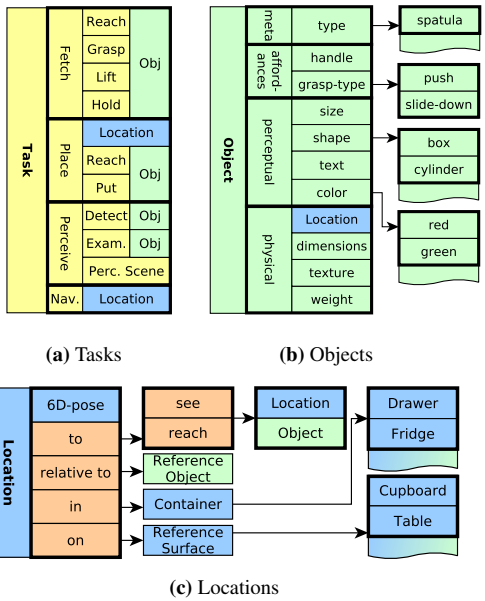
Current command patterns for cognitive agents rely on a fixed sequence of actions to lead to a desired goal when interpreted from natural language [12, 13] or when designed in a plan language [14, 15] and automaton-like structures [16]. To make a cognitive agent not only capable of performing its task as described, but also reactively change its behaviour due to external influences, we see potential in the proper modeling of concurrent support and monitoring tasks that can alter or interrupt the performance of a given task.

An agent that needs to be able to satisfy contextual constraints must be aware of both, the state of its primary task, and the state of the world around it. As most plans for robot behaviour are designed for their sole purpose and don't explicitly consider every possible interference that might occur in an open, real world, a carefully designed agent must be able to judge different contextual constraints with respect to its current actions. We thus reason that an autonomous agent must be able to concurrently monitor the world around it, and have specialized routines that allow it to react to changes in the environment that diverge from its expectance. Such diverging changes must then be able to trigger well-defined signals that inform other tasks about an unusual situation, or the incidence of a predefined situation. Being able to react to a given set of preconditions does not have to require an actively running task process, but can activate a previously inactive, pending skill. An agent might have a whole array of inactive skills at hand, each of them being activated when their knowledge preconditions are fulfilled. Such preconditions might be the position of an object that is being pursued by a perception routine – and once it is found, the newly available knowledge triggers a grasping action. A monitor task, on the other hand, might change the contextual parameters concurrently when a grasped object starts to slip from the robot's gripper, triggering regrasping.

Figure 2 depicts a complex *Object Fetch* task, featuring navigation, perception, and manipulation modules. It also includes a *Monitor Object Location* task, which can monitor the location of an object until it is grasped. A possible reaction triggered by this module would be to adjust the target grasping position in case the object moved unexpectedly.

5. TASK DESCRIPTION LANGUAGE

The human understanding of performable tasks and the amount of information necessary to describe them strongly diverges from the vocabulary today's robotic agents can commonly interprete. This is, in part, due to two prominent reasons. The first being the fact that humans know about certain characteristics, the context, without speaking of them, and the second being that humans find



(a) Tasks **(b)** Objects

(c) Locations

Figure 3: Description language attributes for Tasks, Objects, and Locations for everyday robot activity plans.

Algorithm 1 Sample task description for fetching and placing a cup in a cupboard.

```
1: (an action (to fetch)
2:            (obj (an object (type cup))))
3: (an action (to place)
4:            (obj (an object (in gripper)))
5:            (loc (a location (in cupboard))))
```

out many characteristics on the go while performing tasks, effectively grounding them in the concrete situation. While the former sets the cornerstones of the activity, the latter requires active perception and processing of what is happening, allowing to form new knowledge about the task at hand.

To enable an autonomous agent to formulate both, contextual and grounded task knowledge, it is in need of clearly defined language structures that express such knowledge. To cover common and challenging everyday tasks a robotic agent is due to perform in human environments [2], we map out a versatile – and extensible – set of language attributes in Figure 3.

We divide the entities to be described in three main categories, being tasks, objects, and locations. Tasks primarily have parameters that are necessary for them to do any meaningful work. A fetch and place task for example typically requires specification of an object to fetch, and a destination where to place that object. Assuming that an autonomous agent is sufficiently knowlegable about a task's background story, this can be taken a step further by allowing the agent to deduce necessary parameters itself. The fetch and place task then might become a *"fetch object and put it where it belongs"* task, effectively eliminating the need for a target destination parameter. This is rendered possible by *contextual knowledge* about the task made available to the agent. Given that a valid destination could not be infered, an autonomous agent would then need to enquire about it. To allow such automatic deduction, an agent needs knowledge about what it *expects* to know for performing a task. Language constructs as presented serve as a map for this information. A sample description of a fetch and place task can be specified as shown in Algorithm 1. Besides tasks, locations and objects can

be specified in varying amounts of detail. For both, this introduces a degree of ambiguity: An object might be refered to as *"a red apple located on the kitchen counter next to the toaster"*, but also by *"an object somewhere in the house"*. While in most cases, the former describes a single entity, the latter can describe about anything in a house. Again, contextual information not mentioned in this description can resolve this issue. Kushmerick et al. [17] describe this issue in their *"Bomb and Toilet"* scenario. A robot is supposed to diffuse a bomb hidden in a package by dunking it into a toilet. It is confronted with two situations: One with only one package present, and one with many packages. Contextual knowledge about the nature of bombs and about the task itself allows the agent to deduce that dunking *all* of the packages into the toilet disambiguates the parameter, solving the task. On the other hand, if we were to order the robot to bring a glass from the kitchen, *any* glass – and only *one* instance – would suffice.

Specifying parameters usually omitted when ordering an agent to perform tasks can serve two purposes: Firstly, easing the robot's reasoning and disambiguation process, and secondly, explicitly satisfying constraints. A robot that has two different end-effectors usable for grasping objects, of which one is much stronger than the other, can be told to grasp specific objects only with the weak one to not break them. This choice is not apparent from situations without context, while specifying it makes the agent more competent at what it is doing.

Autonomous agents actively perceiving their environment can *enrich* their knowledge about the task and context themselves while performing it by *examining* their surroundings based on hints from the task description. This task knowledge results from careful observation: Detecting handles on an object and using them can greatly increase the probability of a successful grasp, instead of just pushing the object between the robot's grippers – a coffee cup is a good example for this. Such knowledge can act as *enablers*: Pots too big for a robot's grippers are not graspable at all if no handles are known. We therefore treat an *examine* action as the enrichment of an object description with more information, including sensor data and knowledge from an external database.

Knowledge about tasks and objects does not need to have direct effects on an agent's task parameterization. The correct handling of an object can vary strongly in the wake of seemingly independent context parameters. A spatula must be grasped differently when it is used for flipping a pancake than when putting it into a drawer. A cup's opening may not be obstructed when it is going to be poured from. A glass must be held upright during transport when it contains liquid. These deductions require contextual and task knowledge without which an autonomous agent may not be able to make the right decisions.

5.1 Design-Time Generated Static Knowledge

Spelke [18] argues that *"[a creature knows something about its environment] when [it] systematically draws on what it knows to make inferences about properties of the surroundings that it cannot perceive"*. An agent can be enabled to autonomously find its way around the environment by providing *design-time generated static knowledge*. Besides floor maps or the collision environment, this can imply details about the tasks to perform. As seen in Figure 3a, tasks have a simple signature to leave as much autonomy to the agent as possible. Using static knowledge as task *context*, hints can be asserted about the situation (*"The cups are in the lower drawer today"*, *"The oven is unusable, use the microwave"*), or about how a task should to be performed (*"Don't tilt the full coffee mug when bringing it here"*). The language attributes presented here do not yet provide the complexity of expressing such circumstances, although they are well in reach and we are working on extending the descriptive features of contextual parameterizations to fit these needs.

Algorithm 2 Sampled task description for transporting an object of type spatula onto a table.

```
1: (achieve (loc
2:   (an object (type spatula)
3:              (color black)
4:              (at (a location (in drawer)
5:                              (in kitchen))))
6:   (a location (on table))))
```

5.2 Deducing Constraints

Contextual information is not only supplied by statically user-defined knowledge, but also by dynamically deduced and acquired knowledge. A prominent case is the picking of an object: A perception task acquires the coordinates of an object, supplying a downstream grasping task with its pose. The grasping task gets *dynamic context information* in order to operate correctly. Deducing context from already existing information might require a more fundamental understanding of how object characteristics are linked up. Simplifying the example from above, an agent might deduce from the static knowledge *"The coffee mug is full"* and the understanding that it should not tilt full mugs how to properly bring that mug without spilling the coffee. In order to transport knowledge on *"not tilting full mugs"*, a series of reasoning steps is necessary, which should not be specifically tailored to this one application. Deducing such constraints is therefore not yet part of our overall architecture, but part of our goals.

When deducing task constraints, they can strongly depend on the characteristics of the objects acted on. Objects consist not only of perceptible properties, but also physical characteristics and affordances. The latter describe how an object can or should be handled, what to avoid in operating a given piece of machinery, or how to open a fridge door. Donald Norman [19] refers to affordances as *"the perceived and actual properties of the thing, primarily those [...] the thing could be used [for]"*. When modeling objects, we mainly rely on the affordance of virtual object handles, or *"grasp points"*, that allow a cognitive agent to deduce how to properly grasp or place an object, or open and close a container.

