Project no.:	**027657**
Project full title:	**Perception, Action & Cognition through learning of Object-Action Complexes**
Project Acronym:	**PACO-PLUS**
Deliverable no.:	**D5.1.3**
Title of the deliverable:	**Technical Report: Extensions to PKS in support of dialogue and conversational acts**

Contractual Date of Delivery to the CEC:	31 January 2010	
Actual Date of Delivery to the CEC:	17 June 2010	
Organisation name of lead contractor for this deliverable:	UEDIN	
Author(s): Ronald Petrick and Mark Steedman		
Participant(s): UEDIN		
Work package contributing to the deliverable:	WP4, WP5	
Nature:	R	
Version:	Final	
Total number of pages:	99	
Start date of project:	1st Feb. 2006	**Duration:** 52 month

Project co-funded by the European Commission within the Sixth Framework Programme (2002–2006)
Dissemination Level

PU	Public	**X**
PP	Restricted to other programme participants (including the Commission Services)	
RE	Restricted to a group specified by the consortium (including the Commission Services)	
CO	Confidential, only for members of the consortium (including the Commission Services)	

Abstract:

The core focus of WP5.1 is the generalisation of the basic symbolic representation of OACs and ancillary planning apparatus to communication and language. WP5.1 builds on WP3 to extend the representation of OACs to communicative acts and is tightly integrated with WP4, in particular the theoretical and practical components of WP4.3, to provide the foundational infrastructure needed to support high-level linguistic concepts. This deliverable primarily focuses on the PKS planner and its associated components, and highlights the extensions required for PKS to support high-level dialogue acts, as an instance of the general problem of planning with incomplete information and sensing. Some of the mechanisms needed to integrate PKS within the robot control and communication architecture are also outlined, as are associated learning techniques and experimental studies. The related deliverable D5.2.3 describes the computational problem of natural language acquisition in greater detail.

Keyword list: Planning with Knowledge and Sensing (PKS), planning for natural language, three-level control architecture, action effect learning, robot integration, dialogue planning infrastructure

Table of Contents

1. Executive Summary

The core focus of WP5.1 is the generalisation of the basic symbolic representation of OACs and ancillary planning apparatus to communication and language. More specifically, this work builds on WP3 to extend the representation of OACs to communicative acts (Task 5.1.1) and is tightly integrated with WP4, in particular the theoretical and practical components developed as part of WP4.3, to provide the foundational infrastructure needed to support high-level linguistic concepts (Task 5.1.2). This deliverable primarily focuses on the planning mechanisms of our architecture, namely the PKS ("Planning with Knowledge and Sensing") planner [6, 7] and its associated components. In particular, we outline the extensions required for PKS to support high-level dialogue and conversational acts within the three-level control and communication architecture developed as part of WP1 and WP4. (The associated deliverable D5.2.3 describes the computational problem of natural language acquisition in greater detail.)

PKS is a state-of-the-art contingent planner that constructs plans in the presence of incomplete information and sensing actions. Like most AI planners, PKS operates best in discrete, symbolic state spaces defined using logical languages. Unlike traditional planners, PKS builds plans at a more abstract "knowledge level" by representing and reasoning about how the planner's knowledge changes during planning. In particular, PKS is based on an extended version of STRIPS [4], where actions are described in terms of their effects on the planner's knowledge state, rather than the world state. PKS also supports numerical reasoning, run-time variables [3], and features like functions that arise in real-world planning scenarios.

In deliverable D5.1.2, we previously described how PKS could be extended "downwards" to support low-level continous control systems of the kind used in PACO-PLUS. In this deliverable, we focus on the "upwards task", and use PKS as a planning framework for natural language and communication. We view the problem of planning dialogue acts as an instance of the general problem of planning with incomplete information and sensing, and outline the mechanisms needed to extend ordinary action planning to dialogue planning. To do so, we work at the level of speech acts and focus on the changes we must make to the representational and reasoning components of the "standard" version of PKS. In particular, these extensions enable the planner to reason about the incomplete knowledge of multiple agents, and obtain further information through dialogue acts modelled as information-gathering sensing actions. Thus, we treat the dialogue problem as isomorphic to the problem of planning with incomplete information and sensing for a single planning agent. This work can be seen as a practical implementation of the theory first outlined in [8].

This deliverable also provides an updated description of the control architecture we use to integrate the planner and its associated components on the PACO-PLUS robot platforms. A central aspect of this work is the inclusion of a new plan execution monitor that operates together with the PKS planner to control high-level replanning and resensing activities. We also describe an extended version of a mechanism for learning high-level planning actions, a new planner resulting from our work extending PKS, and the results of experimental studies investigating the wider applicability of planning techniques to challenging problems in natural language generation.

Seven additional documents are attached to this deliverable, highlighting the important role of planning in the PACO-PLUS architecture for both dialogue planning and task planning. These documents also describe the support mechanisms developed as part of WP4 that enable PKS to generate plans (dialogue or otherwise) that can be executed in the lower level robot control spaces. More generally, these components provide the infrastructure needed to support the longer term objectives of language and communication in WP5. Here we briefly sketch the relation of each paper to this workpackage and deliverable, and make links to the specific contributions of each paper.

[A] *(Internal PACO-PLUS Technical Report)* This document provides a snapshot of the extensions currently being implemented in PKS to support knowledge-level dialogue planning, as well as associated control mechanisms required for both task and dialogue plan execution. This document reviews PKS's

basic knowledge representation framework, and the additions we require to extend this framework to multi-agent contexts supporting plan generation in the PACO-PLUS demonstration scenarios. This document builds on the integration architecture developed as part of WP4 (see [E] and [G] below) to support language and interaction as part of WP5. (Earlier versions of this document appeared in deliverables D4.3.5 and D5.1.2.)

[B] *(Journal paper to appear in Computational Intelligence, Special Issue on Planning and Scheduling Applications)* This paper describes the results of experiments designed to test the feasibility of current off-the-shelf planners as a general problem solving mechanism for challenging problems in natural language generation. Such a study is a necessary companion to the dialogue planning work of PACO-PLUS, in order to assess the more widespread potential of planning techniques for natural language processing. This document describes work performed as part of WP5.

[C] *(Paper to be presented at the 2010 European Conference on Artificial Intelligence in Lisbon, Portugal)* In previous work reported in deliverable D5.1.2, we describe a mechanism for learning STRIPS-style [4] actions effects from world state snapshots of the form produced by the PACO-PLUS control architecture described in [A]. This approach is based on a voted kernel perceptron learning model [1, 5] which operates over a compact vector representation using deictic features embodying a notion of attention. The present paper extends our previous approach and enables us to apply our learning mechanism in noisy and partially observable planning domains. Such domains are characteristic of the kind we investigate in PACO-PLUS, and those that arise in real-world dialogue scenarios. For the purposes of an initial evaluation, we demonstrate our approach on standard benchmarks from the International Planning Competition. This document describes work developed as part of WP4 and WP5, with connections to WP6. (An earlier version of this work appeared in deliverable D5.1.2.)

[D] *(Paper presented at the ICAPS 2009 Workshop on Planning and Learning in Thessaloniki, Greece)* This document presents a more detailed, but older version of [C]. This work was developed as part of WP4 and WP5, with connections to WP6. (An earlier version of this work appeared in deliverable D5.1.2.)

[E] *(Paper presented at the ICAPS 2009 Workshop on Planning and Plan Execution for Real-World Systems in Thessaloniki, Greece)* This document presents an overview of our approach to robot control in PACO-PLUS, by using Object-Action Complexes (OACs) as a mechanism for overcoming the representational differences that arise between different components of an integrated robot system. This paper also describes the role of the high-level plan execution monitor developed for PKS, as a means of controlling replanning and resensing activities during plan execution. Such a control mechanism is required for both task and dialogue plan execution in PACO-PLUS systems. This document highlights work performed as part of WP4 and WP1, with connections to WP5 and WP8.

[F] *(Paper presented at the ICAPS 2009 Workshop on Generalized Planning: Macros, Loops, Domain Control in Thessaloniki, Greece)* This paper describes the structure of a new planner called P^2 (Planning with Programs) that features a rich action representation language where action effects can be described by program segments. P^2 is derived from the original PKS planner, and is the direct result of implementation work designed to enhance the basic PKS codebase to support dialogue planning. While this new planning mechanism will not initially be used for dialogue planning in PACO-PLUS, it provides a baseline for future applications of generalised planning techniques to natural language processing. This document describes work performed as part of WP4 and WP5.

[G] *(Poster presented at the 2010 International Conference on Cognitive Systems in Zurich, Switzerland)* This document describes the integration of the PKS planner on the ARMAR robot platform, with a focus on task execution in the system architecture. Since the extensions required for communication and dialogue are built on top of the ordinary PKS system, the mechanisms for high-level task plan execution in the PACO-PLUS architecture also form the core of the mechanisms used for dialogue plan

execution. This document provides a snapshot of the execution architecture currently implemented on ARMAR, and highlights work performed as part of WP4 and WP1, with connections to WP5 and WP8.

Together, these papers report a number of significant developments:

- A prototype version of the dialogue planner, built as an extension to the PKS planner.
- Results of a comprehensive set of experiments applying general purpose planning techniques to challenging problems in natural language generation.
- An extension of our action effect learning mechanism to noisy and partially observable planning domains, plus initial experiments applying the learning mechanism to real state data from the SDU robot/vision system.
- A new version of the PKS planner including a re-implementation of the core PKS plan generation algorithms, and an initial version of a plan execution monitor for PKS.
- A spin-off planner (P^2), capable of planning with an extended representation language supporting action effects described as program segments.
- Changes to the PACO-PLUS control architecture supporting the integration of the new version of PKS and its plan execution montior, and providing the necessary framework for dialogue planning.
- An implementation of many of the theoretical components planned and previously reported under WP4 and WP5, providing a complete path from continuous low-level representations to high-level models for planning and language. The representational structures underlying these components (and those previously implemented) make use of the OAC concept, previously defined as part of WP4, and provide the necessary infrastructure to go beyond the planned work of WP5.

A number of questions remain open at the time of this report and constitute further work.

- Integration activities are ongoing to incorporate the PKS-based dialogue planner onto the ARMAR robot platform.
- A comprehensive set of experiments are planned to evaluate the effectiveness of our dialogue planner and plan execution monitor in complex domains.
- Additional analysis of the action models learnt from the SDU robot/vision state data is planned, along with further testing of our action effect learning mechanism.
- We are continuing to investigate the role of probabilistic models in high-level plan generation and monitoring processes. Since nondeterminacy will undoubtedly arise as the result of perception and action at the robot/vision level, we are continuing to investigate how best to manage such information at the higher control levels. Our current approach makes use of rapid replanning [9] which has been successfully applied by planners that have competed in the probabilistic track of the International Planning Competition [2].

Besides the connections to WP1, WP4, and WP5 already mentioned, this workpackage also has interactions with other workpackages including WP2, WP3, WP6, and WP7.

2. Publications Associated with D5.1.3

[A] **Integrating Low-Level Robot/Vision with High-Level Planning and Sensing in PACO-PLUS**
Ronald Petrick, Christopher Geib, and Mark Steedman
Internal PACO-PLUS Technical Report, January 2010.

> **Abstract:** This document describes UEDIN's contribution to ongoing integration work in PACO-PLUS, to link low-level robot platforms with high-level planning systems. We investigate two robot domains from the planning-level point of view, as the basis for our integration work: an object manipulation task in the KIT kitchen domain using the ARMAR robot platform, and an object stacking problem using SDU's robot/vision system. A high-level action representation is developed for each integration scenario, for the purpose of goal-directed planning, by abstracting the capabilities of a robot and its working environment. We also present a common message-passing and control architecture to facilitate communication in the integrated system. High-level planning is provided by the PKS planner, which UEDIN has extended for use in robotic and linguistic domains. We also briefly discuss a number of related integration tasks being pursued by UEDIN, such as plan execution monitoring, high-level action learning, and dialogue planning. This document describes components developed as part of WP4 to provide high-level support for low-level continuous control systems, and forms the basic infrastructure needed to support language and communication in WP5. It also forms part of the project-wide integration work reported in WP1, with connections to WP8. The attached version of the report (as of 2010-05-01) reflects the current state of integration activities, including the first version of the plan execution monitor and prototype changes to PKS supporting dialogue planning.

[B] **Experiences with Planning for Natural Language Generation**
Alexander Koller and Ronald Petrick
Computational Intelligence, Special Issue on Planning and Scheduling Applications, 2010, to appear.

> **Abstract:** Natural language generation (NLG) is a major subfield of computational linguistics with a long tradition as an application area of automated planning systems. While current mainstream approaches have largely ignored the planning approach to NLG, several recent publications have sparked a renewed interest in this area. In this paper, we investigate the extent to which these new NLG approaches profit from the advances in planner expressiveness and efficiency. Our findings are mixed. While modern planners can readily handle the search problems that arise in our NLG experiments, their overall runtime is often dominated by the grounding step they perform as preprocessing. Furthermore, small changes in the structure of a domain can significantly shift the balance between search and preprocessing. Overall, our experiments show that the off-the-shelf planners we tested are unusably slow for nontrivial NLG problem instances. As a result, we offer our domains and experiences as challenges for the planning community.

[C] **Learning action effects in partially observable domains**
Kira Mourão, Ronald Petrick, and Mark Steedman
Proceedings of the European Conference on Artificial Intelligence (ECAI 2010), 2010.

> **Abstract:** We investigate the problem of learning action effects in partially observable STRIPS planning domains. Our approach is based on a voted kernel perceptron learning model, where action and state information is encoded in a compact vector representation as input to the learning mechanism, and resulting state changes are produced as output. Our approach relies on deictic features that assume an attentional mechanism that reduces the

size of the representation. We evaluate our approach on a number of partially observable planning domains, and show that it can quickly learn the dynamics of such domains, with low average error rates. We show that our approach handles noisy domains, conditional effects, and that it scales independently of the number of objects in a domain.

[D] **Learning action effects in partially observable domains**
Kira Mourão, Ronald Petrick, and Mark Steedman
Proceedings of the ICAPS 2009 Workshop on Planning and Learning, pp. 15–22, 2009.

Abstract: We investigate the problem of learning action effects in partially observable STRIPS planning domains. Our approach is based on a voted kernel perceptron learning model, where action and state information is encoded in a compact vector representation as input to the learning mechanism, and resulting state changes are produced as output. Our approach relies on deictic features that embody a notion of attention and reduce the size of the representation. We evaluate our approach on a number of partially observable planning domains, adapted from domains used in the International Planning Competition, and show that it can quickly learn the dynamics of such domains, with low average error rates. Furthermore, we show that our approach handles noisy domains, and scales independently of the number of objects in a domain, making it suitable for large planning scenarios.

[E] **Combining Cognitive Vision, Knowledge-Level Planning with Sensing, and Execution Monitoring for Effective Robot Control**
Ronald Petrick, Dirk Kraft, Norbert Krüger, and Mark Steedman
Proceedings of the ICAPS 2009 Workshop on Planning and Plan Execution for Real-World Systems, pp. 58–65, 2009.

Abstract: We describe an approach to robot control in real-world environments that integrates a cognitive vision system with a knowledge-level planner and plan execution monitor. Our approach makes use of a formalism called an Object-Action Complex (OAC) to overcome some of the representational differences that arise between the low-level control mechanisms and high-level reasoning components of the system. We are particularly interested in using OACs as a formalism that enables us to induce certain aspects of the representation, suitable for planning, through the robot's interaction with the world. Although this work is at a preliminary stage, we have implemented our ideas in a framework that supports object discovery, planning with sensing, action execution, and failure recovery, with the long term goal of designing a system that can be transferred to other robot platforms and planners.

[F] **P^2: A Baseline Approach to Planning with Control Structures and Programs**
Ronald Petrick
Proceedings of the ICAPS 2009 Workshop on Generalized Planning: Macros, Loops, Domain Control, pp. 59–64, 2009.

Abstract: Many planners model planning domains with "primitive actions," where action preconditions are represented by sets of simple tests about the state of domain fluents, and action effects are described as updates to these fluents. Queries and updates are typically combined in only very limited ways, for instance using logical operators and quantification. By comparison, formalisms like Golog permit "complex actions," with control structures like `if-else` blocks and `while` loops, and view actions as programs. In this paper we explore the idea of planning directly with complex actions and programs. We describe the structure of a simple planner based on undirected search, that generates plans by simulating

the execution of action programs before they are added to a plan. An initial evaluation compares this approach against a classical heuristic planner using a domain whose program structures have been compiled into ordinary PDDL actions. Initial results illustrate that in certain domains, planning directly with programs can lead to a significant performance improvement. This work offers a baseline planner to compare against alternate approaches to planning with programs.

[G] **Connecting Knowledge-Level Planning and Task Execution on a Humanoid Robot using Object-Action Complexes**
Ronald Petrick, Nils Adermann, Tamim Asfour, Mark Steedman, and Rüdiger Dillmann
Poster in the proceedings of the International Conference on Cognitive Systems (CogSys), 2010.

Abstract: This poster presents a snapshot of the current integration of the PKS planner on the ARMAR humanoid robot platform. The planner is responsible for building high-level plans, and operates closely with an execution monitor that makes decisions concerning plan continuation, object resensing, and replanning. High-level plans are executed on the ARMAR robot as a series of robot-level manipulation and sensing actions. Task planning and task execution are connected using Object-Action Complexes (OACs), a universal representation usable at all levels of a cognitive architecture, combining the representational and computational efficiency of STRIPS rules, and the object- and situation-oriented concept of affordance, together with the logical clarity of formalisms like the event calculus.

References

[1] M.A. Aizerman, E.M. Braverman, and L.I. Rozonoer. Theoretical foundations of the potential function method in pattern recognition learning. *Automation and Remote Control*, 25:821–837, 1964.

[2] Daniel Bryce and Olivier Buffet. The uncertainty part of the 6th international planning competition. *http://ippc-2008.loria.fr/wiki/*, 2008.

[3] Oren Etzioni, Steve Hanks, Daniel Weld, Denise Draper, Neal Lesh, and Mike Williamson. An approach to planning with incomplete information. In *Proceedings of KR-92*, pages 115–125, 1992.

[4] Richard Fikes and Nils Nilsson. STRIPS: a new approach to the application of theorem proving to problem solving. *AI Journal*, 2:189–208, 1971.

[5] Yoav Freund and Robert Shapire. Large margin classification using the perceptron algorithm. *Machine Learning*, 37:277–296, 1999.

[6] Ronald P. A. Petrick and Fahiem Bacchus. A knowledge-based approach to planning with incomplete information and sensing. In *Proceedings of AIPS-02*, pages 212–221, 2002.

[7] Ronald P. A. Petrick and Fahiem Bacchus. Extending the knowledge-based approach to planning with incomplete information and sensing. In *Proc. of ICAPS-04*, pages 2–11, 2004.

[8] Mark Steedman and Ronald P. A. Petrick. Planning dialog actions. In *Proceedings of the 8th SIGdial Workshop on Discourse and Dialogue (SIGdial 2007)*, pages 265–272, Antwerp, Belgium, September 2007.

[9] Sungwook Yoon, Alan Fern, and Robert Givan. FF-Replan: A baseline for probabilistic planning. In *Proceedings of the 17th International Conference on Automated Planning and Scheduling*, pages 352–359, 2007.

Appendix A

Integrating Low-Level Robot/Vision with High-Level Planning and Sensing in PACO-PLUS

Technical Report

Ronald Petrick, Christopher Geib, and Mark Steedman*
University of Edinburgh
2010-05-01

Abstract

This document describes UEDIN's contribution to ongoing integration work in PACO-PLUS, to link low-level robot platforms with high-level planning systems. We investigate two robot domains from the planning-level point of view, as the basis for our integration work: an object manipulation task in the KIT kitchen domain using the ARMAR robot platform, and an object stacking problem using SDU's robot/vision system. A high-level action representation is developed for each integration scenario, for the purpose of goal-directed planning, by abstracting the capabilities of a robot and its working environment. We also present a common message-passing and control architecture to facilitate communication in the integrated system. High-level planning is provided by the PKS planner, which UEDIN has extended for use in robotic and linguistic domains. We also briefly discuss a number of related integration tasks being pursued by UEDIN, such as plan execution monitoring, high-level action learning, and dialogue planning. This document describes components developed as part of WP4 to provide high-level support for low-level continuous control systems, and forms the basic infrastructure needed to support language and communication in WP5. It also forms part of the project-wide integration work reported in WP1, with connections to WP8.

Revision history

2010-05-01 : Update reflecting the current state of integration activities, including the first version of the plan execution monitor and prototype changes to PKS supporting dialogue planning.

2009-07-10 : Minor revision of the document.

2009-01-29 : This report presents a status update on UEDIN's integration work, extending and replacing two previous UEDIN technical reports: *A Scenario for Integrating Low-Level Robot/Vision, Mid-Level Memory, and High-Level Planning with Sensing* (2008-07-20) and *A Scenario for Integrating Low-Level Robot/Vision and High-Level Planning with Sensing* (2008-05-30).

**Contact: rpetrick@inf.ed.ac.uk*

Contents

List of Figures

List of Tables

1 Introduction

In this document we describe the state of integration work designed to link low-level robot systems with high-level planning components as part of WP4. This work forms part of the project-wide integration work reported in WP1, and is connected to WP8.

We focus on two robot domains here, as the basis for our integration tasks: an object manipulation task in the KIT kitchen domain using the ARMAR robot platform [Asfour et al., 2006, 2008], and an object stacking problem using SDU's robot/vision system [Kraft et al., 2008]. In this document we will discuss UEDIN's contribution to ongoing integration efforts, from the point of view of the planning task and required high-level representation in these scenarios.

High-level planning capabilities are supplied by the PKS planner [Petrick and Bacchus, 2002, 2004], which UEDIN is extending for use in robotic and linguistic domains as part of WP4 and WP5. PKS is a state-of-the-art knowledge-level planner that constructs plans in the presence of incomplete information. Unlike traditional planners, PKS builds plans at the "knowledge level", by representing and reasoning about how the planner's knowledge state changes during plan generation. Actions are specified in a STRIPS-like [Fikes and Nilsson, 1971] manner in terms of action preconditions (state properties that must be true before an action can be executed) and action effects (the changes the action makes to properties of the state). PKS is able to construct conditional plans with sensing actions, and supports numerical reasoning, run-time variables [Etzioni et al., 1992], and features like functions that arise in real-world planning scenarios.

Like most AI planners, PKS operates best in discrete, symbolic state spaces described using logical languages. As a result, integration work between UEDIN and KIT/SDU has centred around the design of high-level action representations that abstract the capabilities of a robot and its working environment for goal-directed planning. Integration also requires the ability to communicate information between system components. To this end, UEDIN has developed a socket communication library and message passing protocol (WP4) that facilitates the exchange of messages between the planner and lower-level system components.

Early integration work established a link between SDU's robot/vision system and UEDIN's high-level planning components. More recently, we have focused on combining the high-level planner with KIT's ARMAR robot platform. Although differences between the KIT and SDU systems require different high-level action representations, the "core" concepts in each representation are similar, and the communication architecture is unchanged across platforms. We have also reserved a role for possible mid-level processes which could be added to our architecture, beyond the end of PACO-PLUS.

In the remainder of this document we describe the high-level planning representation developed for each integration scenario, and the associated message-passing and control architecture. In Section 2, we discuss UEDIN's integration work with KIT. In Section 3, we focus on the SDU integration domain. In Section 4, we describe possible extensions to our current action representations. In Section 5, we introduce the current specification of the message passing protocol and communication architecture. In Section 6, we briefly discuss a number of related integration tasks being investigated by UEDIN as part of WP4 and WP5, including plan execution monitoring, action learning, and dialogue planning. Finally, in Section 7 we mention future directions for this work.

2 Object Manipulation in a Kitchen Domain (KIT/UEDIN Integration)

In this section we describe the state of ongoing integration work to link UEDIN's high-level planning components with KIT's *ARMAR robot platform* [Asfour et al., 2006, 2008]. We primarily focus on the action representation used to support planning in the KIT robot domain, and the kinds of plans we can currently build in this environment.

Our work centres around modelling the tasks that the ARMAR robot can perform within the *KIT kitchen environment* (previously described as part of WP1 and WP8). This domain is a real-world kitchen with commonplace objects and appliances (e.g., cereal boxes, cups, plates, fridge, stove, etc.). The kitchen is divided into a number of discrete *workspaces* (e.g., sideboard, cupboard, dishwasher, etc.), each of which support a range of different activities and challenges for the robot. At an abstract level, the tasks mainly involve manipulating the set of objects available in the domain, which may require moving between the workspaces (e.g., the robot may have the task of bringing a juice container from the fridge to the sideboard).

The high-level representation must accurately model the dynamics of the robot's interaction with the kitchen environment in order to enable directed, goal-driven planning to be performed. As a result, there are a number of interesting complexities that must be considered. For instance, both the robot and certain kitchen objects can move between workspaces. Some objects can also be contained within other objects. Moreover, the robot has multiple gripper hands and must decide which gripper it should use to manipulate an object; due to the geography of the kitchen and the hardware limitations of the robot, some objects require that a particular hand be used. The action specification must also encode the robot's ability to upright toppled objects or nudge flat objects to the edge of a surface before grasping. As future work, we will also consider the situation where the robot has incomplete information about the location of certain objects in the kitchen and must therefore actively *sense* the world to find them.

Typically, we will use our domain representation to build plans that direct the robot to relocate objects in the kitchen. For instance, the robot may be given the goal of clearing all dirty dishes to the dishwasher, or collecting the ingredients needed to make breakfast. As a result, our representation must be expressive enough to support such high-level tasks, while permitting efficient planning in a real-world setting.

2.1 High-level domain description

To encode the above scenario, we formally define the sets of actions and properties we require for the high-level planning domain. Our focus will be on building a STRIPS-style representation [Fikes and Nilsson, 1971] that can be used with the PKS planner [Petrick and Bacchus, 2002, 2004].

Constants We begin by defining a list of special constants which denote certain aspects of our domain, such as valid workspace locations, gripper hands, and kitchen objects. In particular, we make use of the following constants:

- *Workspaces*: `cupboard`, `dishwasher`, `fridge`, `sideboard`, `stove`,
- *Gripper hands*: `lefthand`, `righthand`,
- *Objects*: `applejuice`, `calgonitsalt`, `graninijuice`, `measuringcup`, `ricebox`, `vitaliscereal`.

Constants act as labels that let us reference designated objects within our representation and generated plans. In this case, we define five discrete workspaces in the kitchen, two gripper hands, and a set of six objects. Besides the special objects listed above, the kitchen also contains a set of cups and plates, denoted by constants of the form `cup1`, `cup2`, ..., `cupN` and `plate1`, `plate2`, ..., `plateM`, respectively. Some of the defined constants also serve a dual purpose in our representation. For instance, the workspace locations also denote objects that can be manipulated in certain ways (e.g., the dishwasher can be opened and closed). The constant list can easily be extended if new objects are added to the domain.

Actions

The set of available high-level actions is shown at the top of Table 1. All of these actions are considered to be ordinary "physical" planning actions with effects that change the state of the world. These actions correspond to (sets of) low-level motor programs that the robot can execute in the domain. In our current domain specification, we do not consider high-level "sensing" actions that enable the planner to direct the robot to observe and return certain information about the state of the world. (The robot is assumed to have its normal low-level sensors which provide it with world-level information, however.)