5.3 Sampling from the Description Language

Given a Task Description Language (TDL), task specifications to be performed by an autonomous agent can be sampled and executed. While some specifications make sense, such as shown in Algorithm 2, others might simply not work in the current situation – fetching the green ketchup while there is no such object is bound to fail.

To allow flexible, interesting tasks to be sampled, each property (and each individual value) can be annotated with a value on how probable it is to show up in the resulting specification. This way, two things are achieved: First, an agent can explore the space of possible parameterizations and collect information about what went well and what didn't work, and second, only actually executable tasks are performed, as all vocabulary in the language is known to the agent's plan system. Algorithm 3 shows an example parameterization of the task sampler, resulting in an object that has a 70% chance of featuring the `type` property, with a 20% chance of it being `ketchup`, 50% `pancakemix`, and the rest of type `fork`. The same applies for `color`.

Advancing the task description language beyond atomic tasks results from adding another hierarchy and abstraction layer: By in-

Algorithm 3 Sampler configuration for describing an object with optional type and color.

```
(an object
  (a property (type ((ketchup 0.2)
                     (pancakemix 0.5)
                     fork)) 0.7)
  (a property (color ((red 0.7)
                      (yellow 0.2)
                      blue)) 0.3))
```

Algorithm 4 Failure Handling (simplified) for grasping an object and recovering from a manipulation failure.

```
(defun pick-object (obj)
  (with-side-infered obj side
    (with-failure-handling
        ((manip-fail
           (retract-arm side)))
      (perform-grasp obj side))))

(defun perform-grasp (obj side)
  (unwind-protect
      (unless (motion-grasp obj side)
        (signal-fail manip-fail))
    (stop-motion)))
```

troducing *scenarios*, the TDL can support actions like TABLESETTING, DISHWASHING, or SHOPPING. These more complex actions mostly consist of multiple fetch and place or other manipulation primitives used as building blocks, and parameterized for the scenario's specific purpose.

6. PLAN STRUCTURES

Ingham et al. [20] have developed the model-based programming approach RMPL, that allows constraining of concurrently running monitoring processes while performing an otherwise sequential task. Their conditional task execution continuously monitors a set of local variables for related values and fires upon meeting these requirements. In RMPL, they allow for sequential, parallel, and conditional performance of processes. We take this idea a step further and introduce mechanisms for interrupting plan performance when unusual conditions arise. As each of our (sub)plans is accompanied by a definite exit strategy to rewind its own state changes, each level in the hierarchy only has to cope with undoing its own actions. These failure handling strategies allow to encapsulate implicit recovery mechanisms to properly unwind and evaporate tasks created by complex plan hierarchies. Algorithm 4 shows a simplified failure handling case for an object grasping task. The `perform-grasp` function (lines 8–12) signals a failure of type `manip-fail` unless its grasping motion finished successfully, and stops the motion control in either case. Its parent function, `pick-object` (lines 1–6), catches that failure signal and retracts the infered arm to a safe pose.

Figure 2 depicts elements used in a more general object fetching task, consisting of necessary perception, grasping, but also optional navigation and environment articulation plans. Such plans have well-defined behaviours and capabilities, as well as required input parameters and possible outcomes. Nesting them entails advantages in code reusability, modularity and function encapsulation, but also grants semantic meaning to the contained structures. From the figure, it becomes apparent that each of the "skills" (the (sub)plans) has a well-defined entry point, identified by a set of preconditions (shown for the PICK OBJECT and a monitor plan), effectively triggering the start of a plan. Their outcomes can either be result values (white circles) or failure signals (black circles). Unhandled failure signals can pass through plan hierarchy layers until

Algorithm 5 Partially ordered tasks, featuring a navigation task, and a grasping task that starts after navigation finished.

```
(partial-order
    ((:tag navigate
       (perform nav-action)
       (pulse close-enough))
     (:tag grasp
       (perform grasp-action)
       (pulse grasped)))
  (:order close-enough grasp))
```

a plan component signals the capability to handle it. Take for example the PICK OBJECT block, which has a LIFT subplan. When LIFTing signals a failure, the arm shall be RETRACTed to not destroy the manipulation scene. This is basically the cleanup strategy of the PICK OBJECT plan, either resulting in a successful plan exit, or failing anew for a different reason.

In the given example, monitoring tasks could be implemented using the same mechanisms (*"keep an eye on the object while moving it"*, *"monitor the gripper force when grasping the object"*). Such tasks would have similar preconditions as *regular* plans, and run concurrently besides them. So from a conceptual point of view, both follow the same architecture. Take for example a plan that navigates a robotic agent near an object in order to grasp it. The navigation task will only trigger an *"arrived"* signal once it is close enough to grasp, and only then the *"grasp object"* task is supposed to do its work. If the navigation task signals a failure *"not reachable"*, a recovery mechanism connected to this condition starts. A concurrent monitor task, being started by the same preconditions as the navigation task, could watch the object's location and interrupt normal operation in case the object moves or disappears.

Algorithm 5 shows an example implementation of such a task constraint in the CRAM plan language. The partially ordered tasks consist of one task for navigation and one for grasping objects. `(:order close-enough grasp)` constrains the grasping task to wait for the *"close-enough"* signal, which is emitted by navigation upon successful completion. As apparent from the definition of both tasks in their respective `:tag` environments (lines 2 and 5), the only connecting component between them are the signal and a partial ordering mechanism. Actually implemented funtionality is therefore kept to the task's main purpose.

More custom signals can be introduced, featuring detailed information about failure cases and results of both tasks. For the sake of brevity, this is omitted here. Precedent work on task networks was done by James Firby [21].

7. MOBILE MANIPULATION

In [22], Edsinger and Kemp equipped the non-mobile manipulator Domo with a behaviour-based control system, allowing it to place objects it was handed to by a human onto a shelf in vicinity. Their approach *"effectively extends the person's reach"* by letting the robot place an object from where it is standing. While this is a simplified version of a *fetch and place* task as described above, it lacks a distinct feature: Finding and disambiguating the object in question is completely performed by a human operator, only requiring the agent to find a suitable place for putting down the object on a predefined surface. We take this a step further and combine the fetching and placing of objects with a mobile component, allowing an autonomous agent to find and pick up an object from one location and transport it across open space to put it down at a remote location. Therefore, the task becomes not only fully autonomous in structure, but also requires the agent to perform the challenging task of finding, approaching, and properly picking up an object.

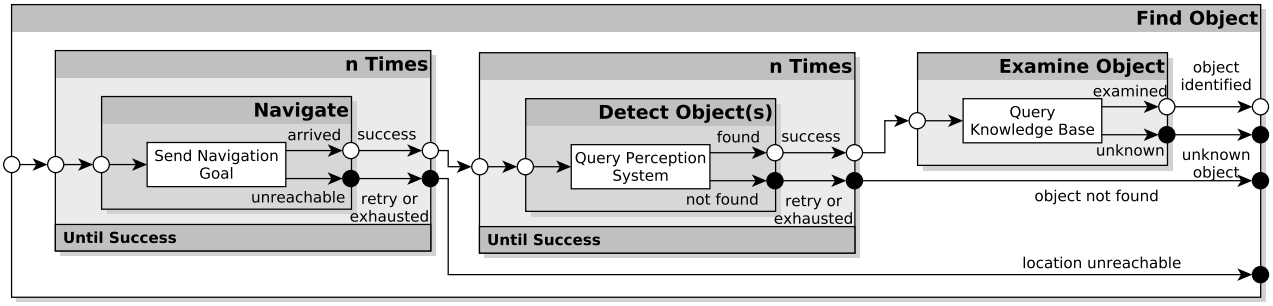


Figure 4: Schematic Implementation of the *Find Object* plan, covering multiple tries of navigation, perception and eventually examining of the found object(s).

Table 1: Distribution of time consumption in different processes while performing a Pick and Place task. Each process is described by its relative occurrence throughout the task, and the absolute amount of time spent on this process.

Process	Rel. Occur.	Abs. Time
High Level Tasks		
Fetching Objects	15.4%	65.19 s
Placing Objects	13.5%	57.21 s
Task Recovery	71.1%	300.72 s
Action Primitives		
Perception Queries	44.6%	158.25 s
Navigation	24.6%	87.31 s
Head Movement	7.8%	27.63 s
Arm Motion Planning	15.2%	53.81 s
Arm Motion Execution	7.8%	27.50 s

With this, we do not only extend the robot's capability to autonomously *acquire* the object to place, but also allow interaction with objects and the environment outside of the robot's immediate vicinity.

When performing manipulation tasks like grasping or transporting objects, such objects might start slipping from a robot's gripper or are taken away by other, possibly malicious agents [23]. To make an agent aware of such situations and put it in a position for reactive behaviour, it needs to be notified about changes in the course of action as early as possible. Monitoring the force exerted on the gripper's sensors can inform an agent about grasp irregularities, while perceptive aids can then give the agent a clue about the reason, such as an object slipping away from the gripper. To not constantly allocate the perception system for watching an object's position in the gripper during transport phases, changes in the gripper force state can trigger a rudimentary check using the vision system to verify that the carried object is still in place by tracking its relative position in the hand [24]. Figure 2 suggests the integration of such a monitoring task.

To ensure a reasonable sequence of actions in pick and place applications, Figure 4 exemplarily shows the structure of a *Find Object* plan. We showcase some of the design patterns we discussed earlier on this example: Inputs and outputs should be defined clearly, and knowledge enrichment extends the process' information gain. A more in-depth modeling of a task is shown in Figure 6. Herein, the process of grasping objects is depicted. An object can be grasped using different – to be infered – arm configurations, and can fail due to a number of well defined reasons. Again, the proper modeling of the task, possible outcomes, and available parameter spaces for the task at hand need to be defined.

Table 2: The agent's static snowledge about table setting

Person	Meal Time	Prefered Food
Tim	Breakfast	Muesli
	Lunch	Soup
	Dinner	Bread
Mary	Breakfast	Bread
	Lunch	Soup
	Dinner	Bread

Food Type	Necessary Object	Seat Placement
Muesli	Bowl	Center
	Spoon	Left
	Muesli	Anywhere near
	Milk	Anywhere near
Bread	Plate	Center
	Knife	Right
	Cup	Left, Back
Soup	Bowl	Center
	Spoon	Left

7.1 Autonomous Tablesetting Experiments

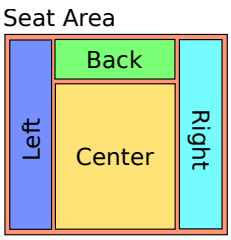
During autonomous mobile manipulation tasks, a number of action and computation primitives are employed that serve actual manipulation, or knowledge processing purposes. Table 1 shows a distribution of time consumption per task during such a single manipulation activity performed by a PR2 robot. The timing information shown was collected using robot experiences mentioned earlier. As apparent from the data shown, the overall time spent on recovery from unsuccessful tasks exceeds all other high level activities. We consider this an important insight, as it points out that the chance for overall failure in uncertain environments is very high when not considering implicit recovery mechanisms (see again Figure 6). To further fortify our point and show the importance of contextual knowledge in real world scenarios, we have conducted a reasoning intensive table setting experiment on a real PR2 in a kitchen household domain.

The experiments performed feature a table setting scenario in which a PR2 robot has to set a meal table in different contexts. By combining contextual and static knowledge, different task parameterizations result. Table 2 depicts the agent's static knowledge about the setting of tables for meals. Additionally, *Tim* likes to sit at seat 1 when he's alone, but moves to seat 2 when he gets company. Mary prefers to sit at seat 1. By varying the time of day and number of guests (the contextual knowledge), different meal scenes are produced, thus changing the task parameterizations for the autonomous agent in both respects, structurally and quantita-

(a) Initial setting, `muesli` and `spoon` already present

(b) Done placing the `bowl` near seat center

(c) Placing the `milk` close to the `bowl`, in the seat area



(d) Seat costmap distribution, based on generic knowledge

Figure 5: Phases of tablesetting experiment (5a – 5c) based on contextual knowledge, and object placement costmap based on generic tablesetting knowledge (5d). `muesli` and `milk` go anywhere in the seat area, while the `bowl` has to be placed in the center.

tively. The rules shown in Table 2 are encoded as a set of PROLOG facts, and – depending on the current contextual information – yield necessary meal objects and their respective positions using the `(required-object ?object ?location)` predicate. An example context for when *Tim* is *alone* and eats *breakfast* results in:

```
bowl at center of seat 1
spoon left of seat 1
muesli near seat 1
milk near seat 1
```

While, when *Mary* joins him, the scene changes dramatically:

```
knife right of seat 1 (for mary)
plate at center of seat 1 (for mary)
cup at left back of seat 1 (for mary)
bowl at center of seat 2 (for tim)
spoon left of seat 2 (for tim)
muesli near seat 2 (for tim)
milk near seat 2 (for tim)
```

In our experiment, we additionally supply the agent with the contextual information that `muesli` and `spoon` are already present on the table, further changing the overall task. The resulting task consists of two pick and place actions, including identification of the objects at their usual storage place, and placing them according to a breakfast table setup. Figure 5 depicts the experiment's course of action. Schematically, the contextual information is added up through several plan layers:

```
(with-context (task table-setting)
  (with-context (guest tim)
    (with-context (present muesli spoon)
      [resulting pick-and-place tasks])))
```

The resulting manipulation tasks are then performed by a PR2 robot, following the contextual parameterization of the aforementioned reasoning process, described in a vague task description using the task description language presented earlier. The agent's knowledge includes hints about how to grasp different kinds of objects – as `bowls` need to be grasped differently than `milk` boxes, and different tasks might require different grasp positions. A vague description for these objects and their grasp points is given below:

Table 3: Distribution of time consumption for single task types during table setting, separated into high level tasks and action primitives.

Process	Rel. Occur.	Abs. Time
High Level Tasks		
Fetching Objects	22.3%	366.58 s
Placing Objects	10.2%	168.67 s
Task Recovery	67.4%	1110.54 s
Action Primitives		
Perceiving Objects	61.3%	566.59 s
Navigation	16.8%	155.75 s
Head Movement	2.2%	20.41 s
Arm Motion Planning	16.0%	148.01 s
Arm Motion Execution	3.7%	34.02 s

```
(an object (type bowl)
           (handle (pose <relative-6d-pose>)
                   (for transport)))
(an object (type milk)
           (handle (pose <relative-6d-pose>)
                   (for transport))
           (handle (pose <relative-6d-pose>)
                   (for transport pouring)))
```

Here, both object types are identified by their type and supply grasping information as virtual handles. The `milk` supplies an additional grasp pose for pouring from the container and not obstructing its opening, thus allowing an agent to perform such a task more competently. The pick and place action primitives during the experiment were described using the same language:

```
(an action (to perceive)
           (obj (an object (type bowl) ..)))
(an action (to pick)
           (obj (an object (type bowl) ..)))
(an action (to place)
           (obj (an object (type bowl)
                           (in gripper) ..))
           (at (a location ..)))
```

7.2 Evaluation

During the experiments, the hierarchical robot plans had access to the respective parts of the contextual information. Using the proposed language constructs, both, initial parameterization of their tasks, as well as recovery actions, are aware of the current situation's details. Unwinding operations therefore are aware of the desired scene's state. Table 3 shows information about the task's time distribution during the agent's autonomous actions. Similar to the data shown in Table 1, recovery tasks take up most of the high level activity time. Given that table setting is more complicated and failure prone than single pick and place tasks, the need for competent and informed failure handling and recovery is even stronger. In complex manipulation environments that are only vaguely known, the manipulation scene can be unrecoverably destroyed by an uninformed agent.

The fact that the agent was able to successfully solve a comparably complex problem described by a minimum of explicit information strongly underpins our point. The contextual knowledge

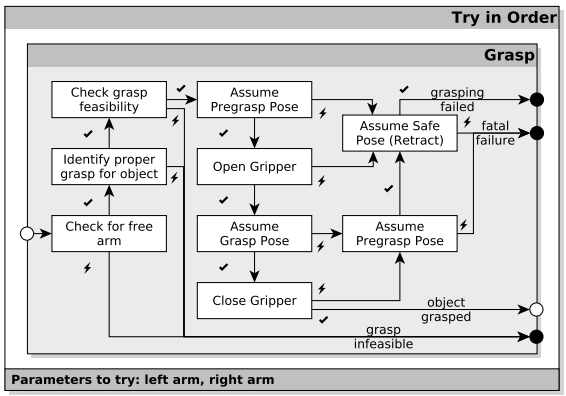


Figure 6: Reasoning, failure handling and recovery steps within a *Grasp* task for picking objects. The task can either succeed, signal infeasibility, fail cleanly during execution, or fail fatally without being able to recover. A set of parameters is tried in order: left arm, right arm.

(static and dynamic) supplied during the task allowed the agent to properly perform the tasks – and in case of an unexpected outcome (like unreachable or occupied locations) back off, unwind its actions without altering the scene, and retrying with a different parameterization. Enriching situational task information therefore makes both, a quantitative and qualitative difference, allowing an agent to explicitly react to forseen failures, and more competently handle unanticipated problems with informed heuristics. Atomic action primitives have a set of expected outcomes and side-effects. As Figure 2 depicts, monitoring of task features, such as the object position in the gripper during transport, can prevent fatal failures due to unrepairable task outcomes – such as a lost object because it slipped from the robot's hand.

8. DISCUSSION (AND OUTLOOK)

In this paper, we described plan design patterns for properly modeling action effects, and bestowing contextual knowledge upon autonomous agents. We presented concepts for concurrent monitoring of task relevant features by partially ordering task execution. To seamlessly describe the actions an agent has to perform and which details to pay attention to, we discussed a task description language that can be used to specify tasks, objects, and locations in various degrees of detail. This language is then used for sampling tasks an agent can try to perform in order to explore its parameter space, and to collect memories of task performances in a guided manner.

The presented topics are implemented in the CRAM plan language and are constantly being refined and extended. As the vocabulary of tasks in Figure 3a is limited to Fetch, Place, Perceive, and Navigate tasks, the variance in task sampling is not very high at the moment. However, we strive to extend this to different kinds of tasks (stirring, pouring, shaking, switching on and off devices, opening and closing arbitrary containers) and also broaden the object and location descriptions interpretable by the system, although more enhanced tasks will also require special hardware capabilities on the agent's side (*"Dance the Twist"*, *"Ride a Unicorn"* [25]). As described, being able to endow agents with very complex contextual knowledge hints is still off limits with the presented language attributes, though we work on extending the framework to support mechanisms for processing, and correctly interpreting such complicated reasoning steps.