High-level actions divide the set of object manipulation tasks into context-dependent operations. For instance, `grasp` can be used to pick up objects from the centre of flat surfaces like the sideboard, while `grasp-fromEdge` is used to pick up (flat) objects from the edge of a surface. The `remove-from` action is used to take objects out of other objects like the fridge. Once the robot is holding an object it can transfer it between hands using the `pass-object` action. Actions also exist for placing objects onto surfaces (`put-down`) or into other objects (`put-in`). Certain objects can be repositioned to enable grasping. For example, flat objects can be moved to the edge of a surface (`nudge-toEdge`) and "toppled" objects can be placed in an upright position (`place-upright`). The task of opening objects is also divided into multiple actions. For instance, `open` is used to open objects that require a single-handed operation (e.g., opening the cupboard) while `open-partial` and `open-complete` allow a more complex, two-step opening procedure (e.g., opening the fridge requires the robot to switch hands halfway through the process). Objects can be closed in a single step using the `close` action. Finally, the robot is able to `move` between workspaces in the kitchen.

All of the above actions are *parametrized* with variables denoting objects, locations, and gripper hands. During planning, these variables are replaced with constants to produce specific action instances. It is these action instances that will ultimately be passed to the robot and converted into low-level motor programs for execution in the real world. We note that many of these actions are *object centric* and modelled with a high degree of abstraction: we do not provide plan-level actions that specify 3D spatial coordinates, joint angles, or similar real-valued parameters. Details of the actual execution of these actions are left to the robot controller. (E.g., `grasp` does not specify the gripper pose that should be used to pick up an object, nor the spatial coordinates of the object's location.)

Properties

High-level properties (predicates and functions) model features of the world, robot, and domain objects, and correspond to abstract versions of information available at the robot level. High-level properties are typically formed by combining information from multiple low-level sensors in particular ways, and packaging that information into a logical form. Like actions, high-level properties can be parametrized and instantiated by defined constants.

Actions	
close(?l,?h)	Close ?l with gripper ?h.
grasp(?o,?l,?h)	Grasp object ?o from ?l using gripper ?h.
grasp-fromEdge(?o,?l,?h)	Grasp object ?o from the edge of ?l using gripper ?h.
move(?l1,?l2)	Move the robot from location ?l1 to location ?l2.
nudge-toEdge(?o,?l,?h)	Nudge flat object ?o to the edge of ?l using gripper ?h.
open(?l,?h)	Open ?l with gripper ?h.
open-partial(?l,?h)	Partially open ?l with gripper ?h.
open-complete(?l,?h)	Finish opening ?l with gripper ?h.
pass-object(?o,?h1,?h2)	Pass object ?o from gripper ?h1 to ?h2.
place-upright(?o,?l,?h)	Put object ?o upright at ?l using gripper ?h.
put-down(?o,?l,?h)	Put object ?o down at ?l using gripper ?h.
put-in(?o,?l,?h)	Put object ?o into ?l using gripper ?h.
remove-from(?o,?l,?h)	Remove object ?o from ?l using gripper ?h.
Properties	
atEdge(?o)	A predicate indicating that object ?o is at the edge of a surface.
flat(?o)	A predicate indicating that object ?o is flat.
gripperEmpty(?h)	A predicate indicating that gripper ?h is empty.
hand(?h)	A predicate indicating that ?h is a valid gripper hand.
inGripper(?o,?h)	A predicate indicating that object ?o is in gripper ?h.
location(?l)	A predicate indicating that ?l is a valid location in the kitchen.
object(?o)	A predicate indicating that ?o is a valid object in the domain.
objLocation(?o,?l)	A predicate indicating that object ?o is at location ?l.
objOpen(?o)	A predicate indicating that the door of object ?o is fully open.
objPartialOpen(?o)	A predicate indicating that the door of ?o is partially open.
robotLocation = ?l	A function indicating that the robot is at location ?l.
toppled(?o)	A predicate indicating that object ?o is in a toppled state.

Table 1: High-level actions and properties in the kitchen domain

The high-level properties in the kitchen domain are shown at the bottom of Table 1. These properties capture the high-level dynamics of the world while leaving certain lower-level properties to the robot system (e.g., 3D coordinates, gripper angles, etc.). For instance, `robotLocation` denotes the location of the robot in the kitchen and `objLocation` models object locations, in terms of the location constants defined above, rather than spatial coordinates. The `atEdge` property indicates an object is at the edge of a particular surface. The grippers' states are modelled by two properties: `inGripper` means that a particular object is in one of the robot's grippers, while `gripperEmpty` means that the gripper is empty. Object openness is represented by two properties that track whether an object is partially open (`objPartialOpen`) or completely open (`objOpen`). Certain object features are also captured in a binary way. For instance, objects may be `flat` or `toppled`. Finally, `location`, `object`, and `hand` are special "type" predicates that map the range of constants into particular classes, letting us restrict the constants that can be instantiated for a given parameter.

2.2 Representing actions for planning

Using the above constants, actions, and properties we can write planning operators for the actions we require. Our current domain encoding is given in Table 2. These actions are formalized for use with the PKS planner, however, we have simplified the syntax here. We note that the `&` and `|` operators in certain action preconditions correspond to conjunction and disjunction operations, respectively. Action effects are defined in terms of the changes they make to the planner's knowledge state, and so references to K_f denote an update to a particular PKS database used to model its knowledge of world facts (similar to a standard STRIPS database).

Restrictions and limitations

Due to the physical layout of the kitchen environment and current hardware limitations of the ARMAR robot, our high-level actions encode a number of constraints which limit their operation. For instance, the `close` action can be used to close the cupboard, dishwasher, or fridge, however the robot's right gripper must be used to close the cupboard and dishwasher; the left gripper must be used to close the fridge. Likewise, the `open` action must be used to open the cupboard and dishwasher, while `open-partial` and `open-complete` must be used to open the fridge. Similar types of constraints exist for other actions in our representation. There are also constraints still under discussion that haven't yet been encoded in our current representation (e.g., can flat objects be in a toppled state? Does the robot need to slide a plate to the edge of the cupboard before removing it?). While some of these restrictions may be lifted in the future, others are necessary for modelling the correct operation of the robot.

We also note that this action representation is preliminary and our encoding may be extended in the future to accommodate new actions or properties. For instance, we are considering the addition of two high-level *sensing* actions: an action that checks a workspace for specific objects, and an action the determines whether an object is in a suitable orientation for grasping or stacking. More discussions are needed with KIT to properly define such actions.

Actions	Preconditions	Effects
close(?l,?h)	((?l=cupboard & ?h=righthand) | (?l=dishwasher & ?h=righthand) | (?l=fridge & ?h=lefthand)) robotLocation=?l (objOpen(?l) | objPartialOpen(?l) gripperEmpty(?h)	del(K_f,objOpen(?l)) del(K_f,objPartialOpen(?l))
grasp(?x,?l,?h)	object(?x) (?l=sideboard | ?l=stove) hand(?h) ¬flat(?x) ¬toppled(?x) robotLocation=?l objLocation(?x,?l) gripperEmpty(?h)	add(K_f,inGripper(?x,?h)) del(K_f,gripperEmpty(?h)) del(K_f,objLocation(?x,?l))
grasp-fromEdge(?x,?l,?h)	object(?x) (?l=sideboard | ?l=stove) hand(?h) flat(?x) atEdge(?x) robotLocation=?l objLocation(?x,?l) gripperEmpty(?h)	add(K_f,inGripper(?x,?h)) del(K_f,gripperEmpty(?h)) del(K_f,objLocation(?x,?l)) del(K_f,atEdge(?x))
move(?l1,?l2)	location(?l1) location(?l2) ?l1 ≠ ?l2 robotLocation=?l1	add(K_f,robotLocation=?l2)
nudge-toEdge(?x,?l,?h)	object(?x) (?l=sideboard | ?l=stove) hand(?h) flat(?x) ¬atEdge(?x) robotLocation=?l objLocation(?x,?l) gripperEmpty(?h)	add(K_f,atEdge(?x))
open(?l,?h)	(?l=cupboard | ?l=dishwasher) ?h=righthand robotLocation=?l ¬objOpen(?l) gripperEmpty(?h)	add(K_f,objOpen(?l))
open-partial(?l,?h)	?l=fridge ?h=lefthand robotLocation=?l ¬objOpen(?l) ¬objPartialOpen(?l) gripperEmpty(?h)	add(K_f,objPartialOpen(?l))

Continued on next page...

Actions	Preconditions	Effects
open-complete(?l,?h)	?l=fridge ?h=righthand robotLocation=?l ¬objOpen(?l) objPartialOpen(?l) gripperEmpty(?h)	add(K_f,objOpen(?l)) del(K_f,objPartialOpen(?l))
pass-object(?x,?h1,?h2)	object(?x) hand(?h1) hand(?h2) ?h1 ≠ ?h2 inGripper(?x,?h1) gripperEmpty(?h2)	add(K_f,gripperEmpty(?h1)) add(K_f,inGripper(?x,?h2)) del(K_f,gripperEmpty(?h2)) del(K_f,inGripper(?x,?h1))
place-upright(?x,?l,?h)	object(?x) location(?l) hand(?h) toppled(?x) robotLocation=?l objLocation(?x,?l) gripperEmpty(?h)	del(K_f,toppled(?x))
put-down(?x,?l,?h)	object(?x) (?l=sideboard | ?l=stove) hand(?h) robotLocation=?l inGripper(?x,?h)	add(K_f,gripperEmpty(?h)) add(K_f,objLocation(?x,?l)) del(K_f,inGripper(?x,?h))
put-in(?x,?l,?h)	object(?x) ((?l=cupboard & hand(?h)) | (?l=dishwasher & ?h=righthand) | (?l=fridge & ?h=lefthand)) robotLocation=?l objOpen(?l) inGripper(?x,?h)	add(K_f,gripperEmpty(?h)) add(K_f,objLocation(?x,?l)) del(K_f,inGripper(?x,?h))
remove-from(?x,?l,?h)	object(?x) ((?l=cupboard & hand(?h)) | (?l=fridge & ?h=lefthand)) robotLocation=?l objOpen(?l) objLocation(?x,?l) ¬toppled(?x) gripperEmpty(?h)	add(K_f,inGripper(?x,?h)) del(K_f,gripperEmpty(?h)) del(K_f,objLocation(?x,?l))

Table 2: Representation of high-level actions in the kitchen domain

2.3 Example plans

In this section we give three examples of plans we can currently generate in the kitchen domain using PKS and the above action descriptions.

Common initial conditions

In each example we consider a scenario with only 3 objects: the vitalis cereal, the apple juice, and a plate. Initially, all the objects and the robot are located at the sideboard. The plate is considered to be a flat object and the apple juice box is in a toppled state. The cupboard, dishwasher, and fridge doors are all closed. Thus, we have the following common initial conditions:

- *Objects names*: `vitaliscereal`, `applejuice`, `plate1`,
- *Initial object locations*: `objLocation(vitaliscereal,sideboard)`, `objLocation(applejuice,sideboard)`, `objLocation(plate1,sideboard)`,
- *Initial robot location*: `robotLocation = sideboard`,
- *Object properties*: `flat(plate1)`, `toppled(applejuice)`.

In each example we consider the goal of returning particular objects to different locations in the kitchen: the vitalis cereal to the cupboard, the plate to the dishwasher, and the apple juice to the fridge. The plan in each case must also ensure that any objects opened should be closed again by the end of the plan. Since our current action representation does not include sensing actions, the resulting plans will be *linear* plans, i.e., simple sequences of actions.

2.3.1 Example 1

Goal: The `vitaliscereal` should be in the `cupboard`.

Plan
`grasp(vitaliscereal,sideboard,lefthand)`
`move(sideboard,cupboard)`
`open(cupboard,righthand)`
`put-in(vitaliscereal,cupboard,lefthand)`
`close(cupboard,righthand)`

In this case, the object manipulation is straightforward. The plan directs the robot to pick up the vitalis cereal with its left gripper, move to the cupboard, open the cupboard door with its right gripper, place the cereal in the cupboard, and close the door.

2.3.2 Example 2

Goal: `plate1` should be in the `dishwasher`.

Plan
`nudge-toEdge(plate1,sideboard,lefthand)`
`grasp-fromEdge(plate1,sideboard,lefthand)`
`move(sideboard,dishwasher)`
`open(dishwasher,righthand)`
`pass-object(plate1,lefthand,righthand)`
`put-in(plate1,dishwasher,righthand)`
`close(dishwasher,righthand)`

Since `plate1` is a flat object, the plan first directs the robot to nudge the object to the edge of the table before grasping it with its left hand. The robot can then move to the dishwasher and open it with its right hand. In this case, the robot must pass the plate between its hands and put it into the dishwasher using its right hand. (This behaviour results from the restriction that ensures the robot only manipulates the dishwasher with its right hand.) The plan finishes by directing the robot to close the dishwasher door.

2.3.3 Example 3

Goal: the `applejuice` should be in the `fridge`.

Plan
`place-upright(applejuice,sideboard,lefthand)`
`grasp(applejuice,sideboard,righthand)`
`move(sideboard,fridge)`
`open-partial(fridge,lefthand)`
`pass-object(applejuice,righthand,lefthand)`
`open-complete(fridge,righthand)`
`put-in(applejuice,fridge,lefthand)`
`close(fridge,lefthand)`

Since the apple juice is initially in a `toppled` state, the plan directs the robot to upright the object before grasping it with its right hand and moving to the fridge. In this case, opening the fridge is a two-step operation that begins with the robot's left gripper and finishes with the robot's right gripper. In between, the robot must pass the apple juice between its hands. Once the fridge is open, the plan directs the robot to put the apple juice in the fridge and close the fridge to complete the plan.

We note that instead of considering the individual goals in the above examples, we could have given the planner the more complex goal of performing all of the above tasks in a single plan (i.e., "clean up the kitchen"). One possible solution that PKS could produce in this case is a plan that conjoins each of the above plan fragments with appropriate `move` actions inserted, to return the robot to the `sideboard` to retrieve the next object.

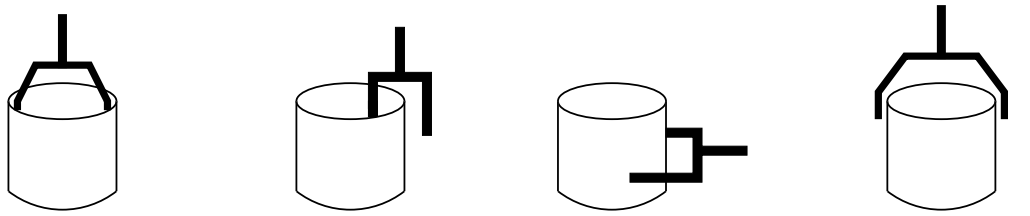
3 Object Stacking with Sensing (SDU/UEDIN Integration)

In this section we discuss a second planning domain, which combines UEDIN's high-level architecture with SDU's *cognitive vision robot platform* [Kraft et al., 2008] (part of WP4.1). While we have recently focused on integration between KIT and UEDIN systems, our work with SDU is ongoing. In particular, we continue to extend our high-level architecture and protocols—which were initially developed for use with SDU (and have been successfully transferred to the KIT system). The more mature state of integration between SDU and UEDIN provides us with an opportunity to develop and experiment with new components (e.g., high-level sensing actions and plan execution monitoring) before deploying them on the KIT platform. Furthermore, by working with multiple robot systems we can better ensure we develop *general* techniques that can be transferred to other platforms—a requirement we believe is essential for cognitive architectures to be successful.

The testing domain we have developed with SDU is a simple object manipulation scenario. We assume a *table* with a number of *objects* that are graspable by the robot. We consider situations with no more than 10 objects and, initially, only 1-3 objects. For simplicity we assume that objects are generally cylindrical in shape but not necessarily identical. In particular, each object can have a different *radius* which determines its size. Objects may or may not be *open* containers which, together with object size, determines whether or not we can *stack* objects inside other objects.

The goal of the scenario is to clear all open objects from the table, by removing them to some designated location (e.g., a shelf, a corner of the table, etc.). The location may also be restricted in some way as to force object stacking in order to successfully complete the task. For instance, there might only be room for 2 objects to sit side by side on a shelf, meaning all other objects would have to be appropriately stacked. The high-level planner will typically have only incomplete information concerning the openness of objects and must therefore plan explicit *sensing* actions to determine whether a particular object is open or not. Unlike ordinary physical actions which change the state of the world, sensing actions typically return information about the world state without necessarily changing it. Object openness plays two important roles in this scenario: as a goal condition that determines which objects should be removed from the table, and as a prerequisite for stacking operations.

This scenario also reserves a role for mid-level memory components (WP4.2) within a testing environment that lets us investigate the interaction between all three levels of the system. For example, consider a plan that includes a high-level sensing action to determine the openness of an object. At the low level, the robot/vision system may be able to ascertain whether an object is open or not by one of two means: it can *poke* an object in order to verify its concavity, or it can *focus* the vision system on the object at a higher level of resolution. A mid-level memory component might be able to make a more informed choice between poking and focusing operations and, thus, could *refine* a high-level plan before passing it to the low level. The robot/vision system must then interpret, understand, and execute the plans generated and refined by the upper levels. Although we are currently interested in establishing a direct connection between the robot/vision system and planner, the opportunity remains for integrating mid-level components in the future.



(a) Grasp Type A (b) Grasp Type B (c) Grasp Type C (d) Grasp Type D

Figure 1: Robot grasp types available to the planner

3.1 High-level domain description

To encode the above scenario, we define a set of high-level actions and properties, as described in Table 3. In contrast to the domain description of the previous section, our representation will include both ordinary "physical" actions that change the state of the world, and high-level "sensing" actions that observe the state of the world, but don't necessarily change it. Furthermore, the set of defined actions differs from that of the KIT scenario (e.g., the KIT domain focuses on multi-handed object manipulation while the SDU domain deals with multiple grasping options and object stacking). Certain aspects of the domain representation, and the high-level control architecture, remain identical however. As in the previous domain, high-level properties and actions not only form the basis of the planner's formal domain representation but are related to low-level features and motor programs.

Physical actions

In discussions with SDU we have agreed to model four types of grasping actions at the planning level, as illustrated in Figure 1. These actions correspond to a subset of the possible grasping options the robot is capable of performing. In general, these actions exhibit the following behaviour:

- *Grasp Type A*: This action can only be used to grasp objects at the top of a stack, or an empty object on the table. Objects must also satisfy a minimum and maximum radius restriction.
- *Grasp Type B*: This action can only be used to grasp objects on the table that are not part of a stack. Objects must also satisfy a minimum radius restriction.
- *Grasp Type C*: This action can only be used to grasp objects that aren't contained in other objects, i.e., the "outermost" object which must be on the table. Objects must also satisfy a maximum radius restriction.
- *Grasp Type D*: This action can only be used to grasp objects that aren't contained in other objects, i.e., objects that are on the table. Objects must also satisfy a maximum radius restriction. For simplicity, we will assume that objects stacked within the object being grasped will not affect the grasp.

For the planner's domain encoding it is necessary to subdivide Grasp Type A into two separate actions, to avoid reasoning about conditional effects. The planner therefore has five grasp actions available to it, corresponding to the four types of grasps available to the robot. (For the purposes of the sample plans in this document we only require Grasp

Types A and D.) Each grasping action takes a single argument, `?x`, denoting the label of an object. We have agreed that each object in the world will be designated by a string of the form `objN`, where `N` is a non-negative integer, e.g., `obj42`.

We have also encoded four actions for moving and manipulating objects when successfully grasped (i.e., the "put" actions in Table 3). Each manipulation action is *object centric* and modelled with a high degree of abstraction. For instance, we do not provide plan-level actions that specify 3D spatial coordinates, joint angles, or similar real-valued parameters. The `putAway` action is particularly generic and should be considered a placeholder for a more complex (possibly predefined) operation that clears an object from the table to its final destination location. For the purpose of this document we will assume that objects are put away onto a shelf. We also note that both `putInto-objOnTable` and `putInto-stack` actions denote stacking operations which will have as a prerequisite the property that objects can only be stacked into open objects.

Sensing actions

The high-level representation also includes a single sensing action, `sense-open(?x)`. At the planning level, this action is modelled as an information gathering or *knowledge-producing action* that provides the planner with information about the openness of an object. The high-level description of this action does not, however, prescribe how the robot/vision system should actually obtain this information. For instance, a `sense-open` action could potentially be executed at the low level as a *poke* operation which tests an object's concavity, or a *focus* operation which directs the vision system to study an object at a higher resolution. (A mid-level memory process could also potentially mediate between these choices.) Currently, the robot/vision system uses a poking operation, but this action is subject to change in the future.

Properties

Table 3 also shows the current set of high-level properties we have defined for this domain. Our list includes a set of predicates and functions which we have agreed could reasonably be provided to the planner from sensor information available at the robot/vision level. These properties are subject to change, however, as our requirements evolve.

3.2 Representing actions for planning

Using the above properties we can write PKS operators for the actions in this domain. For simplicity, we have made the following restrictions in our action encodings: (i) all objects are initially assumed to be on the table, (ii) grasp type C will initially be omitted (grasp type B is not required for our initial examples), and (iii) the `put-onTable` action will initially be omitted since there are no initial object stacks.

Our current domain encoding is given in Table 4. These actions are formalized for use with the PKS planner, however, we have simplified the syntax here. Although most of the details of the actual action encodings can be ignored, we mention two important points. First, each action operator is parametrized with a set of arguments that can denote any object in the world. Thus, all of our actions are object centric. Second, our encoding takes advantage of PKS's ability to work with functions and simple numerical expressions, which we include as part of the action preconditions and effects. For instance, the radius of an object plays a role in determining whether or not it can be stacked inside another object, and the minimum/maximum grasp values help determine whether or not a particular grasp action can be applied. Our domain encoding can be extended as needed to accommodate new actions or properties that may arise in the future.

Actions	
graspA-fromTable(?x)	Grasp object ?x from the table using Grasp Type A.
graspA-fromTopOfStack(?x)	Grasp object ?x from the top of a stack using Grasp Type A.
graspB-fromTable(?x)	Grasp object ?x from the table using Grasp Type B.
graspC-fromTable(?x)	Grasp object ?x from the table using Grasp Type C.
graspD-fromTable(?x)	Grasp object ?x from the table using Grasp Type D.
put-onTable(?x)	Put object ?x onto the table.
putInto-objOnTable(?x,?y)	Put object ?x into object ?y, which is on the table.
putInto-stack(?x,?y)	Put object ?x into object ?y, which is at the top of a stack on the table.
putAway(?x)	Put object ?x away.
sense-open(?x)	Determine whether object ?x is open or not.
Properties	
clear(?x)	A predicate indicating that no object is stacked in ?x.
graspAMinRadius = ?x graspAMaxRadius = ?x graspBMinRadius = ?x graspCMaxRadius = ?x graspDMaxRadius = ?x	Functions indicating the minimum/maximum radius restrictions for each grasp type.
gripperEmpty	A predicate describing whether the robot's gripper is empty or not.
inGripper(?x)	A predicate indicating that the robot is holding object ?x in its gripper.
inStack(?x,?y)	A predicate indicating that object ?x is in a stack with object ?y at its base.
isIn(?x,?y)	A predicate indicating that object ?x is stacked in object ?y.
onShelf(?x)	A predicate indicating that object ?x is on the shelf.
onTable(?x)	A predicate indicating that object ?x is on the table.
open(?x)	A predicate indicating that object ?x is open.
radius(?x) = ?y	A function indicating that the radius of object ?x is ?y.
reachableA(?x) reachableB(?x) reachableC(?x) reachableD(?x)	Predicates indicating that object ?x is reachable by the gripper using a particular grasp.
shelfSpace = ?x	A function indicating that there are ?x empty shelf spaces.

Table 3: High-level actions and properties in the object stacking domain

Actions	Preconditions	Effects
graspA-fromTable(?x)	reachableA(?x) clear(?x) gripperEmpty onTable(?x) radius(?x) ≥ graspAMinRadius graspAMaxRadius ≥ radius(?x)	add(K_f,inGripper(?x)) del(K_f,gripperEmpty) del(K_f,onTable(?x))
graspA-fromTopOfStack(?x)	reachableA(?x) clear(?x) gripperEmpty radius(?x) ≥ graspAMinRadius graspAMaxRadius ≥ radius(?x) (∃ ?z). inStack(?x,?z) onTable(?z)	add(K_f,inGripper(?x)) del(K_f,gripperEmpty) (∀ ?y). isIn(?x,?y) ⇒ del(K_f,isIn(?x,?y)) add(K_f,clear(?y)) (∀ ?z). inStack(?x,?y) ⇒ del(K_f,inStack(?x,?z))
graspB-fromTable(?x)	reachableB(?x) clear(?x) gripperEmpty onTable(?x) radius(?x) ≥ graspBMinRadius	add(K_f,inGripper(?x)) del(K_f,gripperEmpty) del(K_f,onTable(?x))
graspD-fromTable(?x)	reachableD(?x) gripperEmpty onTable(?x) graspDMaxRadius ≥ radius(?x)	add(K_f,inGripper(?x)) del(K_f,gripperEmpty) del(K_f,onTable(?x))
put-onTable(?x)	inGripper(?x)	add(K_f,gripperEmpty) add(K_f,onTable(?x)) del(K_f,inGripper(?x))
putInto-objOnTable(?x,?y)	?x ≠ ?y inGripper(?x) open(?y) clear(?y) onTable(?y) radius(?y) > radius(?x)	add(K_f,gripperEmpty) add(K_f,isIn(?x,?y)) add(K_f,inStack(?x,?y)) del(K_f,clear(?y)) del(K_f,inGripper(?x)) (∀ ?w). inStack(?w,?x) ⇒ del(K_f,inStack(?w,?x)) add(K_f,inStack(?w,?y))
putInto-stack(?x,?y)	?x ≠ ?y inGripper(?x) open(?y) clear(?y) radius(?y) > radius(?x) (∃ ?z). inStack(?y,?z) onTable(?z)	add(K_f,gripperEmpty) add(K_f,isIn(?x,?y)) del(K_f,clear(?y)) del(K_f,inGripper(?x)) (∀ ?z). inStack(?y,?z) ⇒ add(K_f, inStack(?x,?z)) (∀ ?w). inStack(?w,?x) ⇒ del(K_f,inStack(?w,?x)) add(K_f,inStack(?w,?z))
putAway(?x)	inGripper(?x) shelfSpace > 0	add(K_f,onShelf(?x)) add(K_f,gripperEmpty) del(K_f,inGripper(?x)) shelfSpace = shelfSpace - 1
sense-open(?x)	¬K_w(open(?x)) onTable(?x)	add(K_w,open(?x))

Table 4: Representation of high-level actions in the object stacking domain

As with our planning domain in the previous section, the actions in Table 4 use a PKS-style notation which is similar to STRIPS. However, unlike STRIPS, PKS uses multiple databases as the basis for its representation. Thus, references to K_f and K_w in the "effects" section of an action denote two PKS databases: K_f is like a standard STRIPS databases that stores the planner's knowledge of facts, while K_w is a specialized database for storing the effects of sensing actions. Also, $\neg K_w$`open(?x)` in the description of `sense-open` is a knowledge precondition that ensures the planner does not include a sensing action in a plan if it already knows the outcome of the sensing (i.e., if the planner already knows whether an object is open or not then it shouldn't sense the object).

3.3 Example plans

Using the above action descriptions, we give three examples of planning problems we can solve with PKS. In each example we consider a scenario with 2 objects. Each object has a size as indicated by its radius. We also assume certain minimum/maximum values for the grasps but these values don't play a large role in these examples. (For simplicity we use integer values in our examples however we also permit real-valued quantities.)

Common initial conditions

In each example we assume the following initial conditions:

- *Objects names*: `obj1`, `obj2`,
- *Object radii*: `radius(obj1) = 1`, `radius(obj2) = 4`,
- *Initial shelf space*: `shelfSpace = 1`,
- *Initial configuration*: all objects are on the table (no initial stacks).

The goal in each example is to clear the open objects from the table by placing them on a shelf with limited space. In Example 1, the planner initially knows that both objects are open and, thus, can build a *linear* plan as a simple action sequence. In Examples 2 and 3, sensing actions are required: in the second example, the planner knows that one object is not open but does not know whether the second object is open or not; in the third example, the planner does not know whether either object is open or not.

When PKS constructs a plan that includes sensing actions, it can build into the plan a set of *conditional branches* for reasoning about the possible outcomes of a sensing operation. In particular, one branch is constructed for each possible value the sensed property might have. The resulting plans in this case are structured as trees rather than simple linear sequence of actions. In our examples, branch points are denoted by expressions like "`branch(open(objX))`," meaning "branch on the truth value of `open(objX)`." In this scenario, we will only consider branches on binary properties, i.e., properties that can be either true or false. A branch point is followed by two plan sections, labelled as "`K+`" and "`K-`," denoting two disjoint plan branches. The `K+` branch indicates the "knowledge positive" branch where `open(objX)` is assumed to be true. The `K-` branch indicates the "knowledge negative" branch where `open(objX)` is assumed to be false (i.e., ¬`open(objX)` is assumed to be true). Each branch can contain a sequence of actions and possibly other branch points. A `nil` tag along a branch indicates that no further operation takes place along that branch. At execution time, the information returned from a sensing action will let the plan execution monitor decide which branch of the plan it should follow at a branch point. The planner ensures that when conditional plans are constructed, the goals are achieved along every branch of the plan.

3.3.1 Example 1

Initial conditions: The planner initially knows `open(obj1)` and `open(obj2)` are true.

Plan

`graspA-fromTable(obj1)`
`putInto-objOnTable(obj1,obj2)`
`graspD-fromTable(obj2)`
`putAway(obj2)`

Since `obj1` and `obj2` are both initially known to be open the planner does not need to include any sensing actions in the plan. The two objects can simply be stacked and removed from the table.

3.3.2 Example 2

Initial conditions: The planner initially knows that `¬open(obj1)` is true but does not know the state of `open(obj2)`.

Plan

`sense-open(obj2)`
`branch(open(obj2))`
`K+:`
`  graspA-fromTable(obj2)`
`  putAway(obj2)`
`K-:`
`  nil`

Since the planner does not initially know whether `obj2` is open or not it includes a `sense-open` action in the plan. The plan then branches on the two possible outcomes of `open(obj2)`. If `open(obj2)` is true (the `K+` branch) then `obj2` is grasped and removed from the table; if `open(obj2)` is false (the `K-` branch) then no further action is taken. Since the planner initially knows that `obj1` is not open, this object does not need to be removed from the table.

3.3.3 Example 3

Initial conditions: The planner does not initially know the state of `open(obj1)` and `open(obj2)`.

Plan

```
sense-open(obj1)
sense-open(obj2)
branch(open(obj2))
K+:
     branch(open(obj1))
     K+:
          graspA-fromTable(obj1)
          putInto-objOnTable(obj1,obj2)
          graspD-fromTable(obj2)
          putAway(obj2)
     K-:
          graspA-fromTable(obj2)
          putAway(obj2)
K-:
     branch(open(obj1))
     K+:
          graspA-fromTable(obj1)
          putAway(obj1)
     K-:
          nil
```

Since the planner does not initially know whether `obj1` or `obj2` is open, it includes two `sense-open` actions in the plan. It then considers each possible outcome of these actions by constructing a plan with four branches (an initial branch point, followed by a second branch point along each of the top-level branches):

(i) Along the K+/K+ branch where `open(obj2)` and `open(obj1)` are true, both objects are grasped and put away as in Example 1.

(ii) Along the K+/K- branch where `open(obj2)` and ¬`open(obj1)` are true, object `obj2` is grasped and put away.

(iii) Along the K-/K+ branch where ¬`open(obj2)` and `open(obj1)` are true, object `obj1` is grasped and put away.

(iv) Along the K-/K- branch where ¬`open(obj2)` and ¬`open(obj1)` are true, no further action is taken.

Actions	
`pullCloser(?x)`	Pull an object `?x` closer to the robot.
`pullCloser-usingObject(?x,?y)`	Pull object `?x` closer to the robot using object `?y`.
`relocate-forGrasp(?x)`	Relocate object `?x` into a position that permits grasping.
Properties	
`extendsGripper(?x)`	A predicate indicating that object `?x` can be used to extend the robot's gripper.
`inExtendedRange(?x)`	A predicate indicating that object `?x` is in the range of the robot's extended gripper.
`inGraspablePosition(?x)`	A predicate indicating that object `?x` is in a graspable position.
`inRange(?x)`	A predicate indicating that object `?x` is in the range of the robot's ordinary gripper.

Table 5: Additional high-level actions and properties

4 Experimental Extensions to the Integration Domains

We have also defined a set of actions and properties that are not currently part of our integration domains, but could be added to either domain as possible extensions.

4.1 Pulling and relocating actions

Table 5 describes three new actions and four new properties we are currently experimenting with. These additions introduce a simple notion of object distance from the robot, and the requirement that objects be within the robot's reach before they can be manipulated. The `inRange` predicate describes an object as being close enough to the robot to be manipulated by its ordinary gripper, while `inExtendedRange` means an object is outside the ordinary gripper range but reachable using a simple tool (e.g., a stick or hook) that extends the gripper's range. The `pullCloser` action enables the robot to move an object closer to its workspace, with the effect that all objects stacked in that object are also dragged closer. For instance, if the top object in a stack is not within the robot's range but the base object of the stack is, the robot can pull the stack of objects closer in order to manipulate the top object. The `pullCloser-usingObject` action allows the robot to use certain objects in the domain as a gripper extension, to move objects in its "extended" range into its ordinary workspace. Finally, the `relocate-forGrasp` action allows the robot to move an object into a better position in its workspace that facilitates grasping (denoted by the predicate `inGraspablePosition`), for instance by nudging or pushing the object. We note that `inGraspablePosition` does not necessarily indicate that a grasp will actually succeed, but only that the positioning of the object (given its shape, orientation, etc.) does not prevent a grasp attempt.

A preliminary encoding of these actions is given in Table 6, however, there are still problems with our current representation. For instance, the definition of the action `pullCloser-usingObject` does not take into consideration how the "gripper exten-

Actions	Preconditions	Effects

pullCloser(?x)	inRange(?x)
	gripperEmpty
	onTable(?x)
	(∃ ?y).
	  ?y ≠ ?x
	  inStack(?y,?x)
	  inExtendedRange(?y)
		(∀ ?y).
		  inStack(?y,?x)
		  inExtendedRange(?y) ⇒
		  add(K_f,inRange(?y))
		  del(K_f,inExtendedRange(?y))

pullCloser-usingObject(?x,?y)	?x ≠ ?y inExtendedRange(?x) clear(?x) onTable(?x) inGripper(?y) extendsGripper(?y)	del(K_f,inExtendedRange(?x)) add(K_f,inRange(?x))
relocate-forGrasp(?x)	inRange(?x) gripperEmpty onTable(?x) clear(?x) ¬inGraspablePosition(?x)	add(K_f,inGraspablePosition(?x))

Table 6: Representation of additional high-level actions

sion" object has been grasped, only that it is in the gripper. One can imagine a more sophisticated representation where a specific grasp type must be applied to use an object "for pulling". We also do not currently take into consideration the actual length of the object used to extend the gripper, but instead only consider broad ranges. Furthermore, the pullCloser does not mention how an object is actually moved towards the robot; we must decide if this action requires a particular grasp type and whether an object should be grasped with an ordinary grasp action before being pulled closer.

We also note that relocate-forGrasp and inGraspablePosition are quite abstract, and are really generalised versions of actions like nudge-toEdge and properties like atEdge from our first integration domain. While this particular action and predicate combination may seem implausible as a robot-level reflex and sensor, we mention them to highlight the complex learning problem that must take place to move from primitive sensor data to an abstract action representation. In practice, such actions and properties would more likely be applied in particular contexts (like nudge-toEdge for flat objects).

4.2 Example plans

To illustrate the use of the above actions and properties, we give three short examples of planning problems we can solve. These examples assume that the actions in Table 6 have been combined with the action specifications in Table 4 from the SDU/UEDIN robot stacking scenario. (These actions can also be added to the KIT/UEDIN scenario with few changes required.) In each example we consider a domain with four objects, with the goal of removing all open objects from the table.

4.2.1 Example 1

Initial conditions: The planner initially knows that `obj1`, `obj2`, and `obj3` are all on the table and open. Object `obj4` is not open but can be used as a gripper extension. Object `obj3` is known to be outside the range of the gripper.

Plan
`graspA-fromTable(obj2)`
`putAway(obj2)`
`graspD-fromTable(obj4)`
`pullCloser-usingObject(obj3,obj4)`
`put-onTable(obj4)`
`graspA-fromTable(obj3)`
`putInto-objOnTable(obj3,obj1)`
`graspD-fromTable(obj1)`
`putAway(obj1)`

In this plan the robot first grasps and removes `obj2` from the table. It then uses `obj4` to pull `obj3` into its working space, before stacking `obj3` in `obj1` and removing the stacked objects from the table.

4.2.2 Example 2

Initial conditions: The planner initially knows that `obj1`, `obj2`, and `obj3` are all open, and that `obj4` is not open. Objects `obj1` and `obj2` are initially on the table. Object `obj3` is stacked in `obj1` but is outside the range of the gripper. Object `obj1` is within the range of the gripper however it can only be grasped using grasp type B.

Plan
`pullCloser(obj1)`
`graspA-fromTopOfStack(obj3)`
`putInto-objOnTable(obj3,obj2)`
`graspB-fromTable(obj1)`
`putAway(obj1)`
`graspD-fromTable(obj2)`
`putAway(obj2)`

In this plan the robot first pulls `obj1` closer, bringing `obj3` into its working space. Because `obj1` can only be grasped using grasp type B, the entire stack cannot simply be removed to the shelf. Instead, the robot must unstack `obj3`, stack `obj3` in `obj2`, and then remove `obj1` and `obj2` from the table. Object `obj4` plays no role in this plan.

4.2.3 Example 3

Initial conditions: The planner initially knows that `obj1`, `obj2`, and `obj3` are all on the table and open. Object `obj4` is not open but can be used as a gripper extension. Object `obj3` is known to be outside the range of the unextended gripper. Object `obj2` is not in a graspable position on the table.

Plan
`graspA-fromTable(obj1)`
`putAway(obj1)`
`graspD-fromTable(obj4)`
`pullCloser-usingObject(obj3,obj4)`
`put-onTable(obj4)`
`relocate-forGrasp(obj2)`
`graspA-fromTable(obj3)`
`putInto-objOnTable(obj3,obj2)`
`graspD-fromTable(obj2)`
`putAway(obj2)`

In this case, the plan directs the robot to remove `obj1` from the table. It then uses `obj4` to pull `obj3` into the range of the gripper, relocates `obj2` to a better position that facilitates grasping, then grasps `obj3` and stacks it in `obj2` before removing `obj2` from the table. (Alternatively, the planner could have constructed a plan that stacked `obj3` in `obj1` before removing `obj1` and `obj2` from the table.)

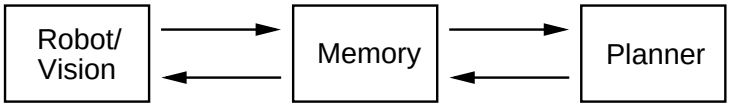


Figure 2: Flow of messages between the three system levels

5 Message Passing Protocol and Control Architecture

In this section we describe a simple domain-independent message passing protocol and control architecture for exchanging information between the low-level robot/vision, mid-level memory, and high-level planning components in the system. We begin by defining a set of messages that can be passed between the system levels. We then describe the structure of the control architecture, and provide details of a communication library supplied by UEDIN that implements our protocols. Both integration domains in this document currently use our message passing protocol and architecture which we believe is sufficiently general to support future domains on the PACO-PLUS robot platforms.

5.1 Message definitions

We define a set of 10 messages that capture the interactions between the three levels of the system. Each message is defined by its *type* and *content*. A message's type is simply its name or label. Depending on the message type, a message may also contain specific content or data to be sent. The message passing protocol we have defined is currently based on a *point-to-point* model, where each message is sent by a particular system component to another component. Moreover, the message set is designed in such a way that messages are (generally) defined in send/receive pairs so that only certain messages can be initiated by a "sending" level, with an appropriate response being sent by the "receiving" level. The current set of defined messages is given in Table 7 and the send/receive message pairs are given in Table 8.

5.2 Message passing control algorithms

The message passing protocol is initially driven by the robot/vision level of the system. Because of the paired send/receive nature of our message set, the upper system levels are forced to coordinate their operations in order to respond appropriately to lower-level messages. Currently, communication only takes place between two "adjacent" levels of the system, i.e., the robot and memory, or the memory and planner (see Figure 2). This means that all communication between the robot and planner must flow through the memory level, which typically acts as a forwarding service, but may also observe or refine the flow of messages (see below). Because the message passing protocol is mainly driven by the robot level, the memory and planning levels operate as message servers that respond to message queries. This protocol also permits certain message exchanges between the planner and memory levels that can interrupt the standard robot-driven process. It is also worth noting that nothing in the implementation of the communication architecture prevents us from expanding this protocol to permit direct point-to-point communication between any two components of the system.

Message Type	Description
MSG_STATE_UPDATE	Provide updated state information *Sender/Destination:* Robot to Memory, or Memory to Planner *Content:* World state specification
ACK_STATE_UPDATE	Acknowledge state update message *Sender/Destination:* Planner to Memory, or Memory to Robot *Content:* NONE
MSG_ACTION_REQUEST	Request a new action *Sender/Destination:* Robot to Memory, or Memory to Planner *Content:* NONE
ACK_ACTION_REQUEST	Acknowledge new action request for execution *Sender/Destination:* Planner to Memory, or Memory to Robot *Content:* NONE
MSG_ACTION_SUBMIT	Submit a new action for execution *Sender/Destination:* Planner to Memory, or Memory to Robot *Content:* Action specification
ACK_ACTION_SUBMIT	Acknowledge receipt of new action and start of action execution *Sender/Destination:* Robot to Memory, or Memory to Planner *Content:* NONE
MSG_ACTION_STOPPED	Provide alert that execution of last submitted action has stopped *Sender/Destination:* Robot to Memory, or Memory to Planner *Content:* Action execution return value (1 = success or 0 = failure)
ACK_ACTION_STOPPED	Acknowledge termination of last submitted action *Sender/Destination:* Planner to Memory, or Memory to Robot *Content:* NONE
MSG_PLAN_REQUEST	Request entire plan from planner *Sender/Destination:* Memory to Planner *Content:* NONE
MSG_PLAN_SUBMIT	Submit a complete plan *Sender/Destination:* Planner to Memory *Content:* Plan specification

Table 7: Message types defined in the message passing protocol

Message type sent	Expected response
MSG_STATE_UPDATE	ACK_STATE_UPDATE
MSG_ACTION_REQUEST	ACK_ACTION_REQUEST
MSG_ACTION_SUBMIT	ACK_ACTION_SUBMIT
MSG_ACTION_STOPPED	ACK_ACTION_STOPPED
MSG_PLAN_REQUEST	MSG_PLAN_SUBMIT

Table 8: Send/receive message pairs

5.2.1 Robot-level control loop

At the robot level, the message-processing control loop follows a simple structure where the robot essentially drives the message-passing process and the upper levels of the system respond to queries. The robot-level control loop defines a synchronous cycle where a message is sent and its acknowledgement is received before the next message can be sent. As a result, the robot only executes one action at a time and provides updates on the state of the world before the next action begins.

At an abstract level, the interaction between the robot and the higher levels follows the *RobotLevelControlLoop* pseudo code given in Figure 3(a). After an initial report on the world state, the main communication cycle consists of an action request by the robot, which is fulfilled by the upper levels (ultimately the planner), an indication from the robot when the action has finished executing, followed by an update on the new state of the world. Messages to and from the robot level all pass through the memory level. Thus, a request made by the robot for a planning-level service (e.g., requesting a new action) will ultimately reach the planner after being forwarded through the memory.

5.2.2 Memory-level control loop

Unlike the more tightly-regulated control loop of the robot level, communication at the memory level is more loosely structured using a client-server architecture. In particular, the memory is able to respond to requests from both the robot and the planner, as well as initiate certain messages of its own. The pseudo code for the memory-level control algorithm is given in Figure 3(b).

In most cases, the memory will initially act as a forwarding service that delivers messages from the robot to the planner, and messages from the planner to the robot. One possible extension for future work is the receipt of MSG_ACTION_SUBMIT messages from the planner. Before forwarding such messages, a mid-level component could inspect the message contents to check for sensing actions to be refined (as shown in Figure 3(b)). In the context of the SDU/UEDIN integration scenario described in this document, the memory could then transform all `sense-open` actions into *poke* or *focus* operations before passing them on to the robot. A similar approach could also be used to refine grasp operations specified by the planner. This protocol also supports a possible bottom-up

role for a memory components, where the middle level "abstracts" subsymbolic robot-level information into a symbolic form understandable by the planner.

The memory is also able to directly request information about the structure of a plan from the planner. The planner will respond with a complete description of the current plan, which may be a conditional plan with branches. The memory can then use this information as needed, for instance to refine a plan before passing it to the robot level.

5.2.3 Planning-level control loop

The planning level control loop also operates in a client-server fashion, responding to messages sent from the memory level (but typically originating from the robot level). The planning level is responsible for constructing high-level plans and feeding the actions, one at a time, to the robot level through the memory level. The planner also receives world state updates from the robot (again, through the memory) as well as status reports as to the success or failure of performed actions.

The memory level is also able to interact with the planner to request a complete description of the current plan. This part of the protocol provides the memory level with greater information about a plan's structure, which could be analyzed in order to help direct future operations of the memory level, or refine actions destined for the robot. Future versions of the communication protocol may also allow the planner to directly "push" such plan information to the lower levels, for instance as a result of replanning operations. The general planning-level control algorithm is given in Figure 3(c).

The message passing architecture we have outlined has a number of advantages. First, the protocol clearly separates the operations of the three system levels and the interactions between the levels, with the mid-level memory level acting as a form of mediator or interpreter. For instance, this protocol allows for the possibility of different content formats for messages flowing between the lower and upper levels of the system (e.g., messages exchanged between the robot and memory could contain subsymbolic information, while messages exchanged between the memory and planner could contain symbolic information). Also, changing the communication protocol for one pair of levels need not force changes to the interaction of another pair of levels. Finally, the message set has been designed to support more complex and flexible control architectures which may arise in the future. For our current integration tasks, however, the existing process is more than sufficient.

5.3 Socket communication library and sample code

For ease of implementation we have defined a set of C++ classes for manipulating message types and message contents. These classes work in conjunction with a lightweight socket library (also written in C++) that we have developed for Linux, to facilitate communication between system components.

At the code level, message types are chosen from a list of predefined `enum` types, and message contents are simple C++ strings. Currently, the content of `MSG_STATE_UPDATE` messages must be a list of instantiated properties from the list of available domain properties that form the world state. Similarly, the content of `MSG_ACTION_SUBMIT` messages must be a single instantiated action from the set of available domain actions. The content of the `MSG_PLAN_SUBMIT` message will be a plan similar to the example plans we have

```
Proc RobotLevelControlLoop
    Send: MSG_STATE_UPDATE; Receive: ACK_STATE_UPDATE;
    while !termination loop
        Send: MSG_ACTION_REQUEST; Receive: ACK_ACTION_REQUEST;
        Receive: MSG_ACTION_SUBMIT; Send: ACK_ACTION_SUBMIT;
        Send: MSG_ACTION_STOPPED; Receive: ACK_ACTION_STOPPED;
        Send: MSG_STATE_UPDATE; Receive: ACK_STATE_UPDATE;
    endLoop
endProc
```

(a)

```
Proc MemoryLevelControlLoop
    while !termination loop
        choose
            Send: MSG_PLAN_REQUEST;
        or
            Wait for message receive;
            case MSG_ACTION_SUBMIT:
                if action is sense-open then
                    Replace sense-open with poke or focus operation;
                endIf
                Forward message;
            case MSG_PLAN_SUBMIT:
                Update memory with received plan;
            case all other message types:
                Forward message;
        endChoose
    endLoop
endProc
```

(b)

```
Proc PlannerLevelControlLoop
    while !termination loop
        Wait for message receive;
        case MSG_STATE_UPDATE:
            Update world model;
            Send: ACK_STATE_UPDATE;
        case MSG_ACTION_UPDATE:
            Send: ACK_ACTION_REQUEST
            Construct plan/get next action in plan;
            Send: MSG_ACTION_SUBMIT; Receive: ACK_ACTION_SUBMIT;
        case MSG_ACTION_STOPPED:
            Process action success/failure;
            Send: MSG_ACTION_SUBMIT;
        case MSG_PLAN_REQUEST:
            Construct plan/get entire plan;
            Send: MSG_PLAN_SUBMIT;
    endLoop
endProc
```

(c)

Figure 3: Message passing control algorithms

seen earlier in the document, but encoded as a Prolog-style list (see Section 5.4 for an example). A plan iterator class is provided for inspecting the structure of conditional plans in this format. (For more details, refer to the sample code distributed with the socket library.)

For initial testing purposes the system terminates a plan by having the planner send a `MSG_ACTION_SUBMIT` message to the memory level in response to an action request, with the string `"EOP"` as its content. The memory level then passes this message to the robot. Both the memory and robot levels must then send a final `ACK_ACTION_SUBMIT` message to the level above, at which point all system levels are free to terminate communication. In the future, plan termination will force the suspension of the main control loop (i.e., the planner will not send an action) until a new goal is given to the planner and a new plan is constructed.

The communication library is distributed with a set of sample programs that implement the basic message passing protocol described in this document for the three levels of the system. These programs focus solely on the communication interface, with little additional functionality. (For instance, the memory level program simply forwards messages and always requests a complete plan after the first robot-level request for an action.) It is hoped that these programs will serve as the basis for developing more sophisticated modules that can simply be plugged into the communication architecture. A series of pregenerated plans are also included with this software, to test the message exchange process between the three levels.

Finally, we note that the current implemented version of the communication library defines a set of experimental message types for introducing new objects, new properties, and new actions into the planning-level domain description. While these messages currently have no effect on the communication protocol, these messages are reserved for future extensions to the architecture.

5.4 Message passing example

To better understand the flow of messages between the three system levels, we consider the scenario in Example 2 of Section 3.3, where the planner is given the goal of clearing the open objects from a table and constructs the conditional plan:

Plan

`sense-open(obj2)`
`branch(open(obj2))`
`K+:`
`  graspA-fromTable(obj2)`
`  putAway(obj2)`
`K-:`
`  nil`

Figure 4 shows the messages sent by all three levels during the execution of the action `sense-open(obj2)` in this plan (i.e., a complete cycle of the robot-level control loop).

We note that the first message sent by the robot, `MSG_STATE_UPDATE`, provides the planner with its initial description of the world. We assume that upon initialization the robot/vision system will send a complete world description, as a bootstrapping action.

From the perspective of the planning system this message is no more than a particularly large state update and requires no extra machinery.

Given an initial state description, the planner constructs a plan to achieve a given high-level goal. The planner sends the actions in this plan to the robot/vision system one step at a time, through the memory, in response to action requests. After the execution of each action the robot/vision system reports an update of the world state back to the planner, again, through the memory. In Figure 4 these updates are described in terms of state changes, however, we have agreed that state updates will initially include a complete (or as near as possible to complete) description of the new world state.

For many of the messages sent in this example, the memory level acts as a forwarding service between the robot and the planner. (In the future the memory could take on a more active role as a mediator or translator between the robot and planner.) One notable exception is the occurrence of the `MSG_ACTION_SUBMIT` message. Since the action specified in the content of this message is a sensing action, `sense-open(obj2)`, the example illustrates how the memory could refine this action by choosing between a *poke* and a *focus* operation. In this case, `focus(obj2)` is chosen as the refined action and the modified message is forwarded to the robot.

Figure 4 also illustrates the results of a `MSG_PLAN_REQUEST` message from the memory to the planner. In this case, the planner responds with a plan of the form:

```
[sense-open(obj2),branch(open(obj2),
            [graspA-fromTable(obj2),putAway(obj2)],[])].
```

This plan corresponds to the complete conditional plan given above, encoded in a Prolog-style list format for transmission using the communication library. (The communication library provides a helper class for processing plans in this compact format.)

We note that according to the message passing protocol, `MSG_PLAN_REQUEST` messages could be sent by the memory at other times during its control loop, or not at all, producing slightly different message orderings than those shown in Figure 4. (In the sample code the memory sends a `MSG_PLAN_REQUEST` after the first `MSG_ACTION_SUBMIT` message is received.) Similarly, alternate message orderings—including messages sent in parallel by different levels—could also arise since the robot, memory, and planner all run as independent processes. (E.g., message 13 could be sent at the same time as message 11, or even before it.) The implementation of the message passing protocol ensures that such ordering differences do not lead to problems like deadlock, however.

5.5 Reimplementation of the message passing protocol in ICE

At present, UEDIN continues to support the socket communication library outlined in this document and its implemention of the message passing protocol. However, as part of ongoing integration work with the ARMAR robot platform, KIT has reimplemented the PACO-PLUS message passing protocol using the Internet Communications Engine (ICE) middleware (`http://www.zeroc.com/ice.html`), as a means of facilitating the exchange of information between system levels and components. This software remains fully compatible with the message passing protocol described in this document, but may be adapted in the future. An overview of the current status of PKS integration on ARMAR using the ICE-based communication protocol is shown in [Petrick et al., 2010].

	Robot-level messages	Memory-level messages	Planner-level messages
1.	**MSG_STATE_UPDATE**: "onTable(obj1),...,!clear(obj1)"		
2.		(Forward to planner) **MSG_STATE_UPDATE**: "onTable(obj1),...,!clear(obj1)"	
3.			**ACK_STATE_UPDATE**
4.		(Forward to robot) **ACK_STATE_UPDATE**	
5.	**MSG_ACTION_REQUEST**		
6.		(Forward to planner) **MSG_ACTION_REQUEST**	
7.			**ACK_ACTION_REQUEST**
8.		(Forward to robot) **ACK_ACTION_REQUEST**	
9.			**MSG_ACTION_SUBMIT**: "sense-open(obj2)"
10.		*Refine* sense-open(obj2) *to* focus(obj2) (Forward to robot) **MSG_ACTION_SUBMIT**: "focus(obj2)"	
11.		(Send to planner) **MSG_PLAN_REQUEST**	
12.			**MSG_PLAN_SUBMIT**: "[sense-open(obj2), branch(open(obj2), [graspA-fromTable(obj2), putAway(obj2)], [])]"
13.	**ACK_ACTION_SUBMIT**		
14.		(Forward to planner) **ACK_ACTION_SUBMIT**	
15.	**MSG_ACTION_STOPPED**: "1"		
16.		(Forward to planner) **MSG_ACTION_STOPPED**: "1"	
17.			**ACK_ACTION_STOPPED**
18.		(Forward to robot) **ACK_ACTION_STOPPED**	
19.	**MSG_STATE_UPDATE**: "open(obj2)"		
20.		(Forward to planner) **MSG_STATE_UPDATE**: "open(obj2)"	
21.			**ACK_STATE_UPDATE**
22.		(Forward to robot) **ACK_STATE_UPDATE**	
23.	...	...	...

Figure 4: Example of messages passed during the execution of sense-open(obj2)

6 Related High-Level Integration Work

In this section we briefly describe a number of related integration tasks that are currently being investigated by UEDIN as part of WP4 and WP5.