Our discussed approach was successfully implemented and tested on a real PR2 robot, performing complex tasks in a kitchen environment. The ability to draw upon contextual knowledge and comprehensive (knowledge backed) failure repair capabilities allowed it to properly handle expected, and unexpected task failures, successfully performing its vaguely described task.

9. RELATED WORK

Firby [21] proposed the design of task networks for constraining tasks and branching execution thereof based on computational task outcomes. While his task formulation was specifically designed to be used locally in Reactive Action Packages (RAPs) [26], we combine this fundamental principle with the presented approaches of signal throwing in order to allow multilayer failure recovery.

Langley et al. [25] presented the ICARUS architecture for controlling cognitive agents in complex physical environments while performing pick and place tasks. In ICARUS, the MÆANDER component is responsible for executing plans generated by the DÆDALUS planner component. While MÆANDER performs according to (concurrently) delivered plans by DÆDALUS, the responsibility for recovering from unexpected situations is fixed in the high level planning component. While their system design is built upon similar principles with respect to reactivity, concurrency and versatility, our approach puts more emphasize on cascaded control loops for failure recovery before falling back to a higher abstraction layer in order to not replan globally, but first try to recover from the current problems locally.

Simmons et al. [27] have designed a robot architecture using the PRODIGY planning system to control an autonomous agent that is to perform office delivery tasks. Their architecture controls the Xavier robot which mainly does navigation tasks in a structured, but possibly unprecisely modeled environment of office rooms. Besides the proper integration of planning, vision, and execution modules, they explicitly account for imprecise knowledge on two levels: Low level components are supposed to signal whether they reached their goal, while the high level planner verifies the outcome of every action. Possible recovery mechanisms are then replanned by the planning system by adding *subgoals* to the current plan. While we extend the approached domain to a more complex Fetch and Place mobile manipulation scenario, we also extend this principle idea by allowing lower level components to perform relatively simple recovery actions within their capabilities, potentially saving planning systems from unnecessarily replanning more complex tasks.

In their recent work, Stock et al. [28] integrated an off-the-shelf HTN planner with a state machine based execution approach in order to plan, execute, and learn from task performance. Their approach tightly couples these components and includes a common representation of tasks, areas, and entities that are generated during plan execution, and could afterwards be used for performance evaluation and method learning for robot behaviour enhancement. In their paper, a PR2 robot is employed in a restaurant scenario and reasons about a serving task, which is afterwards being executed, tracing its events. While their execution architecture is well thought through and generates comprehensive execution traces, they did not focus on failure handling or possibly recovery. Equipping such an architecture with extensive failure handling strategies as presented in our work could well benefit cases in which the integrated planner wrongly plans based on insufficient or faulty information.

Acknowledgements

This work was supported in part by the DFG Project BayCogRob within the DFG Priority Programme 1527 for Autonomous Learning and the EU FP7 Projects *RoboHow* (Grant Agreement Number 288533) and *SAPHARI* (Grant Agreement Number 287513).

10. REFERENCES

[1] D. Leidner, A. Dietrich, F. Schmidt, C. Borst, and A. Albu-Schäffer, "Object-centered hybrid reasoning for whole-body mobile manipulation."

[2] C. C. Kemp, A. Edsinger, and E. Torres-Jara, "Challenges for robot manipulation in human environments," *IEEE Robotics and Automation Magazine*, vol. 14, no. 1, p. 20, 2007.

[3] C. Belta, A. Bicchi, M. Egerstedt, E. Frazzoli, E. Klavins, and G. J. Pappas, "Symbolic planning and control of robot motion [grand challenges of robotics]," *Robotics & Automation Magazine, IEEE*, vol. 14, no. 1, pp. 61–70, 2007.

[4] W. K. Edwards and R. E. Grinter, "At home with ubiquitous computing: seven challenges," in *Ubicomp 2001: Ubiquitous Computing*. Springer, 2001, pp. 256–272.

[5] J. Forlizzi and C. DiSalvo, "Service robots in the domestic environment: a study of the roomba vacuum in the home," in *Proceedings of the 1st ACM SIGCHI/SIGART conference on Human-robot interaction*. ACM, 2006, pp. 258–265.

[6] N. Roy, G. Baltus, D. Fox, F. Gemperle, J. Goetz, T. Hirsch, D. Margaritis, M. Montemerlo, J. Pineau, J. Schulte, *et al.*, "Towards personal service robots for the elderly," in *Workshop on Interactive Robots and Entertainment (WIRE 2000)*, vol. 25, 2000, p. 184.

[7] J. Winkler, M. Tenorth, A. K. Bozcuoglu, and M. Beetz, "CRAMm – memories for robots performing everyday manipulation activities," *Advances in Cognitive Systems*, vol. 3, pp. 47–66, 2014.

[8] R. O. Fikes and N. J. Nilsson, "STRIPS: A New Approach to the Application of Theorem Proving to Problem Solving," AI Center, SRI International, Tech. Rep. 43R, 1971. [Online]. Available: http://www.ai.sri.com/shakey/

[9] M. J. Schoppers, "Universal plans for reactive robots in unpredictable environments," in *Proceedings of the Tenth International Joint Conference on Artificial Intelligence (IJCAI-87)*, J. McDermott, Ed. Milan, Italy: Morgan Kaufmann publishers Inc.: San Mateo, CA, USA, 1987, pp. 1039–1046. [Online]. Available: citeseer.ist.psu.edu/schoppers87universal.html

[10] M. Beetz, A. Kirsch, and A. Müller, "RPL-LEARN: Extending an autonomous robot control language to perform experience-based learning," in *3rd International Joint Conference on Autonomous Agents & Multi Agent Systems (AAMAS)*, 2004.

[11] K. Erol, J. Hendler, and D. Nau, "HTN planning: Complexity and expressivity," in *Proceedings of the National Conference on Artificial Intelligence*. John Wiley & Sons LTD, 1994, pp. 1123–1123.

[12] S. Tellex, T. Kollar, S. Dickerson, M. R. Walter, A. G. Banerjee, S. J. Teller, and N. Roy, "Understanding natural language commands for robotic navigation and mobile manipulation." in *AAAI*, 2011.

[13] C. Matuszek, E. Herbst, L. Zettlemoyer, and D. Fox, "Learning to parse natural language commands to a robot control system," in *Experimental Robotics*. Springer, 2013, pp. 403–415.

[14] M. Hsu and T.-S. Weng, "Assisted instruction case design of robot creative assembly and control program design," in *Proceedings of the 7th WSEAS/IASME International Conference on Educational Technologies (EDUTE'11)*, 2011, pp. 106–111.

[15] D. Di Marco, M. Tenorth, K. Häussermann, O. Zweigle, and P. Levi, "Roboearth action recipe execution," in *Frontiers of Intelligent Autonomous Systems*. Springer, 2013, pp. 117–126.

[16] H. Costelha and P. Lima, "Robot task plan representation by petri nets: modelling, identification, analysis and execution," *Autonomous Robots*, vol. 33, no. 4, pp. 337–360, 2012.

[17] N. Kushmerick, S. Hanks, and D. Weld, "An algorithm for probabilistic planning," *Artificial Intelligence*, vol. 76, pp. 239–286, 1995.

[18] E. Spelke, "Initial knowledge: Six suggestions," *Cognition*, vol. 50, no. 1, pp. 431–445, 1994.

[19] D. A. Norman, *The Design of Everyday Things*, reprint ed. Basic Books, Sept. 1990. [Online]. Available: http://www.worldcat.org/isbn/0465067107

[20] M. Ingham, R. Ragno, and B. C. Williams, "A reactive model-based programming language for robotic space explorers," in *International Symposium on Artificial Intelligence, Robotics, and Automation in Space (i-SAIRAS)*, Montreal, Canada, 2001.

[21] J. Firby, "Task networks for controlling continuous processes," in *Proceedings of the Second International Conference on AI Planning Systems*, K. Hammond, Ed., Morgan Kaufmann, 1994, pp. 49–54.

[22] A. Edsinger and C. C. Kemp, "Manipulation in human environments," in *Humanoid Robots, 2006 6th IEEE-RAS International Conference on*. IEEE, 2006, pp. 102–109.

[23] A. Fagiolini, F. Babboni, and A. Bicchi, "Dynamic distributed intrusion detection for secure multi-robot systems," in *Robotics and Automation, 2009. ICRA'09. IEEE International Conference on*. IEEE, 2009, pp. 2723–2728.

[24] D. Kragic and M. Vincze, "Vision for robotics," *Found. Trends Robot*, vol. 1, no. 1, pp. 1–78, 2009.

[25] P. Langley, K. B. McKusick, J. A. Allen, W. F. Iba, and K. Thompson, "A design for the icarus architecture," *ACM SIGART Bulletin*, vol. 2, no. 4, pp. 104–109, 1991.

[26] J. Firby, "An Investigation into Reactive Planning in Complex Domains," in *Proceedings of the Sixth National Conference on Artificial Intelligence*, Seattle, WA, 1987, pp. 202–206.

[27] R. Simmons, R. Goodwin, K. Haigh, S. Koenig, J. O'Sullivan, and M. Veloso, "Xavier: Experience with a layered robot architecture," *ACM magazine* Intelligence, 1997.

[28] S. Stock, M. Günther, and J. Hertzberg, "Generating and executing hierarchical mobile manipulation plans," 2014.