6.1 Plan execution monitoring

Although we are able to construct plans for the proposed integration scenarios, a second high-level component is needed in order to monitor plan execution and control replanning/resensing activities. As part of WP4, UEDIN has built a *plan execution monitor* that is responsible for assessing both action failure and unexpected state information that result from feedback provided to the planner from the execution of planned actions at the robot level. The difference between predicted and actual states is used to decide between (i) continuing the execution of a plan, (ii) resensing activities that target a portion of a scene at a higher resolution to produce a more detailed state report, and (iii) replanning from new/unexpected states. In particular, rapid replanning techniques used by planners

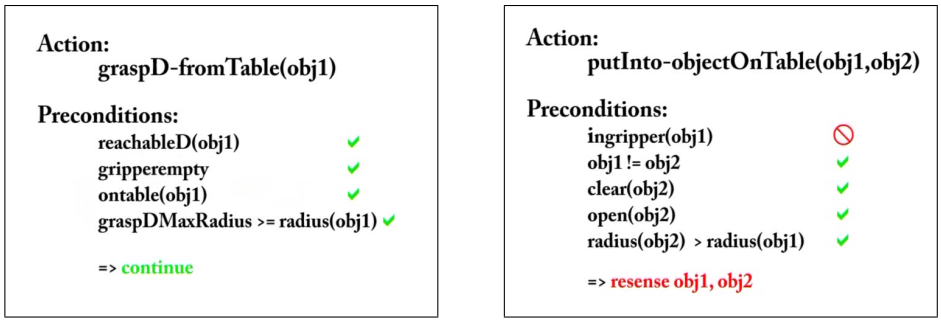


Figure 5: Plan execution monitoring in the SDU/UEDIN domain (screenshots from deliverable D8.1.3, SDU/UEDIN)

such as FF-Replan [Yoon et al., 2007] have been successfully employed in domains such as those in the probabilistic track of the International Planning Competition [Bryce and Buffet, 2008].

To aid in the assessment of (ii), the plan execution monitor provides the vision system with a list of the objects considered "relevant" to the execution of the action that is reported to have failed, based on the high-level action description. Using this information, the vision system can target particular parts of the scene with greater resolution in order to reevaluate the sensors that provide information about these objects. This operation may lead to new information about the world state.

For instance, Figure 5 shows the plan execution monitor being used to control execution in the SDU/UEDIN domain. The first image shows a scenario where the monitor decides the current action should proceed as planned: before applying the `graspD-fromTable(obj1)` action, the monitor verifies that the preconditions of the action are satisfied in the current state, i.e., `reachableD(obj1)`, `gripperempty`, `ontable(obj1)`, and `graspDMaxRadius >= radius(obj1)` all hold. The second image shows a scenario where the monitor is considering whether to apply the `putInto-objectOnTable(obj1,obj2)` action. In this case, all the preconditions are satisfied except for `ingripper(obj1)`. The monitor decides to resense the properties of the two objects contributing to the action, `obj1` and `obj2`. After resensing, the monitor will then decide whether to continue with the current action or replan entirely.

The plan execution monitor also has the task of managing the execution of conditional plans that contain sensing actions like `sense-open`. When a sensing action is ultimately executed at the robot level, the result of the sensing will be returned to the planner as part of the standard state update cycle (see Section 5). When faced with a conditional branch point in a plan, the plan execution monitor makes a decision as to the correct plan branch it should execute, based on the current state information. If such information is unavailable, for instance due to a failure at the robot/vision level, resensing or replanning activities are be triggered as above. It is important to note that the robot/vision system is never aware of the conditional nature of a plan, and never receives a "branch" operation like those shown in the example plans. From the point of view of the robot, it only receives a sequential stream of actions. This is also the case for any memory level components, except when a complete plan is requested. In such situations a fully-specified conditional plan is transmitted to the memory level.

An initial version of the high-level plan execution monitor for PKS has been build and

integrated into the SDU robot/vision system as part of the communication architecture [Petrick et al., 2009]. Currently, we are in the process of integrating the plan execution monitor with the ARMAR robot platform and the ICE-based communication infrastructure. Since our dialogue planner (see Section 6.3) is built on top of standard PKS, this integration is particularly important, as the plan execution monitor will provide execution control for both task and dialogue planning in the integrated system.

6.2 High-level action learning in robot domains

In previous work [Mourão et al., 2008] reported in WP5, and included in deliverable D5.1.2, we describe a mechanism for learning STRIPS-style actions effects from world state snapshots of the form produced by the control architecture in Section 5. In particular, this work addresses the *accommodation learning problem* at the planning level, i.e., learning the prediction function or action mapping between states. (E.g., see the function T in the PACO-PLUS Object-Action Complex (OAC) definition provided as part of WP1 and also reported elsewhere in related deliverables.)

Using machine learning techniques to learn action models is not a new idea. Prior approaches have applied a variety of techniques including inductive learning [Wang, 1995], directed experimentation [Gil, 1994], logical inference [Shahaf and Amir, 2006], heuristic search [Pasula et al., 2007], and support vector machines (SVMs) [Doğar et al., 2007].

Our approach differs from previous approaches. We use *kernel perceptron learning* [Aizerman et al., 1964, Freund and Shapire, 1999], combined with *deictic referencing* [Pasula et al., 2007] which reduces the complexity of our representation and, hence, the learning problem. As a result, we believe this technique will also allow our approach to scale to larger problem instances. Initial experiments using data simulated from the SDU/UEDIN integration domain have shown our approach to be quite efficient at learning action effects in this domain, resulting in high quality models with low error rates. This work also illustrates how a high-level action representation, usable by a planner like PKS, can be learnt (rather than preprogrammed) from data generated through a robot's interaction with the world.

In more recent work, our learning mechanism was also tested with noise-tolerant variants, including SVMs, kernel perceptrons with margins, and the voted perceptron which produced the best performance. [Mourão et al., 2009, 2010] details extensions of this learning approach to noisy and partially observable domains, with scalability experiments using planning scenarios from the International Planning Competition (e.g., BlocksWorld, Depots, and ZenoTravel).

Recent work has also focused on learning action models from real robot domains. In conjunction with SDU, we have collected real state data from the SDU robot/vision environment and the object stacking scenario described in Section 3. To collect the needed data, the integration scenario was tested by initially disallowing stacks of objects. Instead, objects could only be grasped from the table, put onto the table or shelf, or sensed for openness. Once initial data was collected, stacks of height 2 were permitted but the early learnt models were used to prune the state space, by reducing the number of no-op and failure-prone actions. Initial cross-validation results indicate a 2-4% error on individual features, with an overall correctness of 80% on full state predictions. Additional analyses on this data are planned to assess the quality of the action rules induced in such real-world environments.

6.3 Dialogue planning for language and communication

We have primarily focused on robot-planner integration in this document, with an emphasis on standard task planning. As outlined in the objectives of workpackage WP5, however, the mechanisms supporting the symbolic representation of actions and the ancillary planning apparatus will be generalised to language and communication. In particular, support for dialogue planning using speech acts will be built on top of the standard planning interface provided by PKS for ordinary task planning in PACO-PLUS. (Deliverable D5.1.3 contains a description of recent papers associated with this work.)

6.3.1 Motivation and background

The ability to *plan* is essential for an intelligent agent acting in a dynamic and incompletely known world: generating a sequence of actions that changes the world to bring about certain goal conditions often requires complex forward deliberation about the knowledge it requires, and the effects of the agent's actions. Agents that act in the world through natural language *dialogue* with other agents often face similar challenges: a speaker tries to change the mental state of the hearer for some purpose, by applying actions that correspond to the utterance of words or sentence. As a result, the task of choosing appropriate conversational moves has obvious parallels to planning, with both problems requiring the ability to reason about actions, beliefs, and goals.

The link between natural language and planning has a long tradition [Perrault and Allen, 1980, Appelt, 1985, Hovy, 1988, Young and Moore, 1994, Stone, 2000], from early approaches using structures like speech acts and discourse relations, and "beliefs, desires and intentions" (BDI)-based approaches [Litman and Allen, 1987, Bratman et al., 1988, Cohen and Levesque, 1990, Grosz and Sidner, 1990]. However, this work often suffered due to the inefficiency of the planning techniques available at the time.

Current mainstream research has instead tended to segregate standard task planning from dialogue planning, capturing the latter with more specialised approaches such as finite state machines, information state approaches, speech-act theories, dialogue games, or other rule-based approaches [Lambert and Carberry, 1991, Traum and Allen, 1992, Green and Carberry, 1994, Young and Moore, 1994, Chu-Carroll and Carberry, 1995, Matheson et al., 2000, Beun, 2001, Asher and Lascarides, 2003, Maudet, 2004].

Recently, there has also been a renewed interest in applying modern planning techniques to problems in natural language, including sentence generation [Koller and Stone, 2007], instruction giving [Koller and Petrick, 2008], accommodation [Benotti, 2008], as well as dialogue [Steedman and Petrick, 2007, Brenner and Kruijff-Korbayová, 2008]. However, experiments using off-the-shelf planners [Koller and Petrick, to appear] also demonstrate that while planning offers a promising alternative to current approaches, large problem instances can still be a challenge for the current generation of modern planners.

6.3.2 Dialogue planning as knowledge-level planning

Our approach to dialogue planning is motivated by the observation that certain types of dialogue actions can be modelled as instances of *sensing actions* or *information gathering actions*. In this view, the dialogue generation problem can instead be treated as an instance of the more generation AI problem of planning with incomplete information and sensing [Stone, 2000].

Plan 1	Plan 2
Go to the station	Go to the station
Buy a ticket	Buy a ticket
Check the departure board	**Ask someone for information**
Go to the track	Go to the track
Board the train	Board the train
...	...

Figure 6: Example of two plans for taking a train journey

For instance, Figure 6 shows two simple plans for taking a train journey. While most of the plan steps are identical, one important action is different in each plan: "check the departure board" in Plan 1, and "ask someone for information" in Plan 2. In the first case, the act of checking the departure board can be viewed as an observational action which returns information to the agent (e.g., the track number for a particular train). In the second case, asking someone for information is an example of a *speech act*, but one with a similar effect to checking the departure board: it also returns information to the agent. As a result, both actions serve as information gathering steps in the plan. This leads to the obvious questions: can we reason about dialogue acts in the same way as ordinary actions, and can we use the same machinery for planning such actions?

The problem of reasoning about incomplete information and sensing is very closely connected to the problem of reasoning about knowledge and action—a problem that has been well studied in the literature (e.g., [Moore, 1985, Scherl and Levesque, 1994, 2003, Stone, 1998], among others). Since we are interested in applying planning techniques to this problem, the PKS planner provides an appropriate starting point: PKS is designed around the concept of modelling the planner's knowledge state and how that knowledge state evolves during the plan generation process.

However, dialogue planning also seems to require additional machinery than is currently provided by the standard PKS planner. For instance, dialogues involve multiple participants, while PKS was designed as a single agent planner. Plans also involve mixed-initiative discourse among participants, with different participants performing actions that together can be seen as contributing to the overall "plan". Finally, if speech acts like *ask* and *tell* are treated as planning-level actions, does PKS support the correct level of representation in order to properly capture the effects of such actions? In other words, are knowledge-level planning techniques suitable for the kinds of dialogue contexts we'd like to consider?

In order to address the first two concerns, our solution is to (minimally) extend the PKS planner in order to introduce a notion of "agency" into the planner. Existing knowledge representation, reasoning, and plan generation mechanisms will all be extended to use this new feature, which will act as a type of "index" into otherwise unchanged PKS data structures and algorithms. In order to address the third concern, we plan to test our planner in a number of domains, starting with the PACO-PLUS KIT kitchen domain, but also considering other common benchmarks from the dialogue system literature (e.g., the modem troubleshooting domain), in order to properly evaluate the feasibility and effectiveness of our approach.

6.3.3 Extending PKS for dialogue planning

Dialogue participants

A simple notion of agency is introduced into PKS by using labels (modalities) for referencing dialogue participants and a special kind of knowledge container called *common ground*. In particular, we use labels of the following form:

[S]	Speaker supposition
[H]	Hearer supposition
[X], [Y], ...	Other participant/agent suppositions
[C_{XY}]	Common ground between X and Y

We permit such labels to be defined at the domain (i.e., user) level and customised to the particular context. These labels can also be nested in order to represent complex statements, e.g.,

[S] p	"The speaker supposes p."
[S] [H] p	"The speaker supposes the hearer supposes p."
[H] [C_{SH}] [S] p	"The hearer supposes it's common ground between the speaker and hearer that the speaker supposes p."

Knowledge assertions

Standard PKS provides a basic set of *knowledge assertions*, modelled through its database representation and primitive query language, that allow it to represent and reason about simple statements of the agent's knowledge. These assertions are not unique to PKS, but arise in many of the formal treatments of knowledge from the literature. In particular, PKS builds knowledge expressions based on the following three assertions:

Kp	"Know p",
$K_v t$	"Know the value of t",
$K_w p$	"Know whether p".

One of the strengths of PKS is its ability to model the effects of sensing actions. Using the K_w and K_w assertions, PKS can represent indefinite information (at plan time), of the kind returned by many common sensing actions.

To leverage this mechanism for dialogue planning, we allow knowledge assertions to be combined with agent labels, in order to express agent-indexed knowledge assertions, e.g.,

[S] $\neg K combo(safe) = c_1$	"The speaker doesn't know the combination of the safe is equal to c_1."
[S] [H] $K_v track$	"The speaker knows the hearer knows the value of the track (i.e., the track number)."
[S] [C_{SH}] $K_w open(boxA)$	"The speaker knows it's common ground between the speaker and the hearer that they know whether *boxA* is open."

Reasoning with labelled knowledge

While these minimal additions address many of the knowledge representation concerns we have previously considered, they do not specify how PKS can actually use such extended knowledge assertions, or infer certain conclusions from the agent labels. To do this, we adapt a series of rules for reasoning about speaker-hearer suppositions and

Action	Preconditions	Effects
$ask(\mathrm{X}, \mathrm{Y}, p)$	$\neg\,[\mathrm{X}]\,p$ $[\mathrm{X}]\,[\mathrm{Y}]\,p$	$add(K_f, [\mathrm{C_{XY}}]\,\neg\,[\mathrm{X}]\,p)$
$tell(\mathrm{X}, \mathrm{Y}, p)$	$[\mathrm{X}]\,p$ $[\mathrm{X}]\,\neg\,[\mathrm{C_{XY}}]\,p$	$add(K_f, [\mathrm{Y}]\,p)$

Figure 7: Example PKS dialogue actions for *ask* and *tell*

common ground modalities first introduced in [Steedman and Petrick, 2007]:

A1.	$[\mathrm{X}]\,\phi \Rightarrow \phi$	Supposition Veridicality
A2.	$[\mathrm{X}]\,\neg\phi \Rightarrow \neg\,[\mathrm{X}]\,\phi$	Supposition Consistency
A3.	$\neg\,[\mathrm{X}]\,\phi \Rightarrow [\mathrm{X}]\,\neg\,[\mathrm{X}]\,\phi$	Negative Introspection
A4.	$[\mathrm{C}]\,\phi \Leftrightarrow ([\mathrm{S}]\,[\mathrm{C}]\,\phi \wedge [\mathrm{H}]\,[\mathrm{C}]\,\phi)$	Common Ground
A5.	$[\mathrm{X}]\,[\mathrm{C}]\,\phi \Rightarrow [\mathrm{X}]\,\phi$	Common Ground Veridicality

Most of these rules are not unique to this work (e.g., A1, A2, and A3), but can be found in other formal representations of knowledge. Here, we require rules similar to these to augment PKS's standard inference procedure, however, we don't require them in full generality. Instead, we restrict the recursive depth in which we apply these rules (in particular, the depth of agent label reasoning) in order to improve the efficiency of the inference process. This restriction is also in line with PKS's standard inference procedure which is sound, but incomplete. (In practice, this restriction hasn't been a problem and we don't expect this to be problematic for the dialogue extensions.)

An important observation, however, is that the new reasoning rules we introduce into PKS are primarily required for reasoning about *knowledge*. With the exception of A4 and A5, which reason about common ground information, these rules do not encode any specific conversational rules, or rules for intent recognition. Instead, all reasoning is performed at the knowledge level, in line with the standard PKS approach.

Knowledge-level dialogue actions

The final step in extending PKS to dialogue planning involves adapting its representation of actions. In particular, we extend the standard PKS action description language to allow labelled knowledge assertions to be included as queries and database updates in standard PKS actions.

For instance, Figure 7 shows an example of two speech acts, *ask* and *tell*, encoded as extended PKS dialogue actions. In this case, the knowledge preconditions and effects are captured by a series of knowledge assertions, modelled using existing PKS databases and queries, and the new label mechanism. In the case of *ask* (i.e., "agent X asks agent Y about p"), the knowledge preconditions encode the constraints that "it's not the case that agent X knows p" (the first precondition) and "agent X knows that agent Y knows p" (the second precondition). The effects of *ask* are modelled by a single addition to the K_f database, namely that the planner knows "it's common ground between X and Y that X doesn't know p". In the case of *tell* (i.e., "agent X tells agent Y p"), we use a similar type of encoding. As preconditions, the action encodes that "X knows p" (the first precondition) and "X doesn't know p is common ground between X and Y" (the second precondition). The effects of *tell* are described by a single K_f update, encoding the fact that "Y knows p".

Robot1:	Let's make breakfast.	
Robot2:	Do you know where the milk is?	[ask(loc(milk))]
Robot1:	The milk is in the fridge.	[tell(loc(milk) = fridge)]
Robot2:	Is the cereal at the sideboard?	[ask(loc(cereal) = sideboard)]
Robot1:	No.	[tell(no)]
Robot2:	Where is the cereal?	[ask(loc(cereal))]
Robot1:	The cereal is in the cupboard.	[tell(loc(cereal) = cupboard)]
Robot2:	Okay. I suggest I go to the fridge, pickup the milk, bring it to the sideboard, then go the cupboard, pickup the cereal, and bring it to the sideboard.	[assert-plan(move(sideboard,fridge),...)]

Figure 8: Sample dialogue in the KIT kitchen domain

Generating dialogue plans

With the extended PKS action representation in place, plans can be constructed using dialogue acts, without changing the underlying plan generation mechanism. In particular, since plan generation simply involves database update, query, and inference, all changes to plan generation are encapsulated in these lower-level mechanisms, discussed above. As a result, we can build plans by chaining together actions, using our extra rules for representing and reasoning about agent labels, as a byproduct of the standard PKS plan generation process. (See [Steedman and Petrick, 2007] for a description of the type of reasoning that is performed during plan generation using speech acts like *ask* and *tell*.)

There are two interesting observations arising from this type of planning mechanism when applied to dialogue. First, plan generation takes place in the space of multi-agent plans. In particular, while we have a single planner (i.e., an agent generating a plan from that agent's perspective), the plan may contain actions that must be performed by other agents. However, in practice, the planner has no guarantee that other agents will actually *perform* those actions; those actions might simply be possible (or even probable) but are chosen purely from reasoning about the knowledge state of the planner, and it's beliefs about the knowledge states of others. As a result, we see this mechanism working in conjunction with a plan execution monitor that supports replanning, in order to construct new plans when the system detects that things have gone wrong. Second, one strong advantage of this approach is that both direct and indirect speech acts (and other linguistic phenomena, we expect) can be generated from the same mechanisms for reasoning about knowledge and common ground. As mentioned above, no specific conversational rules are encoded into the approach. Thus, this approach remains general purpose like other domain independent planners.

6.3.4 Testing domain and current state of integration in PACO-PLUS

In PACO-PLUS, we have focused on generating plans in the KIT kitchen domain described in Section 2. In particular, we are interested in building plans that require the agent to engage in dialogue with another agent to gather information that fills in the "gaps" that exist in its own knowledge. This can be seen as a form of collaborative knowledge exchange, leading to the construction of a task plan to be executed by one of the agents in the kitchen environment. For instance, Figure 8 shows an example of the kinds of dialogues we have been considering in PACO-PLUS.

At present, we have implemented a prototype version of the PKS-extended dialogue planner, using the mechanisms described above, and are currently in the process of in-

tegrating it into the PACO-PLUS communication architecture, for inclusion on the ARMAR robot platform. To date, we have only performed limited off-line testing using the KIT kitchen domain description. For instance, the following example is a conditional plan for finding the milk in the kitchen, by querying another agent for its location, retrieving the milk, and then bringing it to the sideboard:

```
ask-location(robot2,robot1,milk)
tell-location(robot1,robot2,milk)
move(sideboard,location(milk))
branch(location(milk))
    K(location(milk) = fridge):
        open-partial(fridge,lefthand)
        open-complete(fridge,righthand)
        remove-from(milk,fridge,righthand)
        close(fridge,lefthand)
    K(location(milk) = stove):
        grasp(milk,stove,righthand)
    ...
move(location(milk),sideboard)
put-down(milk,sideboard,righthand).
```

Once integration work has been completed, a comprehensive evaluation study is planned, to investigate the effectiveness of our approach in the PACO-PLUS demonstration domain. We also intend to apply our approach to other benchmark domains from the natural language dialogue systems literature.

Finally, an in-depth technical publication is planned for a future ICAPS and/or SIGDIAL conference, describing the planner extensions we have implemented and the results of our evaluation. However, we foresee this work extending beyond the end of PACO-PLUS.

7 Discussion

In this document we described two high-level action representations enabling goal-directed planning in low-level robot domains. Although the focus of our integration effort has shifted towards the KIT kitchen domain, which supports a more complex robot platform and real-world planning environment, our action descriptions, message passing protocol, and communication library also continue to support the SDU robot/vision platform at the present time. Furthermore, while we continue to extend our integration work for the final demonstrations of the PACO-PLUS systems, a number of important issues remain for future work beyond the end of PACO-PLUS.

1. All high-level grasp operators abstract the task of grasping into single action steps. We may extend the planner's representation to provide "finer-grained" actions that split the act of grasping into a sequence of steps like `positionForGraspA(obj1)`, `graspA-fromTable(obj1)`, `lift(obj1)`. Such actions would provide more detailed execution instructions to the robot system and, on failure, the robot system could more accurately indicate to the planner the specific aspect of the grasp that failed. Initially such sequences could be generated by simply "macro-expanding"

certain actions (like grasps) in a plan.

2. The execution of the high-level sensing action `sense-open` requires the implementation of a robot-level test that determines the openness of a particular object. For instance, the robot could perform a "poke" operation that attempts to determine the concavity of the object, or a more vision-based "focus" operation to study the object at a higher resolution. The test should not be part of the ordinary sensor report produced by the robot, but should instead be a special demand-driven operation. As discussed earlier in the document, this may also be a good place for the inclusion of mid-level processes to guide the choice of refinement operations. We could also consider similar refinements for grasp actions and generate plans with abstract actions like `grasp(obj1)`, leaving the choice of more specific robot-level actions like `graspA(obj1)` or `graspD(obj1)` to lower system levels.

3. There are many places where incomplete world state information can be introduced into the system, resulting from the interaction of the robot in a real-world setting. Thus, we must always be aware of the limitations of the system's capabilities, and the traditional AI assumption that we have complete models of the state changes resulting from executed actions. In PACO-PLUS, we have tried to ensure that our action models and state updates are as complete and correct as possible. However, this remains an area for future work beyond the end of PACO-PLUS.

4. A more complex interaction between the robot, memory, and planning levels might be desirable in the future. For instance, the planning level may require the ability to terminate an action during its execution if it has an undesirable outcome, or alert the memory about a replanning operation. This would require a more asynchronous architecture, including state update messages from the robot during action execution, as well as the ability to issue halt commands from the planning level. We also see the possibility of a more comprehensive "bottom-up" role for the memory level, as an abstraction component that mediates between the robot/vision level and the high-level planner. Such extensions should not require a significant reworking of the message passing protocol.

5. We also envision a more significant extension to the message passing protocol to support the addition of new objects, properties, and actions (i.e., "the birth of an object/property/action") into the high-level planning representation as a result of memory-level reasoning. Partial support for such messages already exists in the socket library, however, future versions of the message passing protocol must more fully specify these new message types.

6. Integration continues on the ARMAR platform to incorporate a new version of the basic PKS planner, the extended dialogue planner, and the plan execution monitor. Significant testing is also needed to properly evaluate the feasibility and effectiveness of these new modules in real-world contexts.

Acknowledgements

This document builds on discussions between PACO-PLUS partners from SDU, UL, ULg, and UEDIN at a meeting held in Leiden in September 2007. It also contains the results of more recent discussions with KIT. Special thanks go to Nils Adermann for his

help with the ARMAR robot platform and KIT kitchen environment, and Dirk Kraft for his contributions to integration and testing on the SDU side.

References

M. Aizerman, E. Braverman, and L. Rozonoer. Theoretical foundations of the potential function method in pattern recognition learning. *Automation and Remote Control*, 25: 821–837, 1964.

D. Appelt. *Planning English Sentences*. Cambridge University Press, Cambridge, England, 1985.

T. Asfour, K. Regenstein, P. Azad, J. Schröder, N. Vahrenkamp, and R. Dillmann. ARMAR-III: An Integrated Humanoid Platform for Sensory-Motor Control. In *Humanoids*, 2006.

T. Asfour, P. Azad, N. Vahrenkamp, K. Regenstein, A. Bierbaum, K. Welke, J. Schröder, and R. Dillmann. Toward humanoid manipulation in human-centred environments. *Robotics and Autonomous Systems*, 56(11):54–65, 2008.

N. Asher and A. Lascarides. *Logics of Conversation*. Cambridge University Press, Cambridge, 2003.

L. Benotti. Accommodation through tacit sensing. In *Proceedings of the 12th Workshop on the Semantics and Pragmatics of Dialogue*, pages 75–82, London, United Kingdom, 2008.

R.-J. Beun. On the generation of coherent dialogue. *Pragmatics and Cognition*, 9:37–68, 2001.

M. Bratman, D. Israel, and M. Pollack. Plans and resource-bounded practical reasoning. *Computational Intelligence*, 4:349–355, 1988.

M. Brenner and I. Kruijff-Korbayová. A continual multiagent planning approach to situated dialogue. In *Proceedings of the 12th Workshop on the Semantics and Pragmatics of Dialogue*, London, United Kingdom, 2008.

D. Bryce and O. Buffet. The uncertainty part of the 6th international planning competition. *http://ippc-2008.loria.fr/wiki/*, 2008.

J. Chu-Carroll and S. Carberry. Response generation in collaborative negotiation. In *Proceedings of ACL-95*, pages 136–143. ACL, 1995.

P. Cohen and H. Levesque. Rational interaction as the basis for communication. In P. Cohen, J. Morgan, and M. Pollack, editors, *Intentions in Communication*, pages 221–255. MIT Press, Cambridge, MA, 1990.

M. R. Doğar, M. Çakmak, E. Uğur, and E. Şahin. From primitive behaviors to goal directed behavior using affordances. In *Proc. of IROS 2007*, 2007.

O. Etzioni, S. Hanks, D. Weld, D. Draper, N. Lesh, and M. Williamson. An approach to planning with incomplete information. In *Proc. of KR-92*, pages 115–125, 1992.

R. Fikes and N. Nilsson. STRIPS: a new approach to the application of theorem proving to problem solving. *AI Journal*, 2:189–208, 1971.

Y. Freund and R. Shapire. Large margin classification using the perceptron algorithm. *Machine Learning*, 37:277–296, 1999.

Y. Gil. Learning by experimentation: Incremental refinement of incomplete planning domains. In *Proceedings of the International Conference on Machine Learning (ICML-94)*. MIT Press, 1994.

N. Green and S. Carberry. A hybrid reasoning model for indirect answers. In *Proceedings of ACL-94*, pages 58–65. ACL, 1994.

B. Grosz and C. Sidner. Plans for discourse. In P. Cohen, J. Morgan, and M. Pollack, editors, *Intentions in Communication*, pages 417–444. MIT Press, Cambridge, MA, 1990.

E. Hovy. *Generating natural language under pragmatic constraints*. Lawrence Erlbaum Associates, Hillsdale, NJ, USA, 1988.

A. Koller and R. Petrick. Experiences with planning for natural language generation. In *SPARK 2008 Workshop at ICAPS 2008*, Sept. 2008.

A. Koller and R. P. A. Petrick. Experiences with planning for natural language generation. *Computational Intelligence, Special Issue on Scheduling and Planning Applications*, to appear.

A. Koller and M. Stone. Sentence generation as planning. In *Proceedings of the 45th Annual Meeting of the Association of Computational Linguistics*, pages 336–343, Prague, Czech Republic, 2007.

D. Kraft, N. Pugeault, E. Başeski, M. Popović, D. Kragić, S. Kalkan, F. Wörgötter, and N. Krüger. Birth of the object: Detection of objectness and extraction of object shape through object action complexes. *International Journal of Humanoid Robotics (IJHR)*, 5(2):247–265, 2008.

L. Lambert and S. Carberry. A tripartite plan-based model of dialogue. In *Proceedings of ACL-91*, pages 47–54. ACL, 1991.

D. Litman and J. Allen. A plan recognition model for subdialogues in conversation. *Cognitive Science*, 11:163–200, 1987.

C. Matheson, M. Poesio, and D. Traum. Modeling grounding and discourse obligations using update rules. In *Proceedings of NAACL 2000, Seattle*, 2000.

N. Maudet. Negotiating language games. *Autonomous Agents and Multi-Agent Systems*, 7:229–233, 2004.

R. Moore. A formal theory of knowledge and action. In J. Hobbs and R. Moore, editors, *Formal Theories of the Commonsense World*, pages 319–358. Ablex, Norwood, NJ, 1985. Reprinted as Ch. 3 of Moore [1995].

R. Moore. *Logic and Representation*, volume 39 of *CSLI Lecture Notes*. CSLI/Cambridge University Press, Stanford CA, 1995.

K. Mourão, R. P. A. Petrick, and M. Steedman. Using kernel perceptrons to learn action effects for planning. In *Proc. of CogSys 2008*, pages 45–50, 2008.

K. Mourão, R. P. A. Petrick, and M. Steedman. Learning action effects in partially observable domains. In *Proceedings of the ICAPS 2009 Workshop on Planning and Learning*, pages 15–22, Thessaloniki, Greece, Sept. 2009.

K. Mourão, R. P. A. Petrick, and M. Steedman. Learning action effects in partially observable domains. In *Proceedings of the European Conference on Artificial Intelligence (ECAI 2010)*, Aug. 2010. To appear.

H. M. Pasula, L. S. Zettlemoyer, and L. P. Kaelbling. Learning symbolic models of stochastic domains. *Journal of Artificial Intelligence Research*, 29:309–352, 2007.

C. R. Perrault and J. F. Allen. A plan-based analysis of indirect speech acts. *American Journal of Computational Linguistics*, 6(3–4):167–182, 1980.

R. Petrick, N. Adermann, T. Asfour, M. Steedman, and R. Dillmann. Connecting knowledge-level planning and task execution on a humanoid robot using object-action complexes, Jan. 2010. Poster in the Proceedings of the International Conference on Cognitive Systems (CogSys 2010).

R. P. A. Petrick and F. Bacchus. A knowledge-based approach to planning with incomplete information and sensing. In *Proc. of AIPS-02*, pages 212–221, 2002.

R. P. A. Petrick and F. Bacchus. Extending the knowledge-based approach to planning with incomplete information and sensing. In *Proc. of ICAPS-04*, pages 2–11, 2004.

R. P. A. Petrick, D. Kraft, N. Krüger, and M. Steedman. Combining cognitive vision, knowledge-level planning with sensing, and execution monitoring for effective robot control. In *Proceedings of the Fourth Workshop on Planning and Plan Execution for Real-World Systems at ICAPS 2009*, pages 58–65, Thessaloniki, Greece, Sept. 2009.

R. B. Scherl and H. J. Levesque. The frame problem and knowledge-producing actions. Technical report, University of Toronto, Mar. 1994.

R. B. Scherl and H. J. Levesque. Knowledge, action, and the frame problem. *Artificial Intelligence*, 144(1–2):1–39, 2003.

D. Shahaf and E. Amir. Learning partially observable action schemas. In *Proceedings of the National Conference on Artificial Intelligence (AAAI-06)*. AAAI Press, 2006.

M. Steedman and R. P. A. Petrick. Planning dialog actions. In *Proc. of SIGdial 2007*, pages 265–272, 2007.

M. Stone. Abductive planning with sensing. In *Proceedings of AAAI-98*, pages 631–636, Menlo Park CA, 1998. AAAI.

M. Stone. Towards a computational account of knowledge, action and inference in instructions. *Journal of Language and Computation*, 1:231–246, 2000.

D. Traum and J. Allen. A speech acts approach to grounding in conversation. In *Proceedings of ICSLP-92*, pages 137–140, 1992.

X. Wang. Learning by observation and practice: An incremental approach for planning operator acquisition. In *Proc. of the International Conference on Machine Learning (ICML-95)*, pages 549–557, 1995.

S. Yoon, A. Fern, and R. Givan. FF-Replan: A baseline for probabilistic planning. In *Proceedings of the 17th International Conference on Automated Planning and Scheduling*, pages 352–359, 2007.

R. M. Young and J. D. Moore. DPOCL: a principled approach to discourse planning. In *Proceedings of the 7th International Workshop on Natural Language Generation*, pages 13–20, 1994.

Appendix B

DRAFT (Submitted version) - Final version to appear in *Computational Intelligence, Special Issue on Planning and Scheduling Applications*

Computational Intelligence, Volume 000, Number 000, 0000

Experiences with Planning for Natural Language Generation

Alexander Koller
Cluster of Excellence, Saarland University, Saarbrücken, Germany

Ronald P. A. Petrick
School of Informatics, University of Edinburgh, Edinburgh, UK

Natural language generation (NLG) is a major subfield of computational linguistics with a long tradition as an application area of automated planning systems. While current mainstream approaches have largely ignored the planning approach to NLG, several recent publications have sparked a renewed interest in this area. In this paper, we investigate the extent to which these new NLG approaches profit from the advances in planner expressiveness and efficiency. Our findings are mixed. While modern planners can readily handle the search problems that arise in our NLG experiments, their overall runtime is often dominated by the grounding step they perform as preprocessing. Furthermore, small changes in the structure of a domain can significantly shift the balance between search and preprocessing. Overall, our experiments show that the off-the-shelf planners we tested are unusably slow for nontrivial NLG problem instances. As a result, we offer our domains and experiences as challenges for the planning community.

Key words: natural language generation, planning

1. INTRODUCTION

Natural language generation (NLG; Reiter and Dale 2000) is one of the major subfields of natural language processing, concerned with computing natural language sentences or texts that convey a given piece of information to an audience. While the output of a generation task can take many forms, including written text, synthesised speech, or embodied multimodal presentations, the underlying NLG problem in each case can be modelled as a problem of achieving a (communicative) goal by successively applying a set of (communicative) actions. This view of NLG as goal-directed action has clear parallels to *automated planning*, which seeks to find general techniques for efficiently solving the action sequencing problem.

Treating generation as planning has a long history in NLG, ranging from the initial attempts of the field to utilise early planning approaches (Perrault and Allen 1980; Appelt 1985; Hovy 1988; Young and Moore 1994), to a recent surge of research (Steedman and Petrick 2007; Koller and Stone 2007; Brenner and Kruijff-Korbayová 2008; Benotti 2008) seeking to capitalise on the improvements modern planners offer in terms of efficiency and expressiveness. This paper attempts to assess the usefulness of current planning techniques to NLG by investigating some representative generation problems, and by evaluating whether automated planning has advanced to the point that it can provide solutions to such NLG applications—applications that are not currently being investigated by mainstream planning research.

To answer this question, we proceed in two ways. First, we present two generation problems that have recently been cast as planning problems: the sentence generation task and the GIVE task. In the sentence generation task, we concentrate on generating a single sentence that expresses a given meaning. In this case, a plan encodes the necessary sentence with the actions in the plan corresponding to the utterance of individual words (Koller and Stone 2007). In the GIVE domain ("Generating Instructions in Virtual Environments"), we describe a new shared task that was recently posed as a challenge for the NLG community (Byron et al. 2009). GIVE uses planning as part

[1] Address correspondence to koller@mmci.uni-saarland.de or rpetrick@inf.ed.ac.uk.

of a larger NLG system for generating natural-language instructions that guide a human user in performing a given task in a virtual environment.

Second, we evaluate the performance of several off-the-shelf planners on the planning domains into which these two generation problems translate. Among the planners we test, we explore the efficiency of FF (Hoffmann and Nebel 2001)—a planner that has arguably had the greatest impact on recent approaches to deterministic planning—and some of its descendants, such as SGPLAN (Hsu et al. 2006). All of the planners we test are freely available, support an expressive subset of the Planning Domain Definition Language (PDDL; McDermott et al. 1998), and have been successful on both standard planning benchmarks and the problems of the International Planning Competition (IPC).[1] Using these planners—together with an ad-hoc Java implementation of GraphPlan (Blum and Furst 1997) serving as a baseline for certain experiments—we perform a series of tests on a range of problem instances in our NLG domains.

Overall, our findings are mixed. On the one hand, we demonstrate that some planners can readily handle the *search* problems that arise in our testing domains on realistic inputs, which is promising given the challenging nature of these tasks (e.g., the sentence generation task is NP-complete; see Koller and Striegnitz 2002). On the other hand, these same planners often spend tremendous amounts of time on *preprocessing* to analyse the problem domain in support of the search. On many of our problem instances, the preprocessing time overshadows the search time. (For instance, FF spends 90% of its runtime in the sentence generation domain on preprocessing.) Furthermore, small changes in the structure of a planning domain can dramatically shift the balance between preprocessing and search. As a consequence, we are forced to conclude that the off-the-shelf planners we investigated are generally too slow to be useful in real NLG applications. It is also our hope, however, that these results will spark an interest to improve the quality of planner implementations—especially in the area of preprocessing techniques—and to this end we offer our domains and experiences as challenges for the planning community.

The remainder of this paper is structured as follows. In Section 2, we introduce the idea of NLG as planning and briefly review the relevant literature. In Section 3, we describe a set of planning problems associated with two NLG tasks: sentence planning and situated instruction generation. In Section 4, we report on our experiments with these planning problems. In Section 5 we discuss our results and overall experiences, and conclude in Section 6.

2. NLG AS PLANNING

The task of generating natural language from semantic representations (NLG) is typically split into two parts: the *discourse planning* task, which selects the information to be conveyed and structures it into sentence-sized chunks, and the *sentence generation* task, which then translates each of these chunks into natural language sentences. The sentence generation task is often divided into two parts of its own—the *sentence planning* task, which enriches the input by, e.g., determining object references and selecting some lexical material, and the *surface realization* task, which maps the enriched meaning representation into a sentence using a grammar. The chain of domain planning, sentence planning, and surface realization is sometimes called the "NLG pipeline" (Reiter and Dale 2000).

Viewing generation as a planning problem has a long tradition in the NLG literature. Perrault and Allen (1980) presented an approach to discourse planning in which the planning operators represented individual speech acts such as "request" and "inform". This idea was later expanded, e.g., by Young and Moore (1994). On the other hand, researchers such as Appelt (1985) and Hovy (1988) used techniques from hierarchical planning to expand a high-level plan consisting of speech acts into more detailed specifications of individual sentences. Although these systems covered some aspects of sentence planning, they also used very expressive logics designed to reason about beliefs and intentions, in order to represent the planning state and the planning operators. Most of these systems also used ad-hoc planning algorithms with rather naïve search strategies, which did not scale

[1] See `http://ipc.icaps-conference.org/` for information about past editions of the IPC. Also see (Hoffmann and Edelkamp 2005) for a good overview of the deterministic track of the 2004 competition.

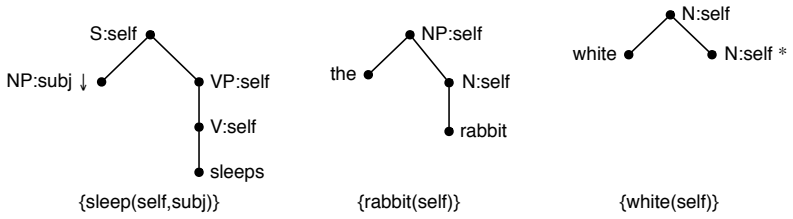


Figure 1: An example grammar in the sentence generation domain.

well to realistic inputs. As a consequence, the NLG-as-planning approach was mostly marginalized throughout the 1990s.

More recently, there has been a string of publications by various authors with a renewed interest in the generation-as-planning approach, motivated by the ongoing development of increasingly more efficient and expressive planners. For instance, Koller and Stone (2007) propose an approach to sentence generation (i.e., the sentence planning and surface realization modules of the pipeline) as planning—an approach we explore in more detail below (Section 3.1). Steedman and Petrick (2007) revisit the analysis of indirect speech acts with modern planning technology, viewing the problem as an instance of planning with incomplete information and sensing actions. In addition, Benotti (2008) uses planning to explain the accommodation of presuppositions, and Brenner and Kruijff-Korbayová (2008) use multi-agent planning to model the joint problem solving behaviour of agents in a situated dialogue. While these approaches focus on different issues compared to the 1980's NLG-as-planning literature, they all apply *existing*, well-understood planning approaches to linguistic problems, in order to utilise the rich set of modelling tools provided by modern planners, and in the hope that such planners can efficiently solve the hard search problems that arise in NLG (Koller and Striegnitz 2002). This paper aims to investigate whether existing planners achieve this latter goal.

3. TWO NLG TASKS

We begin by considering two specific NLG problems: sentence generation in the sense of Koller and Stone (2007), and the generation of instructions in virtual environments (Byron et al. 2009). In each case, we introduce the task and show by example how it can be viewed as a planning problem.

3.1. Sentence generation as planning

One way of modelling the sentence generation problem is to assume a *lexicalized grammar* in which each lexicon entry specifies how it can be combined grammatically with the other lexicon entries, what piece of meaning it expresses, and what the pragmatic conditions on using it are. Sentence generation can then be seen as constructing a grammatical derivation that is syntactically complete, respects the semantic and pragmatic conditions, and achieves all the *communicative goals*.

An example of such a lexicalized grammar is the tree-adjoining grammar (TAG; Joshi and Schabes 1997) shown in Figure 1. This grammar consists of *elementary trees* (i.e., the disjoint trees in the figure), each of which contributes certain *semantic content*. For instance, say that a knowledge base contains the individuals e, r_1 and r_2, and the facts that r_1 and r_2 are rabbits, r_1 is white and r_2 is brown, and e is an event in which r_1 sleeps. We could then construct a sentence expressing the information $\{\mathsf{sleep}(e, r_1)\}$ by combining instances of the elementary trees (in which the *semantic roles*, such as self and subj, have been substituted by constants from the knowledge base) into a TAG derivation as shown in Figure 2. In the figure, the dashed arrow indicates TAG's *substitution* operation, which "plugs" an elementary tree into the leaf of another tree; the dotted arrows stand for *adjunction*, which splices an elementary tree into an internal node. We can then read the sentence "The white rabbit sleeps" from the derivation. Note that the sentence "The rabbit sleeps" would not have been an appropriate result, because "the rabbit" could refer to either r_1 or r_2. Thus, r_2 remains as a *distractor*, i.e., an incorrect possible interpretation of the phrase.

This perspective on sentence generation also has the advantage of solving the sentence planning

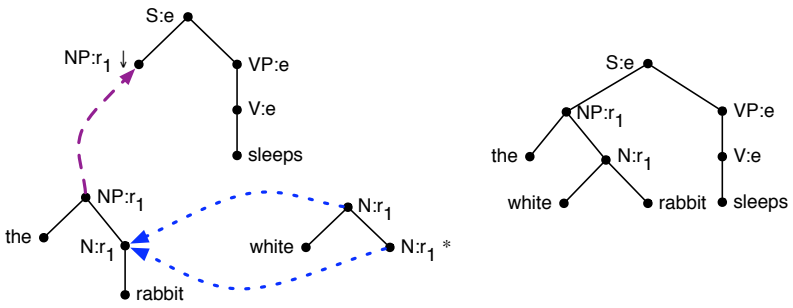


FIGURE 2: Derivation of "The white rabbit sleeps."

```
(:action add-sleeps
   :parameters (?u - node  ?xself - individual  ?xsubj - individual)
   :precondition
       (and (subst S ?u)  (referent ?u ?xself)  (sleep ?xself ?xsubj))
   :effect
       (and (not (subst S ?u))  (expressed sleep ?xself ?xsubj)
            (subst NP (subj ?u))  (referent (subj ?u) ?xsubj)
            (forall (?y - individual)
                (when (not (= ?y ?xself)) (distractor (subj ?u) ?y)))))

(:action add-rabbit
   :parameters (?u - node  ?xself - individual)
   :precondition
       (and (subst NP ?u)  (referent ?u ?xself)  (rabbit ?xself))
   :effect
       (and (not (subst NP ?u))  (canadjoin N ?u)
            (forall (?y - individual)
                (when (not (rabbit ?y)) (not (distractor ?u ?y))))))

(:action add-white
   :parameters (?u - node  ?xself - individual)
   :precondition
       (and (canadjoin N ?u)  (referent ?u ?xself)  (rabbit ?xself))
   :effect
       (forall (?y - individual)
           (when (not (white ?y)) (not (distractor ?u ?y)))))
```

FIGURE 3: PDDL actions for generating the sentence "The white rabbit sleeps."

and surface realization problems simultaneously, which is particularly useful in cases where these two problems interact. For instance, the generation of referring expressions (REs) is usually seen as a sentence planning task, however, syntactic information about individual words in available when the REs are generated (see, e.g., Stone and Webber 1998). (In the example, we require the referring expression "the white rabbit" to be resolved uniquely to r_1 by the hearer, in addition to the requirement that the derivation be grammatically correct.)

However, the problem of deciding whether a given communicative goal can be achieved with a given grammar is NP-complete (Koller and Striegnitz 2002): a naïve search algorithm that computes a derivation top-down takes exponential time and is clearly infeasible to use in practice. In order to circumvent this combinatorial explosion, the seminal SPUD system (Stone et al. 2003), which first established the idea of integrated TAG-based sentence generation, used a greedy, but incomplete, search algorithm. To better control the search, Koller and Stone (2007) recently proposed an alternative approach which converts the sentence generation problem into a planning problem, and solves

the transformed search problem using a planner (Koller and Stone 2007).[2] The resulting planning problem in this case assumes an initial state containing an atom $\mathsf{subst}(S, \mathsf{root})$, encoding the fact that a sentence (S) must be generated starting at the node named root in the TAG derivation tree, and a second atom $\mathsf{referent}(\mathsf{root}, e)$ which encodes the fact that the entire sentence describes the (event) individual e. The elementary trees in the TAG derivation are encoded as individual planning operators.

Figure 3 shows the transformed planning operators needed to generate the above example sentence, "The white rabbit sleeps." Here the action instance $\mathsf{add\text{-}sleeps}(\mathsf{root}, e, r_1)$ replaces the atom $\mathsf{subst}(S, \mathsf{root})$ with the atom $\mathsf{subst}(NP, \mathsf{subj}(\mathsf{root}))$. In an abuse of PDDL syntax, we write $\mathsf{subj}(\mathsf{root})$ as a shorthand for a fresh individual name.[3] At the same time, the operator records that the semantic information $\mathsf{sleep}(e, r_1)$ has now been expressed, and introduces all individuals except r_1 as distractors for the new RE at $\mathsf{subj}(\mathsf{root})$. These distractors can then be removed by subsequent applications of the other two operators. Eventually we reach a goal state, which is characterized by goals including $\forall x \forall y. \neg \mathsf{subst}(x, y)$, $\forall x \forall y. \neg \mathsf{distractor}(x, y)$, and $\mathsf{expressed}(\mathsf{sleep}, e, r_1)$. For instance, the following plan correctly performs the necessary derivation:

(1) $\mathsf{add\text{-}sleeps}(\mathsf{root}, e, r_1)$,
(2) $\mathsf{add\text{-}rabbit}(\mathsf{subj}(\mathsf{root}), r_1)$,
(3) $\mathsf{add\text{-}white}(\mathsf{subj}(\mathsf{root}), r_1)$.

The grammatical derivation in Figure 2, and therefore the generated sentence "The white rabbit sleeps," can be systematically reconstructed from this plan. Thus, we can solve the sentence generation problem via the detour through planning and bring current search heuristics for planning to bear on generation.

3.2. Planning in instruction giving

In the second application of planning in NLG, we consider the recent GIVE Challenge ("Generating Instructions in Virtual Environments"; Byron et al. 2009). The object of this shared task is to build an NLG system which produces natural language instructions which guide a human user in performing a task in a virtual environment. From an NLG perspective, GIVE makes for an interesting challenge since it is a theory-neutral task that exercises all components of an NLG system, and emphasizes the study of communication in a (simulated) physical environment. Furthermore, because the client displaying the 3D environment to the user can be physically separated from the NLG system (provided they are connected over a network), such systems can be cheaply evaluated over the Internet. This provides a potential solution to the long-standing problem of evaluating NLG systems. The first instalment of GIVE (GIVE-1) evaluated five NLG systems on the performance of 1143 users, making it the largest ever NLG evaluation effort to date in terms of human users.

Planning plays a central role in the GIVE task. For instance, consider the example GIVE world shown in Figure 4. In this world, the user's task is to pick up a trophy in the top left room. The trophy is hidden in a safe behind a picture; to access it, the user must push certain buttons in order to move the picture out of the way, open the safe, and open doors. The user must navigate the world and perform these actions in the 3D client; the NLG system must instruct the user on how to do this. To simplify both the planning and the NLG task, the world is discretised into a set of tiles of equal size. The user can turn by 90 degree steps in either direction, and can move from the centre of one tile to the centre of the next tile, provided the path between two tiles is not blocked. Figure 5 shows the encoding of some of the available GIVE domain actions in PDDL syntax. In the example, the shortest plan to solve the task consists of 108 action steps, with the first few steps as follows:

(1) $\mathsf{turn\text{-}left}(\mathsf{north}, \mathsf{west})$,
(2) $\mathsf{move}(\mathsf{pos\_5\_2}, \mathsf{pos\_4\_2}, \mathsf{west})$,
(3) $\mathsf{manipulate\text{-}button\text{-}off\text{-}on}(\mathsf{b1}, \mathsf{pos\_5\_2})$,

[2] See `http://code.google.com/p/crisp-nlg/` for the CRISP system, which implements this conversion.

[3] These terms are not valid in ordinary PDDL but can be eliminated by estimating an upper bound n for the plan length, making n copies of each action, ensuring that copy i can only be applied in step i, and replacing the term $\mathsf{subj}(u)$ in an action copy by the constant subj_i. The terms S, NP, and N are constants.

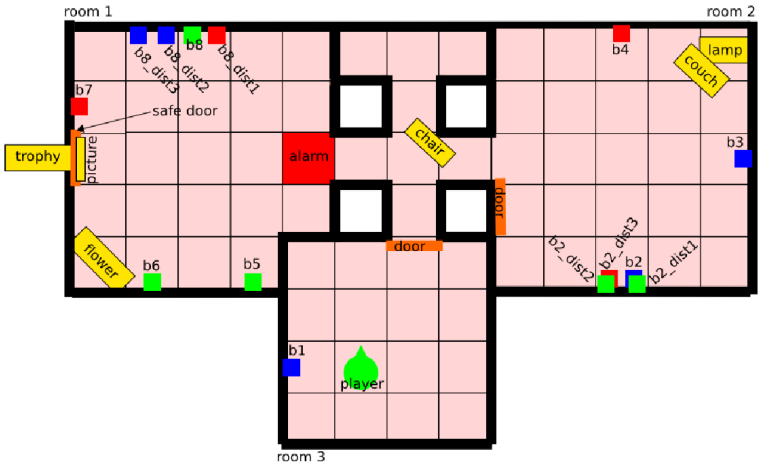


FIGURE 4: Map of an example GIVE world.

```
(:action move
   :parameters (?from - position  ?to - position  ?ori - orientation)
   :precondition
       (and (player-position ?from)  (player-orientation ?ori)
            (adjacent ?from ?to ?ori)  (not (alarmed ?to)))
   :effect
       (and (not (player-position ?from))  (player-position ?to)))

(:action turn-left
   :parameters (?ori - orientation  ?newOri - orientation)
   :precondition
       (and (player-orientation ?ori)  (next-orientation ?ori ?newOri))
   :effect
       (and (not (player-orientation ?ori))  (player-orientation ?newOri)))

(:action turn-right
   :parameters (?ori - orientation  ?newOri - orientation)
   :precondition
       (and (player-orientation ?ori)  (next-orientation ?newOri ?ori))
   :effect
       (and (not (player-orientation ?ori))  (player-orientation ?newOri)))

(:action manipulate-button-off-on
   :parameters (?b - button  ?pos - position  ?alarm - position)
   :precondition
       (and (state ?b off)  (player-position ?pos)  (position ?b ?pos)
            (controls-alarm ?b ?alarm))
   :effect
       (and (not (state ?b off))  (not (alarmed ?alarm))  (state ?b on)))
```

FIGURE 5: Simplified PDDL actions for the GIVE domain.

(4) turn-right(west, north).

Our description of GIVE as a planning problem makes it very similar to the classic Gridworld problem (see, e.g., Tovey and Koenig 2000 or the 1998 edition of the IPC[4]), which also involves route finding through a two-dimensional world map with discrete positions. As in Gridworld, the domain

[4]See `http://ftp.cs.yale.edu/pub/mcdermott/aipscomp-results.html`.

also requires the execution of certain object-manipulation actions (e.g., finding keys and opening locks in Gridworld, or pushing the correct buttons to open doors and the safe in GIVE). However, the worlds we consider in GIVE tend to be much bigger than the Gridworld instances used in the 1998 planning competition, with more complex room shapes and more object types in the world.

To be successful in GIVE, an NLG system must be able to compute plans of the form described above. At a minimum, a discourse planner will call a domain planner in order to determine the content of the instructions that should be presented to the user. This relatively loose integration of NLG system and planner is the state of the art of the systems that participated in GIVE-1. However, it is generally desirable to integrate the planner and the generation system more closely than this. For instance, consider an NLG system that wants to generate the instruction sequence "walk to the centre of the room; turn right; now press the green button in front of you". Experiments with human instruction givers (Stoia et al. 2008) show that this is a pattern that they use frequently: the instruction follower is made to walk to a certain point in the world where the instruction giver can then use a referring expression ("the green button") that is easy for the follower to interpret. An NLG system must therefore tightly integrate discourse planning and planning in the domain of the world map. On the one hand, the structure of the discourse is determined by the needs of the NLG system rather than the domain plan; on the other hand, the discourse planner must be aware of the way in which the instruction "turn right" is likely to change the visibility of objects. Even if an NLG system doesn't implement the generation of such discourse as planning, it must still solve a problem that subsumes the domain planning problem. For these reasons, we consider the GIVE domain planning problem as a natural part of a GIVE NLG system.

4. EXPERIMENTS

We now return to the original question of the paper: is planning technology ready for realistic applications in natural language generation? To investigate this question we consider two sets of experiments, designed to evaluate the performance of several planners on the NLG planning domains from the previous section. Starting with the CRISP domain, we first present a scenario which focuses on the generation of referring expressions with a tiny grammar (Section 4.1). We then look at a setting in which CRISP is used for surface realization with the XTAG Grammar (XTAG Research Group 2001), a large-scale TAG grammar for English (Section 4.2). In the second set of experiments we investigate the GIVE domain. We begin with a domain that is similar to the classic Gridworld (Section 4.3), and then add extra grid cells to the world that are not necessary to complete the task (Section 4.4). We also investigate the role that goal ordering plays in these problems. These experiments are configured in a way that lets us explore the scalability of a planner's search and preprocessing capabilities, and illustrate what we perceive to be one of the main limitations of current off-the-shelf planners for our applications: they often spend a long time computing ground instances, even when most of these instances are not required during plan search.