Robot Action Plans that Form and Maintain Expectations

Jan Winkler and Michael Beetz

***Abstract*— Robots acting in the real world depend on plans designed prior to their deployment. Most designed plans try to take advantage of pre-defined knowledge to circumvent problems tied to the plan's purpose. While this tackles part of the challenges, the real world stays a dynamic and ever-changing place, making anticipation of all possible situations in a plan practically impossible. In this paper, we present a novel approach to give such plans access to episodic memories and experiences. This allows a robot agent to judge its own intentions and to improve itself over time. We achieve this using probabilistic prediction of the course of action, based on the current situation compared to prior experiences. Such predictions directly influence the behaviour of the agent and improve the success rate while repeatedly performing the same task.**

I. Introduction

As we start to require our autonomous robots to accomplish more and more complex tasks, such as fetching a set of objects and placing them somewhere else, we also have to equip them with complex action plans. To amortize the development efforts for such plans we should design them to be generic. They should work for different robots, in different task contexts (cleaning up in a kitchen or picking items from an order list and putting them into a box), for different objects, and in different environments. A generic plan for fetch and place, for example, might ask the robot to go to a place from where it expects to be able to see and then reach the object to be fetched. It then selects and executes appropriate reaching motions and grasps in order to pick up the objects.

Obviously, the performance of the plans and their behavior strongly depends on the capabilities of the robot executing it, the objects to be manipulated, the difficulty of tasks, and the complexity and clutteredness of the environment. Thus, a control decision that might be good for one robot in one context might not be the best choice for another robot or another context. For example, a robot might be equipped with a low-resolution camera and therefore has to go closer to the objects to be picked up to get the necessary pose accuracy.

In this paper we investigate the problem of supporting the adaptation of generic plans to specific robots, tasks, and environments by enabling the robots to automatically learn models of the effects and behavior of their plans and their subplans and by making these models accessible as first-class objects in the plan language. These models equip the robot with expectations about its plans: Given a particular task context the robot can infer whether or not the plan will be successful, which failures to expect, what course a subactivity will take, etc. The expectations can then

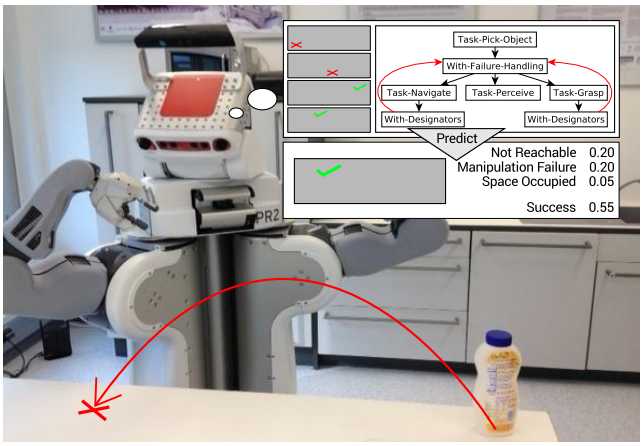


Fig. 1: PR2 Robot predicting the most probable outcome of an object displacement task.

be used as information resources to improve the context-specific plan performance. For example, the robot can stop the execution of a manipulation plan if the probability of damaging an object might grow too high. Or, it might use the expectations to alter standard control decisions according to the expectations.

We propose a framework in which we extend the plan language for robots to integrate the learning of models and the use of learned models. Using this framework a programmer can mark a subplan she wants to use experience models by using the plan pattern (with-behavior-model *plan-name code*), which asks the robot to automatically learn a behaviour model for code and make the outcome of the learning process available as the behavior model of plan-name. Now, she can use e.g. the expression (predict-behavior *plan-name* :success) to predict whether the plan is expected to succeed based on the current execution data.

Building on our earlier research on the collection of episodic memories in robot systems [1], we present an approach for generating action prediction models from such memories. The information in these models allows autonomous agents to deduce the most probable outcome of their actions and the effect of their decisions in each individual situation. Through accumulation of memories, an agent therefore improves its own performance over time, being able to make context aware decisions and forming an intuition about the course of action. Through concurrent comparison of current and memorized task instances, an agent becomes aware of what to expect while performing a task. It cannot only reason about the current situation,

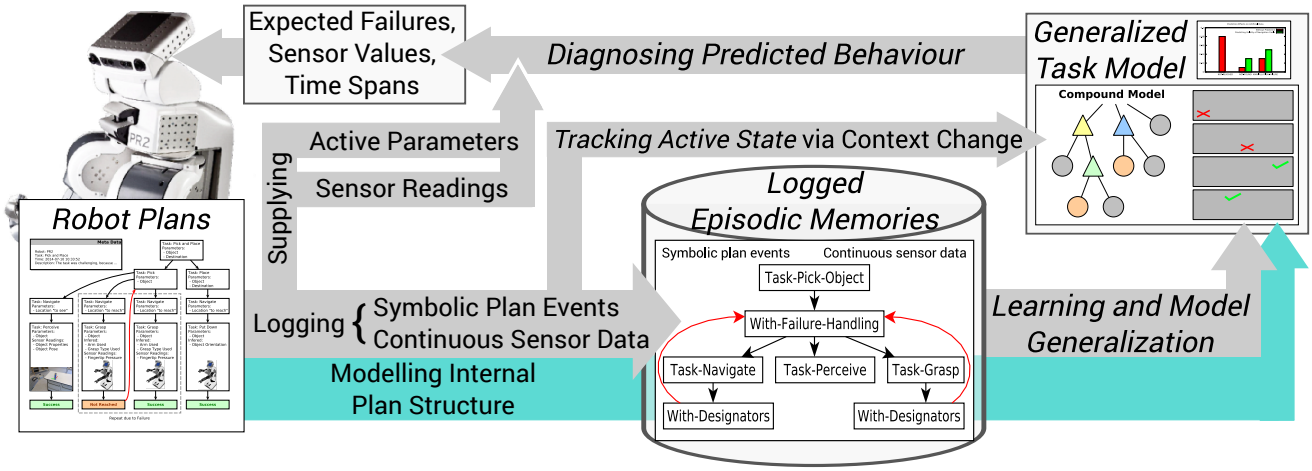


Fig. 2: Architecture Overview of the Prediction System Components and the Exchanged Information.

but arbitrary parts of the model. Unknown situations are dependably identified as all formerly experienced model tasks have definite branch and leaf states. We show that an agent drawing upon this knowledge greatly increases its chances for success in arbitrary tasks, learning decision models based on past experiences in comparison to the current situation. To make these techniques available to an agent, we present plan structure elements to transparently guide decision-making processes fitting the current situation, and to semantically annotate plan parts with relevant features and task identification tags.

In the remainder of the paper we proceed as follows. The following section introduces the conceptual apparatus underlying our approach and describes the overall architecture of the approach. We go into detail about the generalization and abstraction of experience data in Section III, and discuss the composition and application of decision models based on this experience data in Section V and IV. To demonstrate how learned concepts from experience data can be used to improve their original robot plans, experiments on simplified and on real robot task executions and the effect of prediction on task efficiency are presented in Section VI. We conclude with future and related work on the topic in Section VII and VIII, respectively.

II. Overview

Our framework for forming, maintaining, and using expectations, which is depicted in Figure 2, consists of three components. First, using specialized language constructs, we equip robot plans with the ability to annotate task relevant feature parameters to make the plans more expressive and semantically meaningful. Second, these annotations along with other semantic and all relevant sensory information, are stored in a central memory logging system that holds enough information to reconstruct the robot's internal and external state at all times. Third, from a series of these logged memories, we compute a generalized task model that enables an agent to make predictions about the effects of a task, given its current parameterization. This information is then fed back into the robot plans to positively influence parameter choice and task outcome when running plans.

Internally, robot plans have a variety of parameters of which some are task relevant, while others act as task independent control flow operators. Semantic annotation helps to distinguish which parameters are relevant, and enables an agent to inspect correlations between parameters and outcomes of its own plans on a semantic level. Parameter introspection allows an agent to *remember* what happened, and later compare it to new instances of the same plan.

While performing such annotated plans, all semantic and all possibly relevant sensory data is recorded in Episodic Memories. They describe all information to reconstruct the agent's internal and external state, deduce poses of detected objects, reenact triggered external events, and can disclose the result of single tasks and their respective parameterization. The memorized symbolic task tree connects the low-level data to higher level task concepts and plan parts by means of time stamps and unique identifiers.

With semantically interconnected low- and high-level information of whole plan executions available, a generalized model of plans can be computed based on similar task structure, and parameter and result distributions for each task can be learned. Robot plans can be complex collections of activities and can hold a multitude of such learning problems. Given a significant portion of Episodic Memories for the same plan, expectations of parameter and sensor value ranges can be formed. Based on these expectations, an agent can make predictions about which parameter combination usually yields what result, and can deduce potential correlations between them. Therefore, for a given parameterization, the most probable failures, success rates, accompanying sensor readings, and time spans per task can be calculated.