4.1. Experiment 1: Sentence generation (referring expressions)

For the first experiment on sentence generation, we exercise the ability of the CRISP system described in Section 3.1 to generate referring expressions. This problem is usually handled by the sentence planner if sentence planning and surface realization are separated; here it happens as part of the overall generation process.

We consider a series of sentence generation problems which require the planner to compute a plan representing the sentence "Mary likes the $\mathrm{Adj}_1 \ldots \mathrm{Adj}_n$ rabbit." Each problem instance assumes a target referent r, which is a rabbit, and a certain number m of further rabbits $r_1, \ldots, r_m$ that are distinguished by properties $P_1, \ldots, P_n$ with $n \leqslant m$. The problem instance is set up such that r has all properties except for P_i in common with each r_i for $1 \leqslant i \leqslant n$, and $r_{n+1}, \ldots, r_m$ have none of the properties P_i. That is, all n properties are required to describe r uniquely. The n properties are realized as n different adjectives, in any order. This setup allows us to vary the plan length (a plan with n properties will have length $n+4$) and the universe size (the universe will contain $m+1$ rabbit individuals in addition to the individuals used to encode the grammar, which have different types).

We converted these generation problem instances into planning problem instances as described

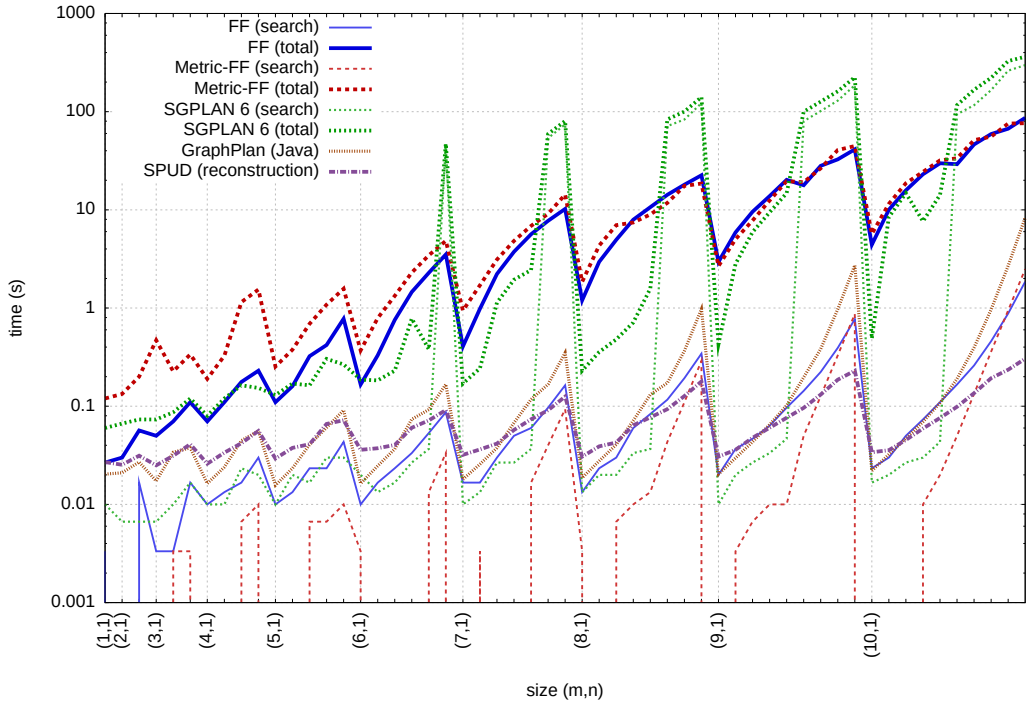


FIGURE 6: Results for the sentence generation domain. The horizontal axis represents parameters (m, n) from $(1, 1)$ to $(10, 10)$ in lexicographical order. The vertical axis is the runtime in milliseconds.

in Section 3, and then ran several different planners on them. We used three off-the-shelf planners: FF 2.3 (Hoffmann and Nebel 2001), Metric-FF (Hoffmann 2002), and SGPLAN 6 (Hsu et al. 2006); all of these were highly successful at the recent IPC competitions, and unlike many other IPC participants, support a fragment of PDDL with quantified and conditional effects, which is necessary in our domain. In addition, we used an ad-hoc implementation of GraphPlan (Blum and Furst 1997) written in Java; unlike the three off-the-shelf planners, this planner only computes instances of literals and operators as they are needed in the course of the plan search, instead of computing all ground instances in a separate preprocessing step. Finally, we reimplemented the incomplete greedy search algorithm used in the SPUD system (Stone et al. 2003) in Java.

The results of this experiment are shown in the graph in Figure 6.[5] The input parameters (m, n) are plotted in lexicographic order on the horizontal axis, and the runtime is shown in seconds on the vertical axis, on a logarithmic scale. These results reveal a number of interesting insights. First, the search times of FF and Metric-FF (shown as thinner lines) significantly outperform SGPLAN's search in this domain—on the largest instances, by a factor of over 100.[6] Second, FF and Metric-FF perform very similarly to each other, and their search times are almost the same as those of the SPUD algorithm, which is impressive because they are complete search algorithms, whereas SPUD's greedy algorithm is not.

Finally, it is striking that for all three off-the-shelf planners, the search only accounts for a tiny

[5] All runtimes in Sections 4.1 and 4.2 were measured on a single core of an AMD Opteron 8220 CPU running at 2.8 GHz, under Linux. FF 2.3 and Metric-FF were recompiled as 64-bit binaries and run with a memory limit of 32 GB. Java programs were executed under Java 1.6.0_13 in 64-bit mode and were allowed to "warm up", i.e., the JVM was given the opportunity to just-in-time compile the relevant bytecode by running the planner three times and discarding the runtimes before taking the actual measurements. All runtimes are averaged over three runs of the planners.

[6] For FF and Metric-FF, we report the "searching" and "total" times reported by the planners. For SGPLAN, we report the "total" time and the difference between the "total" and "parsing" times.

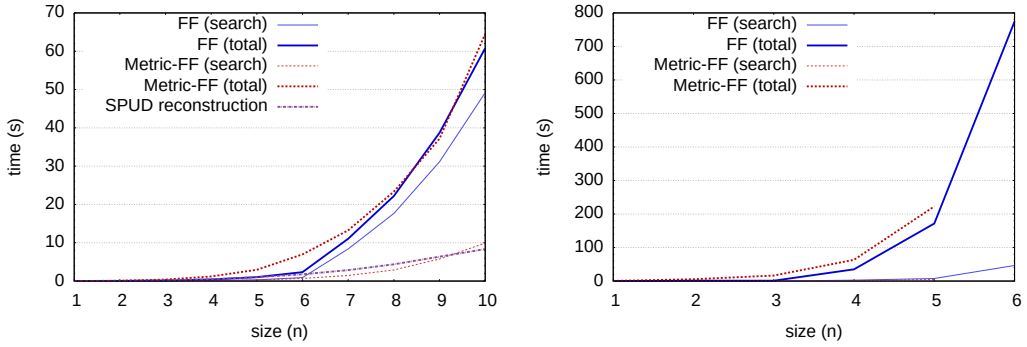


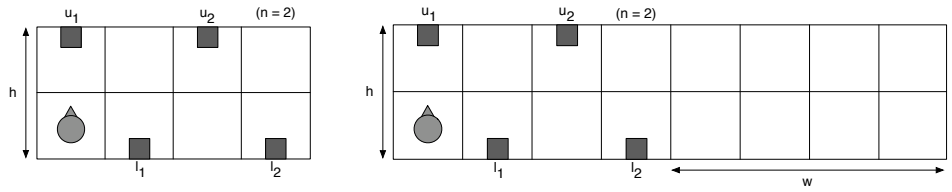
Figure 7: Results for the XTAG experiment, at $k = 1$ and $k = 2$.

fraction of the total runtime; in each case, the preprocessing times are higher than the search times by one or two orders of magnitude. As a consequence, even our relatively naïve Java implementation of GraphPlan outperforms them all in terms of total runtime, because it only computes instances by need. Although FF is consistently much faster as far as pure search time is concerned, our results indicate that FF's performance is much more sensitive to the domain size: if we fix $n = 1$, FF takes 27 milliseconds to compute a plan at $m = 1$, but 4.4 seconds to compute the same plan at $m = 10$. By comparison, our GraphPlan implementation takes 20 ms at $m = 1$ and still only requires 22 ms at $m = 10$.

4.2. Experiment 2: Sentence generation (XTAG)

The first experiment already gives us some initial insights into the appropriateness of planning for the sentence generation domain: on the examples we looked at, the search times were quite acceptable, but FF and SGPLAN spent a lot of time on the initial grounding step. However, one weakness of this experiment is that it uses a tiny grammar, consisting of just the 12 lexicon entries that are needed for the experiment. While the grounding problem can only get worse with larger grammars, the experiment by itself does not allow us to make clear statements about the efficiency of the search. To address this problem, we ran a second sentence generation experiment. This time, we used the XTAG Grammar (XTAG Research Group 2001), a large-scale TAG grammar for English. XTAG contains lexicon entries for about 17,000 uninflected words using about 1100 different elementary trees. Although XTAG does not contain semantic information, it is possible to automatically equip the lexicon entries with inferred semantic representations based on the words in the lexicalized elementary trees. The result is a highly ambiguous grammar: the most ambiguous word, "ask", is the anchor of 314 lexicon entries.

In our experiment, we were especially interested in two questions. First, how would the planners handle the search problem involved in generating sentences with such a large and ambiguous grammar? Second, would it be harder to generate sentences containing verbs with multiple arguments, given that verbs with more arguments lead to actions with more parameters and therefore more instances? To answer these questions, we generated sentences of the form "S and S and ... and S", where each S was a sentence, and n was the number of sentences in the conjunction. Each S was a sentence of the form "the businessman sneezes", "the businessman admires the girl", or "the businessman gives the girl the book"—that is, they varied in the number k of syntactic arguments the verb expects (1 for the intransitive verb "sneeze", 2 for the transitive verb "admire", and 3 for the ditransitive verb "give"). This means that the output sentence for parameters n and k contained $n(2k + 2) - 1$ words. The instances were set up in such a way that the generation of referring expressions was trivial, so this experiment was purely a surface realization task. To achieve reasonable performance, we only generated planning operators for those elementary trees for which all predicate symbols in the semantic representation also appeared in the knowledge base.



(a) Minimal GIVE world (b) GIVE world with extra grid cells

FIGURE 8: Experimental GIVE world configurations.

Figure 7 reports the runtimes we measured in this experiment for FF, Metric FF, and the SPUD reimplementation. We do not report runtimes for SGPLAN, because we could not recompile SGPLAN as a 64-bit binary, and the 32-bit version ran out of memory very quickly. We also do not report runtimes for our Java implementation of GraphPlan, because it was unusably slow for serious problem instances: for $k = 1$ and $n = 3$, it already took over two minutes and exceeded its memory limit of 16 GB for $n > 3$. However, we note that this may be a limitation of our naïve implementation rather than the GraphPlan algorithm itself.

Nonetheless, there are a number of observations we can make in this experiment. First, the experiment confirms that FF's Enforced Hill-Climbing search strategy works very well for the sentence generation task: although we are now generating sentences with a large grammar, FF produces a 39-word sentence ($k = 1, n = 6$) in under a second of search time. This level of efficiency is a direct result of using this particular search strategy: for $k = 1$ and $n > 6$, FF 2.3 (but not Metric-FF) fell back to the best-first search strategy, which causes a dramatic loss of search efficiency. It is also encouraging that Metric-FF still performs comparably to SPUD in terms of pure search time. We believe that FF's technique of evaluating actions by estimating the distance to a goal state for the relaxed problem essentially picks out the same evaluation function as SPUD's domain-specific heuristic, and the enforced hill-climbing strategy needs to backtrack very little in this domain and thus performs similarly to SPUD's greedy search. However, SPUD's incompleteness manifests itself in this experiment by its inability to find any plan for $k > 1$ and $n > 1$, whereas FF and its variants still (correctly) find these plans.

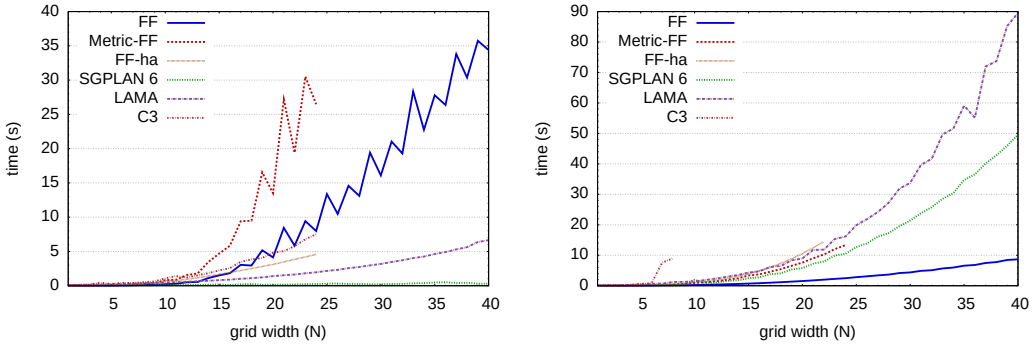
Second, FF's runtime is still dominated by the preprocessing stage. For instance, Metric-FF spends about 10 seconds on search for $k = 1, n = 10$, compared to its total runtime of about 65 seconds. This effect becomes more pronounced as we increase k: for $k = 2$, we reach 65 seconds of total runtime at $n = 4$, but here Metric-FF only spends about a second on search. For $k = 3$, neither FF nor Metric-FF were able to solve any of the input instances within their memory limit. This is consistent with the observation that the planning operators for the verbs have $k + 2$ parameters (see Fig. 3), and thus the number of action instances grows by a factor of the universe size every time we increase k by one. A planner which computes all ground instances of the operators thus takes exponential time in k for preprocessing.

4.3. Experiment 3: Minimal GIVE worlds

We now turn our attention to a set of experiments arising from the GIVE domain. Besides using many of the planners from the previous set of experiments (FF, Metric-FF, and SGPLAN), we also expand our testing to include the FF(h_a) (Keyder and Geffner 2008), LAMA (Richter and Westphal 2008), and C^3 (Lipovetzky et al. 2008) planners. Each of these additional planners competed in the deterministic "sequential, satisficing" track of the 2008 International Planning Competition; all planners performed well on the competition domains, with LAMA the overall winner of the track.[7]

In the first GIVE experiment, we construct a series of grid worlds, similar to the one illustrated in Figure 8(a). These worlds consist of a $N = 2n$ by h grid of positions, such that there are buttons at positions $(2i - 1, 1)$ and $(2i, h)$ for $1 \leqslant i \leqslant n$. The player starts in position $(1, 1)$ and must press all the buttons to successfully complete the game. (The actions in this domain are similar to the

[7]See `http://ipc.informatik.uni-freiburg.de/` for details of the 2008 IPC.



(a) Unordered (b) Ordered

Figure 9: Results for the unordered and ordered minimal GIVE domains with grid height $h = 20$. The horizontal axis is the grid width, N. The vertical axis is the total runtime in seconds.

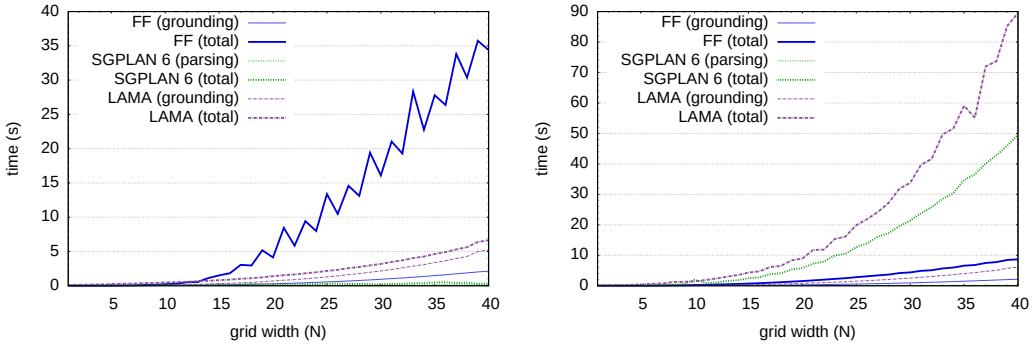
PDDL actions in Figure 5.) We consider two variants of this problem in our tests. In the *unordered* problem, the player is permitted to press the buttons in any order to successfully achieve the goal. In the *ordered* version of the problem, the player is unable to initially move to any grid cell containing a button, except for the cell containing the first button, u_1. Pressing u_1 releases the position of the next button, l_1, allowing the player to move into this cell. Similarly, pressing button l_1 frees button u_2, and so on. The end result is a set of constraints that forces the buttons to be pressed in a particular order to achieve the goal. As a concrete example, the following is a minimal plan (in either variant of the problem) for the case of a 2 by 2 grid with 2 buttons (i.e., $n = 1$, $h = 2$):

(1) move(pos_1_1, pos_1_2, north),
(2) manipulate-button-off-on(u1, pos_1_2),
(3) turn-right(north, east),
(4) move(pos_1_2, pos_2_2, east),
(5) turn-right(east, south),
(6) move(pos_2_2, pos_2_1, south),
(7) manipulate-button-off-on(l1, pos_2_1).

Results for the $h = 20$ case, with the grid width N ranging from 1 to 40, are shown in Figure 9. In the unordered case (Figure 9(a)), the most obvious result is that some of the planners tested—Metric-FF, FF(h_a), and C^3—are unable to solve any problems beyond $N = 24$ on our experimentation machine within the memory limit of 2 GB.[8] While FF, LAMA, and SGPLAN are able to solve all problem instances up to $N = 40$, the total runtime varies greatly between these planners. For instance, FF takes almost 35 seconds to solve the $N = 40$ problem, while LAMA takes around 6.5 seconds. SGPLAN shows impressive performance on $N = 40$, generating a 240 step plan in well under a second. In the ordered case (Figure 9(b)), we again have the situation where Metric-FF, FF(h_a), and C^3 are unable to solve all problem instances. Furthermore, both SGPLAN and LAMA, which performed well on the unordered problem, now perform much worse than FF: FF takes 39 seconds for the $N = 40$ case, while SGPLAN takes 50 seconds and LAMA takes 90 seconds. In real NLG systems, where response time is essential, runtimes over a few seconds are unacceptable.

Preprocessing time (i.e., parsing, grounding, etc.) generally plays less of a role in GIVE, compared with the sentence generation domain; however, its effects still contribute significantly to the overall runtime of a number of planners. Figure 10 shows the grounding time for FF, LAMA, and SGPLAN on the minimal GIVE problems, compared with the total runtime. In the unordered variant of the minimal GIVE domain (Figure 10(a)), the grounding time in LAMA and SGPLAN accounts

[8] All runtimes in Sections 4.3 and 4.4 were measured on a single core of an Intel Xeon CPU running at 3GHz, under Linux. All runtimes are averaged over three runs of the planners. Only 32-bit versions of the planners were used for testing in each case.



(a) Unordered (b) Ordered

FIGURE 10: Comparison of the total runtime and grounding time for selected planners in the $h = 20$ minimal GIVE domain. The horizontal axis is the grid width, N. The vertical axis is the runtime in seconds.

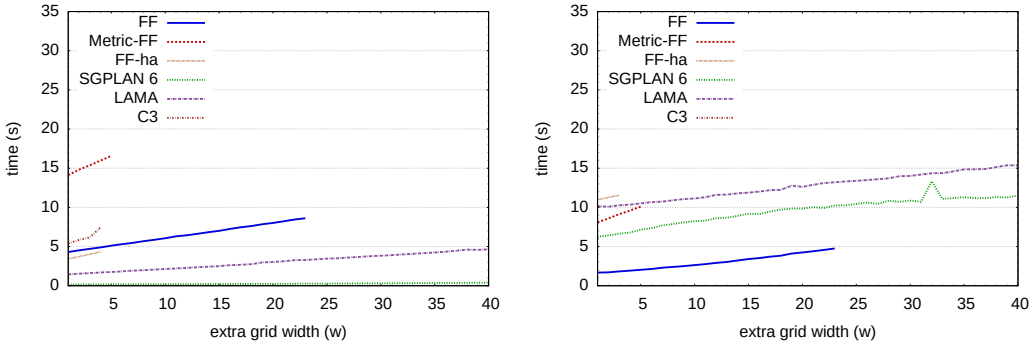
for a significant fraction of the total runtime: SGPLAN spends around 40% of its total runtime on preprocessing; for LAMA, this number rises to at least 80% for our test problems. For FF, the preprocessing time is much less important than the search time, especially for large problem instances. In the ordered case (Figure 10(b)), the actual time spent on preprocessing is essentially unchanged from the unordered case, and search time dominates the total runtime for all three planners. Overall, however, FF is now much better at controlling the search, compared with the other planners and its performance on the unordered variant of the problem.

4.4. Experiment 4: GIVE worlds with extra grid cells

In our last set of experiments, we vary the structure of the GIVE world in order to judge the effect that universe size has on the resulting planning problem. Starting with the GIVE world described in Experiment 3, we extend the world map by adding another w by h empty cell positions to the right of the minimal world, as shown in Figure 8(b). These new positions are not actually required in any plan, but extend the size of the state space and approximate the situation in the actual GIVE domain where most grid positions are never used. We leave the initial state and goal untouched and, again, consider both unordered and ordered variants of the problem.

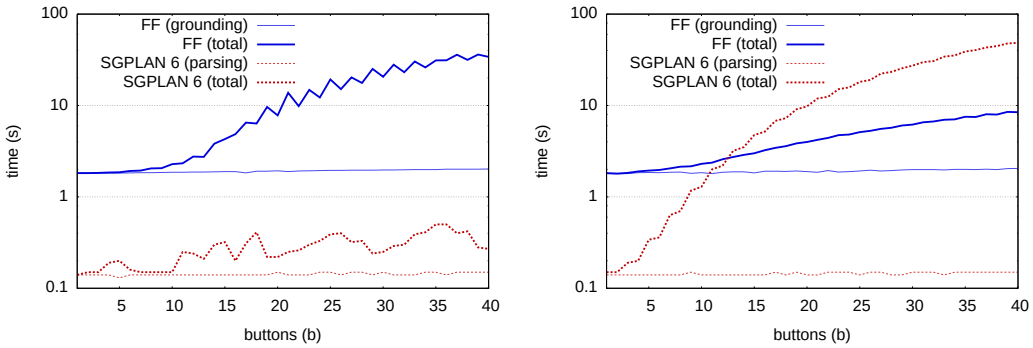
Results for the $h = 20$, $n = 10$ case with w ranging from 1 to 40 are shown in Figure 11. As in Experiment 3, a number of planners again fail to solve all the problems: Metric-FF, FF(h_a), and C^3 solve only a few instances, while FF only scales to $w = 23$. In the unordered version of the domain, SGPLAN easily solves inputs beyond $w = 40$ in well less than a second. LAMA is also reasonably successful on these problems; however, its runtimes grow more quickly than SGPLAN, with LAMA taking almost 5 seconds to solve the $w = 40$ problem instance. In the ordered case, we again see behaviour similar to that of Experiment 3: for the problem instances FF is able to solve, it performs significantly better than LAMA and SGPLAN. (SGPLAN's long term runtime appears to be growing at a slower rate than FF's, and so even if FF could be scaled to larger problem instances, it seems possible that SGPLAN might overtake FF as the better performer.) However, the overall planning times for most of these instances are concerning since times over a couple seconds will negatively impact the overall response time of an NLG system, which must react in real time to user actions.

Finally, we also performed a set of experiments designed to investigate the tradeoff between grounding time and search time on certain grid configurations. For these experiments, we initially fixed the size of the grid and then varied the number of buttons b in the world, thereby creating a series of "snapshots" of particular extra-cell GIVE domains. Figure 12 shows the results of these experiments for the FF and SGPLAN planners, for a fixed size grid of height 20 and width 40, and the number of buttons b ranging from 1 to 40. In each case, the amount of time a planner spends on grounding is relatively unchanged as we vary the number of buttons in a grid, while the search time



(a) Unordered (b) Ordered

FIGURE 11: Results for the unordered and ordered GIVE domains with $h = 20$ and $n = 10$. The horizontal axis is the extra grid width w. The vertical axis is the total runtime in seconds.



(a) Unordered (b) Ordered

FIGURE 12: Results for the GIVE domains with a fixed grid size of height 20 and width 40. The horizontal axis is the number of buttons b. The vertical axis is the runtime in seconds (log scale).

continues to rise (sometimes quite dramatically), as b increases. (We saw a similar effect for other grid configurations we tried.) This observation has important consequences for the design of our GRID worlds: changing the underlying domain structure, even minimally, may result in significant—and often unexpected—performance differences for the planners that must operate in these domains.

5. DISCUSSION

We can draw both positive and negative conclusions from our experiments about the state of planning for modern NLG applications. On the one hand, we found that modern planners are very good at dealing with the *search* problems that arise in the NLG-based planning problems we investigated. In the sentence generation domain, FF's Enforced Hill-Climbing strategy finds plans corresponding to 25-word sentences in about a second. It is hard to compare this number to a baseline because there are no shared benchmark problems, but FF's search performance is similar to that of a greedy, incomplete special-purpose algorithm, and competitive with other sentence generators as well. Thus, research on search strategies for planning has paid off; in particular, the Enforced Hill-Climbing heuristic outperforms the best-first strategy to which FF 2.3 switches for some problem instances. Similarly, SGPLAN's performance on the GIVE domain is very convincing and fast enough for many instances of this application.

On the other hand, each of the off-the-shelf planners we tested spent substantial amounts of time on preprocessing. This is most apparent in the sentence generation domain, where the planners spent almost their entire runtime on grounding the predicates and operators for some problem instances. This effect is much weaker in the GIVE domain, which has a much smaller number of operators and less interactions between the predicates in the domain. However, our GIVE experiments also illustrate that altering the structure of a domain, even minimally, can significantly change a planner's performance on a problem. For instance, in some of our GIVE experiments with extra grid positions, increasing the number of buttons in the world, while keeping the dimensions of the grid fixed, resulted in a significantly larger search time while the preprocessing time remained essentially unchanged.

While the GIVE domain can be defined in such a way that the number of operators is minimized, this is not possible for an encoding of a domain in which the operators model the different communicative actions that an NLG system can use. For instance, in the sentence generation domain, the XTAG planning problem for $k = 2$ and $n = 5$ consists of about 1000 operators for the different lexicon entries for all the words in the sentence, some of which take four parameters. It is not unrealistic to assume a knowledge base with a few hundred individuals. All this adds up to trillions of ground instances: a set which is completely infeasible to compute naïvely.

Of course, it would be premature to judge the usefulness of current planners as a whole, based on just two NLG domains. Nevertheless, we believe that the structure of our planning problems, which are dominated by large numbers of operators and individuals, is typical of NLG-related planning domains. This suggests that while current planners are able to manage many of the search problems in the domains we looked at, they are still largely unsuitable for practical NLG applications because of the time they spend on preprocessing.