Finally, with this information available, a robotic agent can not only make decisions upon its current, narrow situational information. It can make informed decisions based on past failed or successful plan performances, and can *guess* its chances of success in a *task aware manner*. The plan lan-

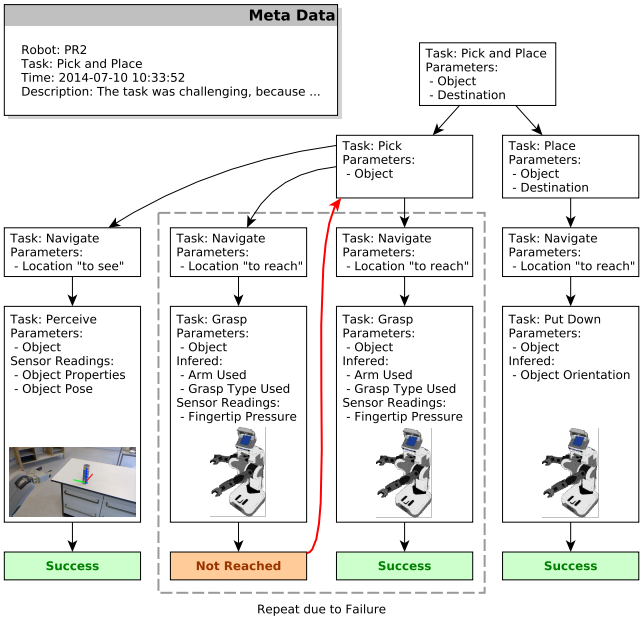


Fig. 3: Structure and Content of a single Episodic Memory instance, consisting of meta data, semantic task tree, thrown and caught failures, parameterizations, and sensor readings.

guage structures used to emit the current task's parameterization also take out this prediction. They transparently request prediction of outcomes based on task relevant parameters, and automatically reparameterize tasks if necessary.

Figure 2 gives an impression of the described architecture and the flow of information.

III. Generating Prediction Models from Episodic Memories

Episodic Memories consist of qualitative task knowledge and quantitative sensor data. In [1], we described how these different kinds of data can be semantically connected to allow access to all of a task's characteristics, as performed by a robot agent. We generalize over a number of such memories to compose a prediction model with a tree structure, describing the overall course of action of all included memories, their outcomes, and associated task parameterizations.

Figure 3 depicts the principle structure of a single task memory episode. It's sensor data portion holds actual robot movements, recorded camera images throughout the task, and external event details such as detected objects, collisions, and events generally not intentionally triggered by the robot itself, such as unvoluntary movement of body parts or the disappearance of objects. The task tree of the episode reflects the internal task structure of the performed robot plan. It includes the task hierarchy, parameterizations and outcomes, as well as failure handling paths – i.e., what kind of failure occured in which task, and where it was handled. When a subtask ended due to success or a failure, it it marked as a leaf state for this branch. To generalize over multiple episodes without losing information about mappings between parameters and outcomes or the task branches visited during

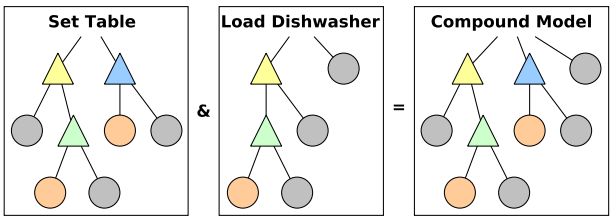


Fig. 4: Generalized Model from Episodic Memories, consisting of a combined task tree, retaining all parameter, failures, sensor value mappings, and leaf states.

execution, we condense the single task trees based on their task contexts, and preserve unique identifiers for each task of the original memories. Figure 4 depicts the structure of such a generalized model.

The task tree shown herein reflects the structure of all included episodic memories. The generalized model is not limited to one structurally equivalent task type, though – generalizing over multiple structurally different tasks yields a tree including all possible pathways, effectively reflecting all included task types, their hierarchies, and occurrences of leaf nodes to mark the possible end of a task branch. A single (sub)task might then have different outcomes – i.e. success, failures – and zero or more subtasks, depending on the content of the original episodic memories. To generate this compound tree, we combine multiple source trees based on the similarity of task types and position of each node.

Algorithm 1 shows a pseudo-code version of the steps to take in this generalization. The CombineTrees function merges all individual task trees into one tree, allowing the UnifyTree function to recurse over every node of the individual trees, comparing them, and merging them into compound branches per task context. The CombineSub function extracts the parameter and leaf state mapping information from every node and stores them, along with the specific context they originate from, into a global storage container for reference during prediction.

IV. Applying Prediction Model to Robot Plans

The generated task models serve as a blueprint showing what the course of action of a known task should look like. In order to make an agent aware of the model part currently being active, we developed and integrated a tracking mechanism that uses the exact same mechanics as the plan logging features mentioned in Section II concerning task begin and end signals. Instead of recording the performed task details though, it descends into the model tree when new tasks are entered (identified by their task type) and ascends out of subtrees when the task finishes. As generalized models can include multiple structurally different task types, their common tree diverges at some point. For an example of combining structurally different trees, refer to Figure 4. The tracking mechanism will therefore descend into the subtree fitting the internal structure of the task currently being performed, allowing models of diverse task hierarchy infor-

Algorithm 1 Task Tree Generalization

```
1: function COMBINETREES(treeList)                    ▷ Structure
2:     combinedTrees ← {}
3:     for tree in treeList do
4:         combinedTrees ← combinedTrees + tree
5:     end for
6:     return {ctx : Toplevel, subs : combinedTrees}
7: end function
8:
9: function UNIFYTREE(tree)                  ▷ Unify to Single Tree
10:    for sub in tree[subs] do
11:        refinedSub ← UnifyTree(sub)
12:        tree[subs][sub] ← CombineSub(refinedSub)
13:    end for
14:    return tree                                  ▷ tree is mutable
15: end function
16:
17: function COMBINESUB(tree)                ▷ Collect Information
18:    ctxTree ← {}
19:    for sub in tree[subs] do
20:        for ctx in sub do
21:            gParams[ctx] ← sub[ctx][param]          ▷ Params
22:            gLeafs[ctx] ← sub[ctx][leafs]       ▷ Leaf States
23:            ctxTree[ctx] ⟸ sub                ▷ Collect by Contexts
24:        end for
25:    end for
26:    return ctxTree
27: end function
```

mation content, without intermixing their distinct structural features.

When new tasks are faced, it is possible that the tracking mechanism encounters task contexts that are not present in the supplied model. In this case, it keeps track of where it left the known model and descends into a virtual branch, denoting the task contexts along the way. When ascending out of the unknown subtree again, it resolves the virtual branch nodes until it re-enters the known model. While being outside of the known task model, prediction does not yield any information.

Prediction of task information does not only work on the currently active task. Being able to tag tasks during plan execution allows prediction requests to enquire information about a specific subtask inside the model. Given this mechanism, plan parts can explicitly predict the outcome of specific subactions in their task tree. This allows to exploit knowledge about the structure of tasks, such as the result of a grasping action, i.e. *"if a robot stood at that location"* or *"if it grasped with the left arm"*. Without being able to explicitly predict future states, an agent would need to enter the situation first, before noticing that its parameterization will most probably not work correctly, which is remedied with our approach.

The presented strategy allows to keep track of which actions could follow the current task performance. Model task contexts are annotated with parameterizations, sensor readings, and leaf states from past experiences, being available to the agent at all times during it's task performance. Additionally, the agent is well aware of when it enters unchartered territory, i.e. when it leaves it's known task model and when it re-enters it. Having all this information at hand, a cognitive agent is now well equipped with access to memorized experiences, and can make the connection to the currently performed task.

V. PREDICTIONS BASED ON TASK RELEVANT FEATURES

Using out plan structure elements, tasks are transparently annotated with relevant feature variables. These variables are used in two situations: First, when recording memories during task performance, and second, when comparing a new situation to past memories. Besides individual task features, a fixed set of variables is annotated: The task type, its depth in the tree, and its result (i.e. success, or the thrown failure type).

To make predictions about the effect of an active parameterization possible, the generalized task model is used to generate a decision tree. This tree is computed from parameterization and outcome information stored for each task node. All tasks along a branch can hold failure information with a relative occurrence, calculated from a series of episodic memories. A decision tree reflects the probability of encountering one of those failures, according to the parameterization chosen during prediction. When predicting the outcome of a task, all of its branches are recursed, and a joint probability of all encountered decision tree results is calculated. The result is a table of potential failures, their probability based on past relative occurrences, and the assumed chance for success.

The result of evaluating a decision tree in different branches strongly depends on that task's parameter relevance. The success of a navigation task partly depends on the destination position, while a grasping task has a larger focus on the pose and shape of the object to pick. During evaluation, only decision tree branches are evaluated that include the currently examined task's relevant features.

The mathematical formalism used for computing the decision tree follows the design principles of the ID3 algorithm, as proposed by Quinlan [2]. It yields the accuracy of the prediction, along with the most probable result. More precisely, the J48 classifier [3] implementation (a variant of the C4.5 algorithm [4]) is used.

Predictions can be made for plan parts that can be semantically referred to, but must not necessarily be active in the current situation. This is accomplished by wrapping a plan part into a tag function which can then be identified by name in the task tree:

```
tag <tag-name> <body-code>
```

Tags are especially powerful when making long-term decisions while choosing parameters. Multiple tags of the same name in a model can be uniquely identified by their *tag path*, denoting successive tag names.

To automatically choose parameters that are transparently annotated in the current memory episode, and that satisfy

TABLE I: Conceptual declaration of the `choose` operator, specifying task relevant features and prediction requirements.

```
choose <parameters p1, ..., pn>
  generators (<p1, ..., pk>, gen1),
             (<pk+1, ..., pm>, gen2), ...
  features <feature1, fnc1>,
           <feature2, fnc2>, ...
  tag tag
  predicting time, sensor-value-1, ...
  satisfying (<fail1, chk1>,
              <fail2, chk2>, ...
  attempts attempts
  body-code ...
```

given failure rate tolerances, we introduce the `choose` operator as a special purpose language construct. The operator, as depicted in Table I, defines a set of parameters `p` that are generated by a – possibly smaller – set of generators `gen`. Based on this, task features are calculated and passed to the decision tree-backed prediction mechanism (optionally featuring a `tag` name as prediction target). The prediction result consists of failure probabilities and requested values, such as the projected time span required to complete the task, or specific sensor readings. After checking the legitimacy of the chosen parameters by verifying the prediction result using custom functions `chk`, either new parameters are chosen, or the `body-code` of the plan is executed. The reparameterization is performed at most `attempts` times before allowing an unpredicted reparameterization for execution. The outcome of `body-code` is again memorized together with the believed to be correct parameters.

Additionally, prior experiences offer information about to-be-expected time spans and sensor readings. Given that both are treated as numerical indicators, we calculate a confidence interval based on all considered memorized values, and return it to the performing plan. In the `choose` operator, the `predicting` property specifies which values to calculate and return. Requesting the `time` required for the current task then considers all instance timespans of this task type, delivering the lower and upper bounds, as well as the mean value of the time in seconds most probably required (with a confidence parameter of $\alpha = 0.95$):

```
TASK = VOLUNTARY-BODY-MOVEMENT-HEAD
CONF-INT = (m=1.26, lb=1.17, ub=1.34)
```

An explicit model of parameter to outcome correlations can not only be used for verifying feature values, but also produce hints on *which* parameters to change *how* in order to reach a target outcome. By inverting the computed decision tree, all pathways to a given result can be determined, yielding valid parameter ranges for all task relevant features. With this technique, reparameterization within `choose` can be *directed* by constraining it into these ranges. As this process can generate multiple solutions, we sort them by the number of currently unsatisfied features. The *shortest* solution is then tried for reparameterization. Robotic agents able to access this information can really draw upon task structure knowledge. By not blindly guessing new parameters, but developing an *intuition* about valid feature ranges, agents can make much more informed decisions, or drop the performance of a task completely if there is no legitimate success path available.

TABLE II: Partial training data for decision trees

x	y	$obj\text{-}dist$	**Task**	**Result**
7	5	?	FIND-OBJ	SUCCESS
8	9	?	FIND-OBJ	NOT-FOUND
5	4	2	GRASP	SUCCESS
2	3	3	GRASP	MANIP-FAILURE
4	0	?	NAVIGATE	SUCCESS
8	5	?	NAVIGATE	NOT-REACHED

Algorithm 2 Decision Tree from Memories

1: $Result \leftarrow SUCCESS$ ▷ Implicit Success
2: **if** $task = NAVIGATE$ **and** $(x > 7$ **or** $y > 5)$ **then**
3: $Result \leftarrow NOT\text{-}REACHED$ ▷ Failure
4: **else if** $task = GRASP$ **and** $obj\text{-}dist > 3$ **then**
5: $Result \leftarrow MANIP\text{-}FAILURE$ ▷ Failure
6: **else if** $task = FIND\text{-}OBJ$ **and** $(x > 7$ **or** $y > 7)$ **then**
7: $Result \leftarrow NOT\text{-}FOUND$ ▷ Failure
8: **end if**

VI. Experimental Evaluation

To show the impact of task aware prediction abilities on plan performance in robotic agents, we split our experimental evaluation in two parts. First, we will show the principle mechanism and its effects on simplified pick and place plans, compirising explicitly encoded rules that we will then recognize in the learned task model. Second, we show the performance impact in real robot plans, featuring an object detection task in which a PR2 robot is to foresee valid search regions, and circumvent failures based on prediction.

Table II shows an excerpt of training data recorded from episodic memories generated through simplified plans. The task relevant features are the target (or current, depending on the task) position of the robot (x, y), and its current distance from an object of interest ($obj\text{-}dist$). While the position is always present in the relevant features, the object distance is only present during the `GRASP` task. Features that are not present in a task are marked as *"?"*, which is then considered by the classifier.

After applying the J48 classifier, a decision tree resulting in Algorithm 2 is produced. It precisely reflects the rules encoded in the simplified plans: Navigation tasks were on purpose obstructed in areas where $x > 7$ or $y > 5$, resulting in a `NOT-REACHED` failure, and grasping would only work on $obj\text{-}dist \leq 3$, otherwise failing with `MANIP-FAILURE`. Objects cannot be found when either $x > 7$ or $y > 7$, resulting in `NOT-FOUND`.

To compare the plan performance with and without prediction, Figure 5 shows the nominal failure occurrences for both cases. The number of sample experiments was increased by a step size of two per trial. In each trial, its number of samples was used for failure occurrence evaluation, and for model training. Figure 5 shows that with growing number of

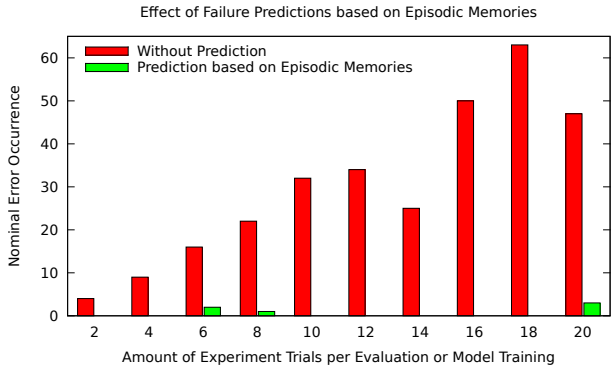


Fig. 5: Nominal failure occurrences for normal (red) and prediction-backed (green) task performance (simplified plans).

experiments per trial, the nominal failure rate grows when not predicting failures based on task parameterization. It also depicts the strong effect of the prediction model, almost nullifying problematic cases.

While this displays the strength of the prediction mechanism in simplified, well-defined environments, we also evaluated real-world experiments on a PR2 robot performing an object search task.

To showcase the validity of our approach on actual robot plans performing in the real world, we define the following scenario. A PR2 robot is positioned in front of a broad table, which is supporting several objects at different locations. The agent is now tasked to find objects of certain kinds on the table, some of which are present, while others are absent. The agent performs this task 20 times for each of the known object types $\{\text{PANCAKEMIX}, \text{SPATULA}, \text{PLATE}, \text{KETCHUP}, \text{MUESLI}\}$. Objects absent from the scene are $\{\text{KETCHUP}, \text{MUESLI}\}$, of which the agent is not informed prior to the task. The agent can decide on the x (vertical) and y (horizontal) position on the table to look at. Some objects are almost always visible (like the PLATE, being centered on the table), while other's visibility depend on the inspected side of the table. From its collected episodic memory, the decision tree shown in Table 3 results.

The result correctly identifies the KETCHUP and MUESLI to be absent and always resulting in a `NOT-FOUND` failure when trying to find them. The PANCAKEMIX is standing on the left side of the table, the PLATE near the center, and the SPATULA on the right. When allowing the agent to perform the same task again while drawing on this new knowledge, its performance greatly increases as shown in Figure 6. As no valid parameters can be generated for either KETCHUP or MUESLI objects, the search is instantly abandoned. As mentioned before, by inverting the decision tree, these endeavours can be identified as infeasible before even starting the task. The remaining failures result from noisy perception and imprecise resolution of table locations. Both, the simplified plans and real robot experiments show a

Algorithm 3 Decision Tree from Memories (robot plans)

1: $Result \leftarrow SUCCESS$ ▷ Implicit Success
2: **if** $type = PANCAKEMIX$ **and** $look\text{-}y > 1.25$ **then**
3: $Result \leftarrow NOT\text{-}FOUND$ ▷ Failure
4: **else if** $type = SPATULA$ **and** $look\text{-}y \leq 1.25$ **then**
5: $Result \leftarrow NOT\text{-}FOUND$ ▷ Failure
6: **else if** $type = PLATE$ **and** $look\text{-}y \leq 0.75$ **then**
7: $Result \leftarrow NOT\text{-}FOUND$ ▷ Failure
8: **else if** $type = KETCHUP$ **or** $type = MUESLI$ **then**
9: $Result \leftarrow NOT\text{-}FOUND$ ▷ Failure
10: **end if**

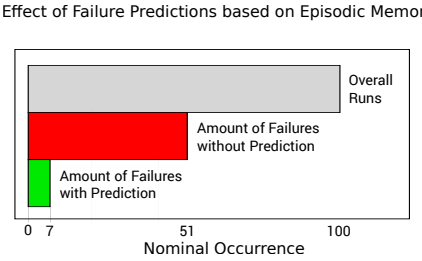


Fig. 6: Nominal failure occurrences for unpredicted (red) and predicted (green) task performance (real robot plans).

major strength of our approach, namely the task-independent prediction of possible outcomes, and the ability to inform an agent about a potentially bad parameterization.