We are also aware that the time a planner invests in preprocessing can pay off during search, and that such techniques have been invaluable in improving the overall running time of modern planners. However, we still suggest that the inability of a planner to scale to larger domains limits its usefulness for applications beyond NLG as well. Furthermore, we feel that the problem of preprocessing receives less research attention than it deserves: if the problem is scientifically trivial then we challenge the planning community to develop more efficient implementations that address the concerns raised in our experiments; otherwise, we look forward to future publications on this topic. To support this effort, we offer our planning domains as benchmarks for future research and competitions.[9]

Finally, we found it very convenient that the recent International Planning Competitions provide a useful entry point for selecting and obtaining current planners. Nevertheless, our experiments exposed several bugs in the planners we tested, which required us to change their source code to make them scale to our inputs. We also found that different planners that solve the same class of planning problems (e.g., STRIPS, ADL, etc.) sometimes differ in the variants of PDDL that they support. These differences range from fragments of ADL that can be parsed, to sensitivity in the order of declarations and the use of "objects" rather than "individuals" as the keyword for declaring the universe. We propose that the case for planning as a mature technology with professional-quality implementations could be made more strongly if such discrepancies were harmonized.

6. CONCLUSION

In this paper, we investigated the usefulness of current planning technology to natural language generation, an application area with a long tradition of using automated planning that has recently experienced a renewed interest in such techniques from NLG researchers. In particular, we evaluated the performance of several off-the-shelf planners on a series of planning domains that arose in the context of sentence generation and situated instruction generation.

Our results were mixed. While some of the planners we tested—in particular, FF and SGPLAN—did an impressive job of controlling the complexity of the search, we also found that all of the off-the-shelf planners we tested spent significant amounts of time on preprocessing, thereby limiting their usefulness for real-world NLG problems. For instance, in the sentence generation domain, FF spent

[9]The PDDL problem generators for our NLG domains are available at `http://www.coli.uni-saarland.de/~koller/projects/crisp`.

90% of its runtime on computing the ground instances of the planning operators; in the instruction-giving domain, which is very similar to Gridworld, a similar effect happened for certain combinations of grid sizes and buttons. As a result, this overly long preprocessing time makes these planners an inappropriate choice for NLG applications, in any but the smallest problem instances. Users who come to planning from outside the field, such as NLG researchers, treat planners as black boxes. This means that search efficiency alone is not helpful when other modules of the planner are slow. From this perspective, we believe that more effort should be spent on optimising the preprocessing components of new and existing planners, with similar vigour as research into the search problem itself. We propose that one line of research might be to investigate planning algorithms that do not rely on grounding out all operators prior to the search, but instead selectively perform this operation when needed. In particular, recent work aimed at analyzing the performance of FF in the sentence generation domain has resulted in a set of minor enhancements to FF's preprocessor and search options, leading to significant performance improvements (often several orders of magnitude better) on this problem (Koller and Hoffmann 2010), compared to the original version of FF tested here.

NLG and planning have a long history in common. The recent surge in NLG-as-planning research presents valuable opportunities for both disciplines. Clearly, NLG researchers who apply planning technology will benefit directly from any improvements in planner efficiency. Conversely, NLG may also be a worthwhile application area for planning researchers to keep in mind. Domains like GIVE highlight certain challenges, such as plan execution monitoring and plan presentation (i.e., summarisation and elaboration), but also offer a platform on which such technologies can be evaluated in experiments with human users. Furthermore, although we have focused on classical planning problems in this work, research related to reasoning under uncertainty, resource management, and planning with knowledge and sensing, can also be investigated in these settings. As such, we believe our domains would provide interesting challenges for planners entered in future editions of the IPC.

Acknowledgements

This work arose in the context of the Planning and Language Interest Group at the University of Edinburgh. The authors would like to thank all members of this group, especially Héctor Geffner and Mark Steedman, for interesting discussions. We also thank our reviewers for their insightful and challenging comments. This work was supported by the DFG Research Fellowship "CRISP: Efficient integrated realization and microplanning", the DFG Cluster of Excellence "Multimodal Computing and Interaction", and by the European Commission through the PACO-PLUS project (FP6-2004-IST-4-27657).

REFERENCES

Appelt, D., 1985. Planning English Sentences. Cambridge University Press, Cambridge, England, 171 pp.

Benotti, L., 2008. Accommodation through tacit sensing. In Proceedings of the 12th Workshop on the Semantics and Pragmatics of Dialogue. London, United Kingdom, pp. 75–82.

Blum, A. and M. Furst, 1997. Fast planning through graph analysis. Artificial Intelligence, **90**:281–300.

Brenner, M. and I. Kruijff-Korbayová, 2008. A continual multiagent planning approach to situated dialogue. In Proceedings of the 12th Workshop on the Semantics and Pragmatics of Dialogue. pp. 67–74.

Byron, D., A. Koller, K. Striegnitz, J. Cassell, R. Dale, J. Moore, and J. Oberlander, 2009. Report on the first NLG challenge on generating instructions in virtual environments (GIVE). In Proceedings of the 12th European Workshop on Natural Language Generation. Athens.

Hoffmann, J., 2002. Extending FF to numerical state variables. In Proceedings of the 15th European Conference on Artificial Intelligence (ECAI-02). pp. 571–575.

Hoffmann, J. and S. Edelkamp, 2005. The deterministic part of IPC-4: An overview. Journal of Artificial Intelligence Research, **24**:519–579.

Hoffmann, J. and B. Nebel, 2001. The FF planning system: Fast plan generation through heuristic search. Journal of Artificial Intelligence Research, **14**:253–302.

Hovy, E., 1988. Generating natural language under pragmatic constraints. Lawrence Erlbaum Associates, Hillsdale, NJ, USA, 224 pp.

Hsu, C. W., B. W. Wah, R. Huang, and Y. X. Chen, 2006. New features in SGPlan for handling soft constraints and goal preferences in PDDL 3.0. In Proceedings of the Fifth International Planning Competition, 16th International Conference on Automated Planning and Scheduling. The English Lake District, Cumbria, United Kingdom, pp. 39–41.

Joshi, A. and Y. Schabes, 1997. Tree-Adjoining Grammars. In Handbook of Formal Languages, edited by G. Rozenberg and A. Salomaa, Springer-Verlag, Berlin, Germany, volume 3. pp. 69–123.

Keyder, E. and H. Geffner, 2008. The FF(h_a) planner for planning with action costs. In Proceedings of the Sixth International Planning Competition.

Koller, A. and J. Hoffmann, 2010. Waking up a sleeping rabbit: On natural-language sentence generation with FF. In Proceedings of the 20th International Conference on Automated Planning and Scheduling. Toronto, ON, Canada, pp. 238–241.

Koller, A. and M. Stone, 2007. Sentence generation as planning. In Proceedings of the 45th Annual Meeting of the Association of Computational Linguistics. Prague, Czech Republic, pp. 336–343.

Koller, A. and K. Striegnitz, 2002. Generation as dependency parsing. In Proceedings of the 40th Annual Meeting of the Association for Computational Linguistics. Philadelphia, PA, USA, pp. 17–24.

Lipovetzky, N., M. Ramirez, and H. Geffner, 2008. C3: Planning with consistent causal chains. In Proceedings of the Sixth International Planning Competition.

McDermott, D. and the AIPS-98 Planning Competition Committee, 1998. PDDL – The Planning Domain Definition Language. Technical Report CVC TR-98-003/DCS TR-1165, Yale Center for Computational Vision and Control, 27 pp.

Perrault, C. R. and J. F. Allen, 1980. A plan-based analysis of indirect speech acts. American Journal of Computational Linguistics, **6**(3–4):167–182.

Reiter, E. and R. Dale, 2000. Building Natural Language Generation Systems. Cambridge University Press, Cambridge, England, 248 pp.

Richter, S. and M. Westphal, 2008. The LAMA planner: Using landmark counting in heuristic search. In Proceedings of the Sixth International Planning Competition.

Steedman, M. and R. P. A. Petrick, 2007. Planning dialog actions. In Proceedings of the Eighth SIGdial Workshop on Discourse and Dialogue. Antwerp, Belgium, pp. 265–272.

Stoia, L., D. M. Shockley, D. K. Byron, and E. Fosler-Lussier, 2008. SCARE: A situated corpus with annotated referring expressions. In Proceedings of the 6th International Conference on Language Resources and Evaluation (LREC 2008).

Stone, M., C. Doran, B. Webber, T. Bleam, and M. Palmer, 2003. Microplanning with communicative intentions: The SPUD system. Computational Intelligence, **19**(4):311–381.

Stone, M. and B. Webber, 1998. Textual economy through close coupling of syntax and semantics. In Proceedings of the Ninth International Workshop on Natural Language Generation. pp. 178–187.

Tovey, C. and S. Koenig, 2000. Gridworlds as testbeds for planning with incomplete information. In Proceedings of the 17th National Conference on Artificial Intelligence. Austin, TX, USA, pp. 819–824.

XTAG Research Group, 2001. A lexicalized tree adjoining grammar for english. Technical Report IRCS-01-03, IRCS, University of Pennsylvania. `ftp://ftp.cis.upenn.edu/pub/xtag/release-2.24.2001/tech-report.pdf`.

Young, R. M. and J. D. Moore, 1994. DPOCL: a principled approach to discourse planning. In Proceedings of the Seventh International Workshop on Natural Language Generation. Kennebunkport, Maine, USA, pp. 13–20.

LIST OF FIGURES

Appendix C

Learning action effects in partially observable domains

Kira Mourão and **Ronald P. A. Petrick** and **Mark Steedman** [1]

Abstract. We investigate the problem of learning action effects in partially observable STRIPS planning domains. Our approach is based on a voted kernel perceptron learning model, where action and state information is encoded in a compact vector representation as input to the learning mechanism, and resulting state changes are produced as output. Our approach relies on deictic features that assume an attentional mechanism that reduces the size of the representation. We evaluate our approach on a number of partially observable planning domains, and show that it can quickly learn the dynamics of such domains, with low average error rates. We show that our approach handles noisy domains, conditional effects, and that it scales independently of the number of objects in a domain.

1 INTRODUCTION AND MOTIVATION

Acquiring a domain model automatically through learning and experience gives an agent greater flexibility to handle unexpected situations, and avoids the need for a predefined world model. Existing approaches either work within the space of transition rules to find a "good" set, or all consistent sets, of rules [2, 3, 12, 15], or they operate at the sensor level by constructing transition rules from actions and robot sensor data coded as sets of objects or raw sensor readings, and predicates derived from this data [7, 10]. The former, high-level, methods have been applied to partially observable [2, 15] or nondeterministic [12] domains, but are not applicable to domains which are both noisy and partially observable; few are also able to learn conditional effects. The latter, low-level, methods can learn in noisy, partially observable domains, but the domains are much simpler, without relations between objects, and sometimes without objects at all. Here, we extend our previous work on learning action models in noiseless, fully observable domains [11]. Our method learns the effects of STRIPS actions [4], extended to admit conditional effects, in deterministic, noisy and partially observable versions of the more complex domains typical of the high-level approaches.

2 REPRESENTATION

We learn action models from sequences of interleaved actions and state observations. Each observation initially encompasses as much of the world state as the agent is able to detect, with some parts of the state potentially unobserved or corrupted by noise. We reduce the size of each observation by only considering objects which can be identified by a deictic reference [1], and then transform each observation into a vector to use as input to the learning model.

A deictic representation maintains pointers to objects of interest in the world, with objects coded relative to the agent or current action.

[1] University of Edinburgh, UK, email: kira.mourao@ed.ac.uk, {rpetrick,steedman}@inf.ed.ac.uk

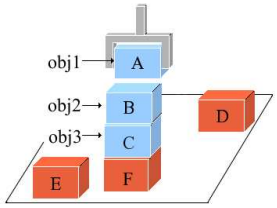


Figure 1. Computing deictic references: an example from the BlocksWorld domain, in which an agent can manipulate a set of blocks on a table. Given the action *stack*(A, B), i.e., stack block A on top of block B, the initial set of objects of interest is $\{A, B\}$. The only object related to A or B is C, since B is on C. Therefore the full set of objects of interest is $\{A, B, C\}$.

We take a similar approach to previous work applying deictic representations to learn domain dynamics [3, 12]. For a given action instance we construct the set of objects of interest, consisting of the set of objects which are parameters of the action, and the objects which, in the current state, are related to any object in the action parameters (see Figure 1). This single step computation is in contrast to previous approaches, where the set of objects under consideration is the full transitive closure under all relations among objects. Also, whereas previous approaches ignored objects if they were not *uniquely* defined by deictic reference, we allow deictic references to any set of objects that are indistinguishable relative to the action parameters.

An input vector representing the reduced state space is then constructed by assigning a bit for each action, 0-ary fluent, and for each possible relation involving only the objects in the reduced state space. The value of a bit is 1 (-1) if the corresponding fluent is true (false), or if the corresponding action is (not) the current action. Bits for unobserved or unused fluents are set to an arbitrary value *N*, which is ignored during learning.

Vectors representing an action's effects on a state are identical in form to the input vectors, except that actions are excluded from the vector, and bits are set to 1 (-1) if the corresponding fluent changes (does not change). Bits corresponding to unobserved or unused fluents are set to *N*.

3 LEARNING MODEL

The task of the learning mechanism is to learn the deterministic associations between action-state pairs and their effects. It is assumed that the number and type of parameters of each action, predicate and function are known. Action preconditions and effects are not known, and effects may be conditional. Disjunctive effects are not allowed. Instead, all effects are conjunctions of predicates, meaning it is sufficient to learn the rule for each predicate separately. Using the vector representation defined above, state transitions can be learnt using a bank of classifiers, one for each bit of the output vector.

To address our learning problem we construct a variant of the perceptron algorithm [13], using the voted perceptron [5], which is

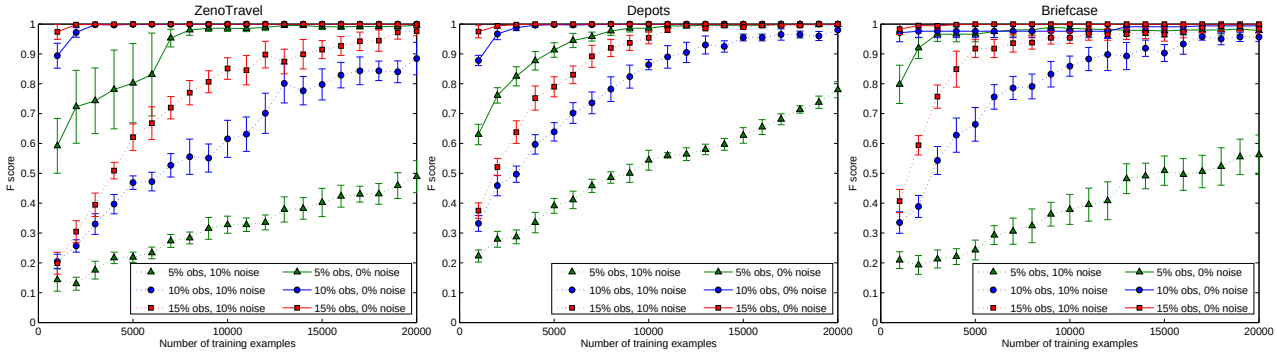


Figure 2. Results of learning action models in standard planning domains. Error bars are 95% confidence intervals. In noiseless, fully observable domains, models fully predict all test cases after less than 200 examples (results not shown). While observing only a small fraction of the state, without noise, the learning model completely predicts the test set after 20000 examples, in many of the test cases: with 15% of the state observable, the F-score is not significantly different from 1 (t-tests, $p > 0.05$) in any of our domains. With noise, the learnt models are clearly poorer, but some aspects of each domain are still learnt: with 15% of the state observable, the F-score is significantly different from 1 for ZenoTravel ($p = 0.004$) and Briefcase ($p = 0.046$), but not for Depots ($p > 0.05$).

noise-tolerant [8] and computationally efficient, producing performance close to the best performing maximal-margin classifiers (e.g. SVMs) on similar problems. We use the DNF kernel [14], which allows the perceptron to run over the feature space of all possible conjunctions of bits in the input space, i.e., the space of possible rules.

4 EXPERIMENTS

We tested the learning model on standard planning domains from the 3rd International Planning Competition (IPC): Depots, ZenoTravel and DriverLog; a standard BlocksWorld domain; and Briefcase, a domain with conditional effects. Sequences of random actions and resulting states were generated from PDDL domain descriptions [9] and used as training and testing data.[2] Specific problems from the IPC were used to set the sizes of the initial states for each sequence. The actual initial states were generated at random using the IPC3 problem generator and a Briefcase state generator [6].

To determine error bounds on our results, we used 10 different randomly generated training and testing sets. Each training set consisted of 1000-20000 actions and matching state observations. Partial observability was simulated by randomly selecting a fraction (5-20%) of bits to retain in each state vector, and setting the remaining bits to *N*. Sensor noise at 10% was simulated by flipping each bit in the state vector with probability 0.1. Each test set was a fully observable, noiseless sequence of 2000 actions and observations. We measured the performance on our test sets by considering the fluents which our model predicted would change versus the fluents which did change, and calculating the balanced F-measure, the harmonic mean of precision and recall (*true positives/predicted changes* and *true positives/actual changes*, respectively). Selected results of the experiments are shown in Figure 2.

5 CONCLUSIONS AND FUTURE WORK

We have presented a method for learning deterministic action models which is fast, scalable and handles noise and partial observability of the world state. Furthermore, the error rate of the predictions made by the model is low. The speed, scalability and accuracy make the approach highly suitable for use in planning applications.

Additionally, our approach can learn conditional effects. Note that the success or failure of an action, which depends on its preconditions, is a form of conditional effect (the action has a null effect unless the preconditions are satisfied). Therefore action preconditions can also be learnt, if examples of action failures as well as action successes are provided.

A key step in future work will be to extract STRIPS-style rules from the sets of ordered pairings of entire states presently learnt by the model, so that the learning model can be integrated with standard planning software. We also plan to apply our method to intrinsically noisy and partially observable real-world robot environments.

[2] All data was generated using the Random Action Generator 0.5 available at `http://magma.cs.uiuc.edu/filter/`.

ACKNOWLEDGEMENTS

This work was funded by the EU PACO-PLUS project (FP6-2004-IST-4-27657) and Edinburgh University Neuroinformatics DTC.

REFERENCES

[1] P. E. Agre and D. Chapman, 'Pengi: an implementation of a theory of activity', in *Proc. of AAAI*, pp. 268–272, (1987).
[2] E. Amir and A. Chang, 'Learning partially observable deterministic action models', *JAIR*, **33**, 349–402, (2008).
[3] S. S. Benson, *Learning Action Models for Reactive Autonomous Agents*, Ph.D. dissertation, Stanford University, 1996.
[4] R. E. Fikes and N. J. Nilsson, 'STRIPS: A new approach to the application of theorem proving to problem solving', *AIJ*, **2**, 189–208, (1971).
[5] Y. Freund and R. Schapire, 'Large margin classification using the perceptron algorithm', *Machine Learning*, **37**, 277–296, (1999).
[6] J. Hoffmann and B. Nebel. FF domain collection. `http://www.loria.fr/~hoffmanj/ff-domains.html`.
[7] M. Holmes and C. Isbell, 'Schema learning: Experience-based construction of predictive action models', in *NIPS 17*, pp. 585–562, (2005).
[8] R. Khardon and G. M. Wachman, 'Noise tolerant variants of the perceptron algorithm', *JMLR*, **8**, 227–248, (2007).
[9] D. McDermott, M. Ghallab, A. Howe, C. Knoblock, A. Ram, M. Veloso, D. Weld, and D. Wilkins, 'PDDL - the planning domain definition language', Technical report, CVC TR-98-003, Yale, (1998).
[10] J. Modayil and B. Kuipers, 'The initial development of object knowledge by a learning robot', *Robot. Auton. Syst.*, **56**(11), 879–890, (2008).
[11] K. Mourão, R. Petrick, and M. Steedman, 'Using kernel perceptrons to learn action effects for planning', in *Proc. of CogSys*, pp. 45–50, (2008).
[12] H. M. Pasula, L. S. Zettlemoyer, and L. P. Kaelbling, 'Learning symbolic models of stochastic domains', *JAIR*, **29**, 309–352, (2007).
[13] F. Rosenblatt, 'The perceptron: a probabilistic model for information storage and organization in the brain', *Psych. Rev.*, **65**(6), 386–408, (1958).
[14] K. Sadohara, 'Learning of boolean functions using support vector machines', in *Proc. of ALT*, pp. 106–118, (2001).
[15] Q. Yang, K. Wu, and Y. Jiang, 'Learning action models from plan examples using weighted MAX-SAT', *AIJ*, **171**(2-3), 107–143, (2007).

Appendix D

Learning action effects in partially observable domains

Kira Mourão
School of Informatics
University of Edinburgh
Edinburgh EH8 9AB, Scotland, UK
k.m.t.mourao@sms.ed.ac.uk

Ronald P. A. Petrick
School of Informatics
University of Edinburgh
Edinburgh EH8 9AB, Scotland, UK
rpetrick@inf.ed.ac.uk

Mark Steedman
School of Informatics
University of Edinburgh
Edinburgh EH8 9AB, Scotland, UK
steedman@inf.ed.ac.uk

Abstract

We investigate the problem of learning action effects in partially observable STRIPS planning domains. Our approach is based on a voted kernel perceptron learning model, where action and state information is encoded in a compact vector representation as input to the learning mechanism, and resulting state changes are produced as output. Our approach relies on deictic features that embody a notion of attention and reduce the size of the representation. We evaluate our approach on a number of partially observable planning domains, adapted from domains used in the International Planning Competition, and show that it can quickly learn the dynamics of such domains, with low average error rates. Furthermore, we show that our approach handles noisy domains, and scales independently of the number of objects in a domain, making it suitable for large planning scenarios.

Introduction and motivation

An agent operating in a real-world domain often needs to do so with incomplete information about its environment. In particular, an agent must often act or make decisions with only partial or noisy information about the state of the world. Automated planning systems are effective at controlling the behaviour of agents in a variety of domains. However, such tools require a model of the domain in which the agent will operate. In real-world domains, such models may not be readily available, nor be sufficiently detailed to account for the subtleties inherent in complex environments.

Acquiring a domain model automatically through learning and experience gives an agent greater flexibility to handle unexpected situations, and avoids the need for a pre-existing model of the world. Learning the dynamics of a domain can be a challenging problem, however, especially in domains where an agent only has partial access to the world state, or external sensors that are susceptible to noise. Furthermore, since a learnt action model may be subsequently used for planning, the resulting learning method should be as accurate, fast, and scalable as possible.

Using machine learning techniques to induce action models is not a new idea, with the literature divided between two main approaches: high-level, logical approaches and low-level, sensor-driven approaches. High-level approaches work within the space of transition rules (Wang 1995; Gil 1994; Amir and Chang 2008; Pasula, Zettlemoyer, and Kaelbling 2007; Benson 1996) to find either a "good" set or all consistent sets of rules. These methods attempt to exploit relational structure in order to improve speed and generalisation performance. Such approaches have also been applied to partially observable (Amir and Chang 2008) or non-deterministic (Pasula, Zettlemoyer, and Kaelbling 2007) domains. Alternatively, low-level methods operate closer to the sensor level. Such approaches construct transition rules from actions and robot sensor data coded as sets of objects or raw sensor readings, and predicates derived from this data (Metta and Fitzpatrick 2003; Holmes and Isbell 2005; Doğar et al. 2007; Modayil and Kuipers 2008). Although many of these methods have had limited success at learning aspects of particular domains, few of them fully address the problem of learning partially observable domains, and fewer still are capable of handling noise.

In this paper we consider the problem of learning the effects of an agent's actions, that is, the transition rules between world states. We focus on learning the effects of STRIPS actions (Fikes and Nilsson 1971) in deterministic and partially observable domains. In particular, we consider actions which affect a subset of the propositional features that make up the world state. Following (Pasula, Zettlemoyer, and Kaelbling 2007; Benson 1996), we use deictic features that embody a notion of attention to produce a compact representation of the domain.

This paper builds on our previous work (Mourão, Petrick, and Steedman 2008) which also used deictic coding to generate a compact vector representation of the world state, and learnt action effects as a classification problem. However, the method only applied to fully observable domains, as the kernel perceptron classifier used there performs badly with noisy or partially observable data. Additionally, the approach was only tested on a single synthetic domain with simulated states which were not necessarily reachable by a sequence of actions in the domain.

Here we extend this work to partially observable (and noisy) domains using kernelised voted perceptrons (Aizerman, Braverman, and Rozoner 1964; Freund and Schapire 1999) to learn action transitions in the domains. Such methods are particularly useful since they provide good performance, in terms of both the training time and the quality of the learnt models. Furthermore, we test our approach against a set of standard planning domains taken from the

3rd International Planning Competition,[1] demonstrating that our method is fast, accurate, and scales independently of the number of objects in the world, thereby making it suitable for large planning scenarios.

Domain learning

Definitions

The action representations we will use are based on the logical representations typically found in planning systems. A *domain* $\mathcal{D}$ is defined as a tuple $\mathcal{D} = \langle \mathcal{O}, \mathcal{P}, \mathcal{A} \rangle$, where $\mathcal{O}$ is a finite set of world objects, $\mathcal{P}$ is a finite set of predicate (relation) symbols, and $\mathcal{A}$ is a finite set of actions. Each predicate and action also has an associated arity. Predicates of arity 0 are referred to as *object independent* properties, while those of arity at least 1 are *object dependent* properties.

A *fluent* is an expression $p(c_1, c_2, \ldots, c_n)$, where $p \in \mathcal{P}$, n is the arity of p, and each $c_i \in \mathcal{O}$. A *state* is any set of fluents, and $\mathcal{S}$ is the set of all possible states. For any state $s \in \mathcal{S}$, a fluent p is true at s iff $p \in s$. The negation of a fluent, $\neg p$, is true at s (also, p is false at s) iff $p \notin s$.

Each action $a \in \mathcal{A}$ is defined by a set of *preconditions*, Pre_a, and a set of *effects*, Eff_a. Pre_a can be any set of fluents and negated fluents. In STRIPS actions each effect $e \in Eff_a$ has the form $add(p)$ or $del(p)$, where p is any fluent. Action preconditions and effects can also be parameterised. An action with all of its parameters replaced with objects from $\mathcal{O}$ is said to be an *action instance*.

Action instances are state transforming. Given a state s and an action instance A, A is *applicable* (or *executable*) at s iff each precondition $p \in Pre_A$ is true at s. An applicable action produces a new state s' that is identical to s, but updated with the effects of A as follows: for each $e \in Eff_A$, (i) if e is an effect $add(p)$ then p is added to s', and (ii) if e is an effect $del(p)$ then p is removed from s'.

Learning model

The task of the learning mechanism is to learn the associations between action-precondition pairs and their effects, that is, rules of the form $\langle A, Pre_A \rangle \rightarrow Eff_A$. As a result of the form of the planning actions we allow, effects of rules are assumed to be deterministic and disjunctive effects (i.e., effects of the form "either p_1 or p_2 changes") are not allowed. Instead, all effects are simply conjunctions of predicates, meaning it is sufficient to learn the rule for each effect predicate separately. Thus, we can treat the learning problem as a set of binary classification problems, with one problem for each effect predicate.

To address our particular learning problem we use the *perceptron* (Rosenblatt 1958), a simple yet fast binary classifier. The perceptron maintains a weight vector $\mathbf{w}$ which is adjusted at each training step. The i-th input vector $\mathbf{x}_i \in \{0,1\}^n$ in a class $y \in \{-1, 1\}$ is classified by the perceptron using the decision function $f(\mathbf{x}_i) = sign(\langle \mathbf{w} \cdot \mathbf{x}_i \rangle)$. If $f(\mathbf{x}_i)$ is not the correct class then $\mathbf{w}$ is set to $\mathbf{w} + y\mathbf{x}_i$; if $f(\mathbf{x}_i)$ is correct then $\mathbf{w}$ is left unchanged. Provided the data is linearly separable, the perceptron algorithm is guaranteed to converge on a solution in a finite number of steps (Novikoff 1963; Minsky and Papert 1969). If the data is not linearly separable then the algorithm oscillates, changing $\mathbf{w}$ at each misclassified input vector.