VII. Conclusion and Future Work

In this work, we presented a novel approach for predicting the outcome and characteristics of tasks performed by robot agents, based on episodic memories that were collected during earlier task performances. We successfully formed a generalized model from episodic memories and discussed a way to compose decision trees for reasoning about the most probable outcome of the current or a semantically named situation and to predict task specific time requirements and sensor readings. Our tracking mechanism for localizing the prediction algorithm in a generalized task model is enabling a robot agent to make parameter and context aware decisions, and over time become better at tasks it performes by avoiding problematic parameterizations. We have shown that the model can cope with unknown states and can return to normal behaviour once the unknown state is over. Our techniques were explained in terms of simplified data to show the principle mechanism, and validated in a real robot scenario.

Our choice of using decision trees for modeling the most probable outcome of a task node serves the purpose of this paper well. The actual outcome prediction algorithm could possibly be exchanged for a different method, such as Random Forests [5] or Bayes Networks. Such distributions would then allow more explorative failure handling techniques, which not only heuristically reparameterize a presumably failing task, but can direct the reparameterization based on the failure probabilities in the prediction results to

maximize information gain.

While our current approach yields good results, the enhancements mentioned above could further improve the agent's prediction skills, and make more complex domains accessible to our techniques. We aim for benchmarking different kinds of probabilistic methods, such as inspected by Sridhar et al. [6], and extend the prediction system to make use of them.

To enable the agent to dynamically recreate it's decision trees and automatically add new experiences *on the go*, the process of manually generating the decision tree must be replaced by an implicit decision tree generation algorithm inside the memory recording module. In-between task runs, e.g. during idle times, the robot agent can then regenerate it's own prediction model based on the new experiences acquired. We have plans to incorporate the C5.0 algorithm, which is an enhancement over the C4.5 algorithm used for decision tree generation based on training data.

To decide *when* to enhance the prediction model with new experience data, the right amount of datasets needs to be identified. A statistically insignificant amount of data might prove to worsen the agent's performance by unrightfully ruling out whole regions of the parameter space, preventing the agent from exploring valid parameterizations. The identification of this significance is a crucial step in automating the self-improvement of such a cognitive, predicting agent. We currently use a thresholding technique to ensure significance, but strive for an automated analysis of actual episode content.

The algorithms and techniques presented in this paper are implemented in our openly available software package `beliefstate`[1], which also incorporates the mentioned robot memory collection functionality.

VIII. Related Work

Enhancing a cognitive agent's behaviour and strategies to improve known, and tackle new tasks is one of the core competences of artificial intelligence and has been approached in various ways. Implementing learning capabilities as part of the robot control language itself is one of the approaches that produced promising results in the past. A prominent example for this is the Reactive Plan Language (RPL, [7]) extension RPL$_{Learn}$ [8] by Beetz et al., and its successor, RoLL [9], by Alexandra Kirsch. RoLL allows for specifying which experience to collect and what tasks to associate with it, such as collecting all experience data from daily household chores and learning concepts from it overnight. The experiences are then abstracted from their specific instances and supply the robot with hints about possibly good parameter choices when approaching the once-encountered problem again, based on former results. While this approach is very sound, works satisfactorily for the problems presented, and in general aims at the same direction as our work, we allow the agent to predict not only good parameter values, but also potential outcomes based on an active parameter selection, expected sensor readings, and different failure probabilities depending on which task is being performed.

Predicting the effects of robot actions based on a model of the current task has also been approached by Beetz et al. [10]. Representing the task at hand as a Probabilistic Hybrid Action Model (PHAM) allows for specifying probability ranges of what might occur, and when to stop performing an action as it is bound to fail. Their concept builds up a system of rules that, when encountering a certain task, specifies the probabilities for potential effects of that task. Being very similar in basic principles, our work diverges from their approach insofar that we extend the robot task domain from pure navigation to the much more complex field of mobile manipulation. Also, by not relying on a fixed task model per se, but building the model solely from experience data, more diverse situations can be learned without additional manual development effort.

Several learning algorithms have been evaluated in the context of a navigation and box-pushing task by Sridhar et al. [6]. They describe a reward driven reinforcement learning approach to enhance a behaviour based robot's performance. Their robot OBELIX performs tasks in the collision free navigation domain and is able to find and push boxes around. The strategy of how to approach a box to push and when to change the approaching direction to not "lose" the box (or the box getting stuck in a corner) is being learned based on the success of its actions. Multiple interesting learning algorithms are evaluated, such as *Q learning*, *Weighted Hamming Distance*, and *Statistical Clustering*, of which some might be applicable to our work at hand.

When building models for effect prediction and task projection, collection of experience and log data is inevitable. In the context of the RACE project, Stock et al. [11] presented the integration of an off-the-shelf HTN planner with a state-machine based plan execution approach that generates data for learning new robot behaviour. They developed a fluent-backed event description formalism that reflects an agent's behaviour, which tasks it performed, and what the task's current parameterization was. While they did not yet apply learning mechanisms to their collected execution traces, the presented formalism fits very well into the overall RACE architecture for experience based autonomous robot learning.

Research on learning-based sensor value prediction in autonomous robot applications was done by Saegusa et al. [12]. Their basic idea is to acquire an unknown environment model while interacting with it, treating the situation as a sensory prediction problem. The model is extended successively based on actual sensor values while interacting with the real environment, and predicting what the sensor values might be for planned future actions, based on that very model. They conclude with experiments on the humanoid robot James and show the effectiveness of their approach in actual experiments on that platform. We combine this basic idea with a semantic task model to allow not only prediction of sensor readings, but also of discrete events and qualitative task outcomes, effectively enhancing the range of predicted information.

[1] `http://github.com/fairlight1337/beliefstate`

State Abstraction was researched thoroughly by Andre et al. [13]. They reason that, when introducing abstracted states, the dimensionality of the search space decreases strongly, greatly easing the search for a solution to the learning problem at hand. Our claims are supported by their findings, not only in state abstraction, but also in terms of *"fixed relative valuation of the possible 'exits' of subroutines"* in MAXQ [14], which is basically the same concept as our task *leaf* states. To judge when a task has ended or a new task was begun, they employ the `call-subpham` function, which reflects the same behaviour as our task entry and exit delimiters. While conceptually they tread a similar path to ours, we differ strongly application-wise. They use their models to generate a sound initial behaviour for an agent, while we concentrate on allowing task look-ahead by predicting potential future world-reactions to the behaviour of an autonomous robot.

ACKNOWLEDGEMENTS

This work is supported by the EU FP7 project *RoboHow* (Grant Agreement Number 288533).

REFERENCES

[1] J. Winkler, M. Tenorth, A. K. Bozcuoglu, and M. Beetz, "CRAMm – memories for robots performing everyday manipulation activities," *Advances in Cognitive Systems*, vol. 3, pp. 47–66, 2014.

[2] J. R. Quinlan, "Induction of decision trees," *Machine learning*, vol. 1, no. 1, pp. 81–106, 1986.

[3] G. Holmes, A. Donkin, and I. H. Witten, "Weka: A machine learning workbench," in *Intelligent Information Systems, 1994. Proceedings of the 1994 Second Australian and New Zealand Conference on*. IEEE, 1994, pp. 357–361.

[4] J. R. Quinlan, "Bagging, boosting, and c4. 5," in *AAAI/IAAI, Vol. 1*, 1996, pp. 725–730.

[5] L. Breiman, "Random forests," *Machine learning*, vol. 45, no. 1, pp. 5–32, 2001.

[6] S. Mahadevan and J. Connell, "Automatic programming of behavior-based robots using reinforcement learning," *Artificial intelligence*, vol. 55, no. 2, pp. 311–365, 1992.

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

[8] M. Beetz, A. Kirsch, and A. Müller, "RPL-LEARN: Extending an autonomous robot control language to perform experience-based learning," in *3rd International Joint Conference on Autonomous Agents & Multi Agent Systems (AAMAS)*, 2004.

[9] A. Kirsch, "Robot Learning Language – Integrating Programming and Learning for Cognitive Systems," *Robotics and Autonomous Systems Journal*, vol. 57, no. 9, pp. 943–954, 2009. [Online]. Available: http://dx.doi.org/10.1016/j.robot.2009.05.001

[10] M. Beetz and H. Grosskreutz, "Probabilistic hybrid action models for predicting concurrent percept-driven robot behavior," in *Proceedings of the Sixth International Conference on AI Planning Systems*. AAAI Press, 2000.

[11] S. Stock, M. Günther, and J. Hertzberg, "Generating and executing hierarchical mobile manipulation plans," 2014.

[12] R. Saegusa, F. Nori, G. Sandini, G. Metta, and S. Sakka, "Sensory prediction for autonomous robots," in *Humanoid Robots, 2007 7th IEEE-RAS International Conference on*. IEEE, 2007, pp. 102–108.

[13] D. Andre and S. J. Russell, "State abstraction for programmable reinforcement learning agents," in *AAAI/IAAI*, 2002, pp. 119–125.

[14] T. G. Dietterich, "The maxq method for hierarchical reinforcement learning." in *ICML*. Citeseer, 1998, pp. 118–126.

Bibliography

[Winkler and Beetz, 2015a] Winkler, J. and Beetz, M. (2015a). Generalized plan design and entity description for autonomous mobile manipulation in open environments. Under review.

[Winkler and Beetz, 2015b] Winkler, J. and Beetz, M. (2015b). Robot action plans that form and maintain expectations. Under review.

[Winkler et al., 2014] Winkler, J., Tenorth, M., Bozcuoglu, A. K., and Beetz, M. (2014). CRAMm – memories for robots performing everyday manipulation activities. *Advances in Cognitive Systems*, 3:47–66.