One solution for non-linearly separable data is to map the input feature space into a higher-dimensional space where the data is linearly separable. However, an explicit mapping may lead to a massive expansion in the number of features, making the classification problem computationally infeasible. Instead, an implicit mapping is achieved by applying the *kernel trick* to the perceptron algorithm (Freund and Schapire 1999), by noting that the decision function can be written in terms of the dot product of the input vectors:

$$f(\mathbf{x}_i) = sign(\langle \mathbf{w} \cdot \mathbf{x}_i \rangle) = sign(\sum_{j=1}^{n} \alpha_j y_j \langle \mathbf{x}_j \cdot \mathbf{x}_i \rangle),$$

where α_j is the number of times the j-th example has been misclassified by the perceptron. By replacing the dot product with a *kernel function* $k(\mathbf{x}_i, \mathbf{x}_j)$ which calculates $\langle \phi(\mathbf{x}_i) \cdot \phi(\mathbf{x}_j) \rangle$ for some mapping ϕ, the perceptron algorithm can be applied in higher dimensional spaces without ever requiring the mapping to be explicitly calculated.

We represent each state s as a vector (see below) and learn state transitions using a bank of kernel perceptrons, one for each output bit, corresponding to a single predicate p. Since in general the problem of learning action effects is not linearly separable, the kernel perceptron is an appropriate choice for this problem. Kernel perceptrons obtain reasonable accuracy at acceptable training and prediction speeds, allowing us to use this approach in practical planning applications. Alternative non-linear classifiers, such as SVMs (Boser, Guyon, and Vapnik 1992), can be substantially slower (Surdeanu and Ciaramita 2007) while performance is not guaranteed to be better (Graepel, Herbrich, and Williamson 2000). To improve the speed of the classifier we use a variant of the kernel perceptron, the *voted perceptron* (Freund and Schapire 1999), which is computationally efficient and produces performance close to the best performing maximal-margin classifiers on similar problems. Preliminary tests using SVMs on our problem give similar results with longer computation times.

The kernel is chosen to allow the perceptron algorithm to run over conjunctions of features in the original input space, as this permits a more accurate representation of the exact conjunction of features (action and preconditions) corresponding to a particular effect. We use the polynomial kernel of degree 3, $k(x, y) = (x.y + 1)^3$ so that feature conjunctions of up to three features make up the feature space.

Representation

We compute a reduced form of the input state space for each action using deictic coding (Agre and Chapman 1987). A deictic representation maintains pointers to objects of interest in the world, with objects coded relative to the agent or current action. Objects which cannot be indexed in this way are excluded from the reduced state for the current action.

[1] See `http://ipc.icaps-conference.org/`.

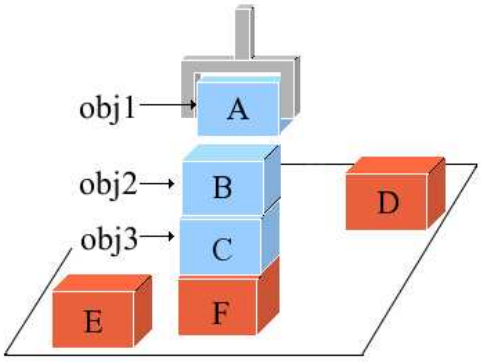


Figure 1: Action *stack*(A, B) results in objects A, B, and C being attended to, while unrelated objects D, E and F are ignored. Objects A, B, and C are referred to by the variables $obj1$, $obj2$ and $obj3$, respectively, in the vector representation shown in Figure 2.

Input vector	Corresponding action/predicate	
−1	*pickup*(*obj1*)	Actions
−1	*putdown*(*obj1*)	
1	*stack*(*obj1*, *obj2*)	
−1	*unstack*(*obj1*, *obj2*)	
−1	*armempty*	Object independent properties
	...	
1	*holding*	Properties of *obj1*
−1	*ontable*	
−1	*clear*	
−1	*on-obj1*	
−1	*on-obj2*	
−1	*on-obj3*	
	...	
−1	*holding*	Properties of *obj2*
−1	*ontable*	
1	*clear*	
−1	*on-obj1*	
−1	*on-obj2*	
1	*on-obj3*	
	...	
−1	*holding*	Properties of *obj3*, included as *obj3* is related to *obj2*
1	*ontable*	
−1	*clear*	
−1	*on-obj1*	
−1	*on-obj2*	
−1	*on-obj3*	
	...	

Figure 2: Input vector representation of the (fully observable) BlocksWorld *stack* action and prior state from Figure 1. The first 4 bits correspond to the 4 domain actions. The bit for *stack* is set to 1 since it is the current action. The 0-ary predicate *armempty* is represented by a single bit, set to -1 since the gripper is holding object A. The first set of object predicates represented in the vector are those for object A since it is the first parameter of *stack*. The second set of object predicates relate to object B, as the second parameter of *stack*, and finally the third and last set of object predicates relate to object C, as it is related to object B by the *on* predicate. If object B were being stacked on object A the predicates for object B would precede those for object A instead.

For instance, such a deictic representation might arise from an attentional mechanism (Ballard et al. 1997).

For a given action instance A we construct the set of objects of interest $\mathcal{O}^{\mathcal{A}}$, by combining a primary set of objects, given by the parameters of the action, and a secondary set of objects which are directly related to the primary set, in the current state. We define a direct relation between objects c_i and c_j to exist in state $s \in \mathcal{S}$ if $\exists p \in \mathcal{P}$ such that $p(c_1, c_2, \ldots, c_n) \in s$ and $c_i, c_j \in \{c_1, c_2, \ldots, c_n\}$

The related objects are identified via deictic references, e.g., if a particular action grasps an object x, then the primary set of objects consists of the "grasped-object" x, and the secondary set might consist of the "object-under-the-grasped-object" y; other objects which are not directly related to x are not represented. We define $M = \max_A |\mathcal{O}^{\mathcal{A}}|$ to be the maximum possible number of objects of interest for any action instance.

Figure 1 presents an example from the BlocksWorld domain, in which an agent can manipulate a set of blocks on a table. Given the action *stack*(A, B), i.e., stack block A on top of block B, the primary set of objects is $\{A, B\}$. The only object related to A or B is C, since B is on C. Therefore the full set of objects of interest is $\{A, B, C\}$.

An input vector representing the state space is constructed as follows. Each action $a \in \mathcal{A}$, and each 0-ary predicate, is represented by a bit. Then for each object $o \in \mathcal{O}^{\mathcal{A}}$, all the possible relations between o and all other objects in $\mathcal{O}^{\mathcal{A}}$ must be represented. This requires at most $\binom{M-1}{n}$ bits for each n-ary predicate, for each object in $\mathcal{O}^{\mathcal{A}}$. The value of a bit is 1 (-1) if the corresponding predicate is true (false), or if the corresponding action is (not) the current action. If a bit corresponds to an unobserved predicate, the value is set to 0. When $|\mathcal{O}^{\mathcal{A}}| < M$ for some action instance A, bits for unused predicates are set to 0. Figure 2 shows the vector representation of the state in Figure 1.

The form of the output vectors representing an action's effects on a state are identical to the input vectors, except that the actions themselves are excluded from the vector, and bits are set to 1 if the corresponding predicate changes, -1 if the corresponding predicate does not change, and 0 if the corresponding predicate was not observed either before or after the current action. During learning, only examples with a known change, i.e. values 1 or -1, are used to train the kernel perceptrons. The ordering of object representations in the vectors is constrained so that two objects with the same role in the same action, but in two different instances of the action, must always be represented at the same position in the vectors.

Deictic coding has a number of benefits. It greatly reduces the size of the input for an algorithm learning to predict action effects, as information is discarded about objects unrelated to the action or its parameters. As a consequence, scalability is improved, because the size of the representation does not increase with the size of the universe.

The size of the representation required for relations between objects is also reduced. Firstly, any relations including discarded objects can be ignored. More importantly, deictic coding means that objects are represented by variables rather than by constants, and so whether we grasp object A sitting on object B, or grasp object C sitting on object D, the "on" relation is always represented as "the-object-I-will-grasp is on the-object-under-the-object-I-will-grasp". Thus, if the representation considers M objects of the possible $|\mathcal{O}|$ in the state space, the number of instances of each binary relation which needs to be represented drops from O($|\mathcal{O}|^2$) to O(M^2). $M < |\mathcal{O}|$ but is otherwise unrelated to $|\mathcal{O}|$, and instead typically depends on the complexity of the domain ($M < 8$ for the domains considered here). Finally, deictic coding creates a strong bias for generalisation.

Experiments

We tested the learning model on standard planning domains from the 3rd International Planning Competition (IPC): Depots and ZenoTravel, as well as a standard BlocksWorld domain. The domains are described in PDDL (McDermott et al. 1998), the standard representation language of the IPC. BlocksWorld is a very simple domain with 4 actions (maximum 2 parameters) and 5 predicates (maximum arity 2). All objects are blocks. The maximum possible number of objects of interest, M, is 3. Depots and ZenoTravel are more complex. Depots has 5 actions (maximum 4 parameters), 6 predicates (maximum arity 2) and 6 types of objects (represented as predicates of arity 1). ZenoTravel has 5 actions (maximum 6 parameters), 8 predicates (maximum arity 2) and 4 types of objects (again represented as predicates of arity 1). $M = 5$ and $M = 7$ for Depots and ZenoTravel respectively.

Sequences of random actions and resulting states were generated from PDDL domain descriptions and used as training and testing data.[2] The number of objects in the state space was higher in the test data than in the training data, to demonstrate that the learnt models could be applied across different instances of the same domain. Specific problems from the IPC were used to set the sizes of the initial states for each sequence.[3] BlocksWorld was initialised with 13 blocks for training and 30 blocks for testing. The actual initial states were generated at random using the IPC3 problem generator and a BlocksWorld state generator (Slaney and Thiébaux 2001).

Partial observability was simulated by randomly selecting a number of predicates from the world to observe after each action. The remaining predicates were discarded and the reduced state vector was generated from the observed fluents. The number of observed predicates was set to approximately 5-20% of the total number of predicates (including negations) required to fully describe the state (BlocksWorld: 209, ZenoTravel: 2116, and Depots: 1764).

[2] All data was generated using the Random Action Generator 0.5 available at `http://magma.cs.uiuc.edu/filter/`.

[3] Depots problem 5, and ZenoTravel problem 9 for training; Depots problem 19 and ZenoTravel problem 14 for testing.

To determine an error bound on our results, 10 runs with different randomly generated training and testing sets were used. Our testing environment was a 2.4 GHz quad-core system with 4 Gb of RAM. All experiments were run on a single CPU. Each training set consisted of sequences of actions and state observations of lengths ranging from 1000-20000, and each test set was a sequence of length 2000.

Results

Using our representation, learning in fully observable domains is easy. Accordingly, the action models in all three domains were learnt in under 250 examples, which was sufficient to fully predict the 2000 test examples (Figure 3(a)).

Partial observability reduces the number of useful examples which can be learnt from, and also reduces the number of useful bits in each example. Substantially more examples are therefore needed to learn the action model. Furthermore, the variance of the errors on the test set is much higher than in the fully observable case. The higher variance is due to the small number of observed predicates during training. Although the test case is fully observable, only a fraction of the state is used for prediction. This can cause the learner to mistake one action for another (a form of perceptual aliasing) and wrongly predict every instance of the action, resulting in a high number of errors on the test set.

In the BlocksWorld domain, observing 30 randomly chosen predicates from the full state description (approximately 15% of the state) over 9000 examples is sufficient to fully predict all the test sets of 2000 examples (Figure 3(b)). Observing fewer predicates also produces good results: training on 9000 examples is sufficient to fully predict eight of the ten test sets for both 10 and 20 observations at each time step. The failures are all instances of the *stack* action. Most incorrectly predict all instances of *stack*, resulting in approximately 25% error on each test set. One case wrongly predicts the results of the *stack* action only when the block being stacked upon is already stacked on top of another block, producing an 8% error.

In the ZenoTravel domain, 300 observations at each step (again, approximately 15% of the state) allows for complete prediction of the test set in three of the ten test cases. Of the remaining test sets, five have only 1 or 2 prediction errors, while the last three wrongly predict every case of the *refuel* action (by not predicting the deletion of the fluent specifying the initial fuel level), resulting in approximately 27% errors on each test set. By increasing the number of observations at each step to 400, eight of the ten test sets can be fully predicted, with only one error and two errors respectively on the remaining test sets (Figure 3(c)). The Depots domain is more challenging for our method and 300 observations at each step (approximately 17% of the state) are only sufficient to completely predict the test set in half the test cases. The other test sets have around 15% error as every case of the *load* action is wrongly predicted. However, with 400 observations at each step, after 14000 steps, all of the test sets are fully predicted (Figure 3(d)).

The poorly predicted test cases in the three domains are all instances of the perceptual aliasing problem discussed above. The problem can be resolved by supplying an exam-

ple to the learner which it would predict incorrectly, so that a new prediction function can be learnt. With a randomly generated sequence of actions, many additional examples could be required before such an example was generated. However, in a planning scenario the necessary example could be identified from a plan execution failure and used to improve the prediction.

Scalability

Our empirical results demonstrate the effectiveness of our approach in a number of standard planning domains. Our approach can also be shown to scale, making it suitable for more complex problems in domains with large numbers of actions and objects. In particular, our approach takes time proportional to a measure of the complexity of the domain, and the number of mistakes made during learning.

For a reduced state vector of length l, there are $l - |\mathcal{A}|$ voted perceptrons each computing one bit of the output vector. In the fully observable case, each voted perceptron learns in time proportional to the number of training examples n, the length of the reduced state vector l, and k, the number of mistakes made so far ($k \leq n$). Thus, the complexity of the learning algorithm is O(l^2n^2). Predictions are made in time O(l^2n). The same analysis applies to the partially observable case, however many more observations are needed, firstly because many observations will not contain the necessary information about whether a predicate has changed as a result of an action, and secondly because there is less information in the partially observed world state from which to learn. The former is mitigated by the fact that the kernel perceptrons only train on "useful" training examples where the change in a predicate is known, and only these examples incur significant computational cost.

Recall that M is the maximum possible number of objects of interest for any action instance. Additionally, we define $P_i = \{p \in \mathcal{P} : arity(p) = i\}$ and $m = max_{p \in \mathcal{P}}(arity(p))$. Then the length of the reduced state vector is given by $l = |\mathcal{A}| + |P_0| + M \sum_{i=1}^{m} \binom{M-1}{i} |P_i|$. Thus, l depends on the number and arity of predicates in the domain, and the maximum possible number of objects of interest. Intuitively it makes sense that more complex domains with more predicates and more inter-relations between objects should require more time to learn and predict. In particular, M does not depend on the number of object instances in the domain. For the planning domains considered here, which only have predicates with arity below 3, M is typically not very large ($M < 8$).

When the domain is fully observable, the number of observations n required by our method to learn the action model does not depend on the number of objects in the world. However, this is not the case for partially observable domains. An observation is only useful to the learning process if it contains fluents relating to the set of objects of interest, and at least one which was observed immediately before the current action, so that it is known whether a change occurred. Under partial observability, the probability of observing 'useful' fluents decreases as the number of objects in the world increases, as the number of 'useful' fluents remains constant while the total number of fluents increases. Thus the number of observations needed to learn the domain increases with the number of objects in the world. However we can learn in a smaller state space and apply the results to a larger one, since the representation is not dependent on the number of objects in the world.

The complexity can be reduced by limiting the number of vectors the voted perceptron stores, which has the effect of fixing k to be constant, and reduces the learning complexity by a factor of n to O(l^2n). However, the reduction of the state to a vector of length l by the deictic representation remains crucial. Accuracy may also be affected if k is too small. Furthermore, our solution is *embarrassingly parallel*, since the learning and prediction of each output bit is independent of the others. Running the calculations for each output bit in parallel would further reduce the complexity of learning and prediction, by a factor of l.

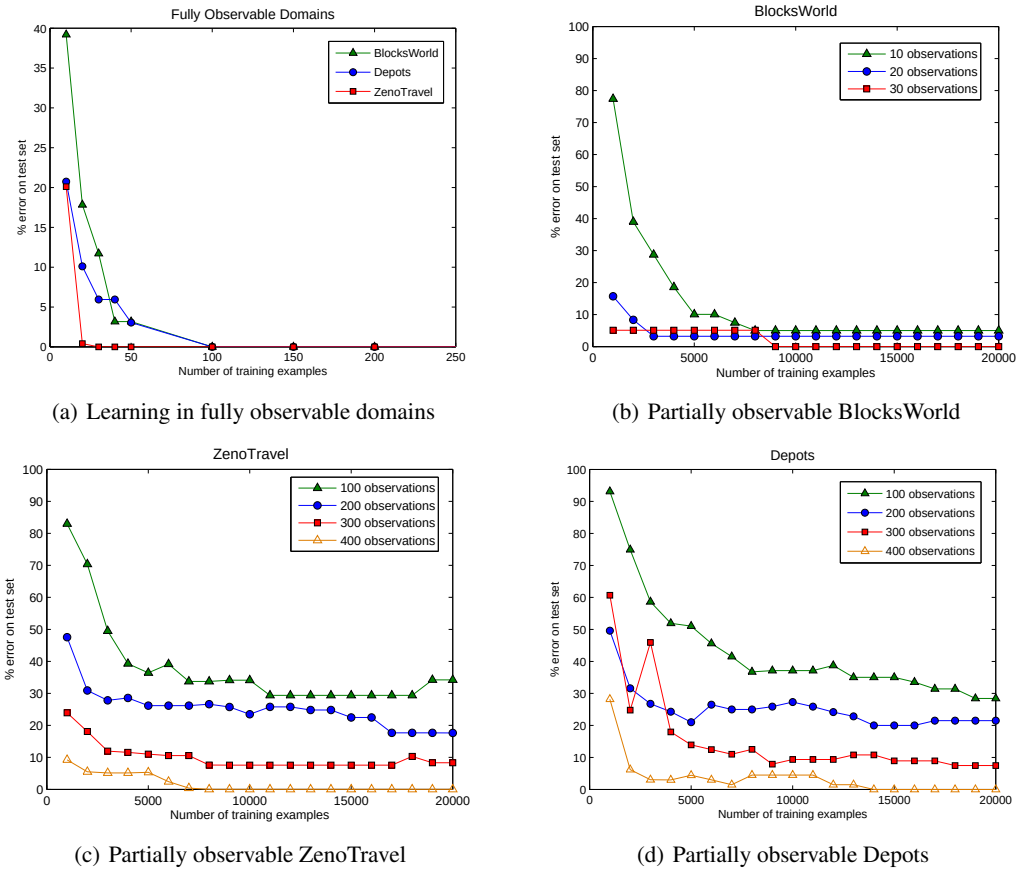
Currently our approach does not directly support typed domains. Instead, types are represented by adding new fluents to the domain description. As with the introduction of new objects, these additional fluents increase the number of observations needed to learn the domain. However, by supporting typed domains, we could also significantly improve the performance of our approach.

Discussion

Our method is dependent on the existence of a structured parametric representation, abstracted from the grounded sensory manifold itself. In particular, actions must be specified whose parameters are exactly those objects acted on, so that the correct set of objects is passed into the reduced state vector. We anticipate such information would be provided by an attentional model applied to a visual scene which picks out the necessary actions and their parameters. Such a model exists (Satish and Mukerjee 2008), and could be used to provide grounded parameterised actions.

Our method also depends on the use of a deictic representation, which both introduces a bias for generalisation and limits the number of objects considered by the learning and prediction process. Deictic representations have previously been applied to learning domain dynamics. (Benson 1996) converts the first-order logic description of the state space into a propositional description by representing objects with deictic variables. Our use of deictic variables is essentially the same. However (Benson 1996) uses the transitive closure of the relations among objects rather than the single step computation which we use. Similarly, (Pasula, Zettlemoyer, and Kaelbling 2007) use deictic references to objects to provide a generalisation bias and to reduce the search space of transition rules.

Our training and testing data was generated to mimic data collected by an agent exploring the world, and corresponds to sequences of actions and observations taken from random walks through the state space. In some domains (e.g. Grid, Freecell) certain actions occur infrequently, if at all, under these conditions, and so learning of such actions may fail. A more guided exploration of the state space may be necessary to learn in these domains. In particular, we do not use exam-



(a) Learning in fully observable domains

(b) Partially observable BlocksWorld

(c) Partially observable ZenoTravel

(d) Partially observable Depots

Figure 3: Results of learning action models in standard planning domains

ple plans as training data, since this presupposes an existing model of the domain which is to be learnt.

Tractability is a significant issue in learning action models in partially observable domains: methods such as HMMs, Dynamic Bayes Nets and Reinforcement Learning scale poorly (Amir and Chang 2008). More tractable methods use schema learning (Holmes and Isbell 2005), build a CNF representation of all possible transition models (Amir and Chang 2008), or convert the problem into a weighted MAXSAT problem (Yang, Wu, and Jiang 2007). In terms of tractability, our approach is competitive with these methods. A direct comparison is not straightforward as our method currently does not produce explicit rules which could be compared.

We learn action models in partially observable *noiseless* domains. Our approach also performs well in fully observable *noisy* domains (submitted). Figure 4 shows the results of our method applied to learning the ZenoTravel domain with 5% and 10% uniform noise, under full observability. We believe our approach will extend to learning in noisy partially observable domains. Future work will investigate this claim. Some earlier work learns action models in probabilistic, partially observable, noisy domains using schema learning (Holmes and Isbell 2005). However, the action models apply to sensor values rather than features of the domain, and so objects and relations between objects are not modelled. Other work uses noiseless domains (Amir and Chang 2008; Yang, Wu, and Jiang 2007), and these methods have not been shown to work in noisy domains.

The form of partial observability can also vary. We follow (Amir and Chang 2008) where a fixed number of randomly chosen fluents are observed after each action in a random sequence of actions. Additionally, (Amir and Chang 2008) do not require knowledge of an initial state, fluents, or the size of the domain in their approach. In contrast, our method currently requires prior knowledge of the fluents and objects in the domain in order to build the state vector representation. (Yang, Wu, and Jiang 2007) describe a different form of partial observability, where the full state is observed at intervals after a fixed number of actions in a plan are executed, in combination with knowledge of the initial state and goal state. Since we rely on observing state changes before and after action application, our approach is not directly applicable to this form of partial observability.

We also believe the form of partial observability we use allows our model to be extended to sensing actions. While prior approaches (including ours) have primarily focused on learning the effects of ordinary actions—actions which

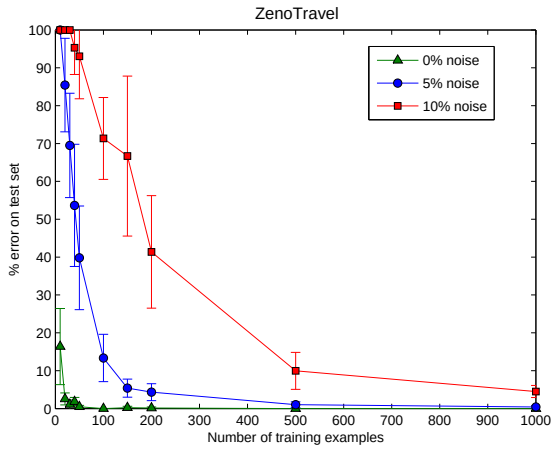


Figure 4: Learning results in noisy and fully observable versions of the ZenoTravel domain. Noise at level p% was simulated by flipping each bit in the state vector with probability p (for $p = 0\%$, $p = 5\%$ and $p = 10\%$). The test sets were noiseless, fully observable sequences of observations and actions, of length 2000.

change both the world state as well as the agent's knowledge state—they have ignored the role of *sensing actions* in partially observable domains. Sensing actions are particularly useful since they provide an agent with information about the state of the world, thereby changing the agent's knowledge state, but without necessarily changing the world state. We admit sensing actions into our account by assuming such actions have parameters (fluents and parameters of the fluents) and that these actions only alter knowledge relating to those parameters, or objects directly related to the parameters of the fluents. Furthermore we disallow sensing actions which are non-deterministic or have disjunctive effects. Then a reduced knowledge state vector can be constructed in the same manner as the reduced world state vector. In the input, the value of each bit indicates whether the corresponding fluent is known or not, and in the output indicates changes to the knowledge state. The learning model can then operate on both the knowledge and world state vectors. Standard actions are learnt as before, with effects now including changes to the knowledge state. Sensing actions are learnt in the same way, but changes to the world state will have to be ignored, since these are unpredictable. The addition of sensing actions would allow our method to be integrated with knowledge-based representations such as those used by the PKS planner (Petrick and Bacchus 2004).

Finally, the relative difficulty of learning action models in different domains is not well understood. For example, from the domain description, the Depots domain appears to be simpler than the ZenoTravel domain since the maximum number of parameters of any action is lower, and there are fewer predicates. Our results show that Depots is harder to learn, however, at least for our learning method. In fact, the method presented in (Yang, Wu, and Jiang 2007) also produces more errors in the Depots domain than in the ZenoTravel domain, suggesting that the additional difficulty may be a feature of the Depots domain rather than the learning method. In general, further investigation of the relative difficulties of learning different domains is important for further research in this area.

Conclusions and Future Work

We have presented a method for learning deterministic action models which is fast, scalable and handles partial observability of the world state. Furthermore, the error rate of the predictions made by the model is low. The speed, scalability and accuracy make the approach highly suitable for use in planning applications. It is straightforward to extend our method to learn the effects of sensing actions. Our method also performs well in noisy domains, and a key step will be to apply it to partially observable noisy domains.

In future work we plan to link our learning mechanism to a planning system applied to more complex domains, such as the problem of learning and planning actions for the ARMAR-III humanoid robot (Asfour et al. 2006) in a real-world robot environment. We also plan to extend the mechanism to learn more sophisticated action representations beyond STRIPS, such as those requiring functions.

Acknowledgements

This work was partially funded by the European Commission through the PACO-PLUS project (FP6-2004-IST-4-27657) and the UK EPSRC/MRC through the Neuroinformatics Doctoral Training Centre, University of Edinburgh.

References

Agre, P. E., and Chapman, D. 1987. Pengi: an implementation of a theory of activity. In *Proceedings of the Sixth National Conference on Artificial Intelligence (AAAI-87)*, 268–272.

Aizerman, M. A.; Braverman, E. M.; and Rozoner, L. I. 1964. Theoretical foundations of the potential function method in pattern recognition learning. *Automation and Remote Control* 25:821–837.

Amir, E., and Chang, A. 2008. Learning partially observable deterministic action models. *Journal of Artificial Intelligence Research* 33:349–402.

Asfour, T.; Regenstein, K.; Azad, P.; Schröde, J.; and Dillmann, R. 2006. ARMAR-III: A humanoid platform for perception-action integration. In *2nd International Workshop on Human-Centered Robotic Systems (HCRS'06)*.

Ballard, D. H.; Hayhoe, M. M.; Pook, P. K.; and Rao, R. P. 1997. Deictic codes for the embodiment of cognition (with commentary). *Behavioral and Brain Sciences* 20.

Benson, S. S. 1996. *Learning Action Models for Reactive Autonomous Agents*. Ph.D. Dissertation, Stanford University.

Boser, B. E.; Guyon, I. M.; and Vapnik, V. N. 1992. A training algorithm for optimal margin classifiers. In *COLT*

'92: Proceedings of the fifth annual workshop on Computational learning theory, 144–152. New York, NY, USA: ACM Press.

Doğar, M. R.; Çakmak, M.; Uğur, E.; and Şahin, E. 2007. From primitive behaviors to goal directed behavior using affordances. In *Proceedings of Intelligent Robots and Systems (IROS 2007)*.

Fikes, R. E., and Nilsson, N. J. 1971. STRIPS: A new approach to the application of theorem proving to problem solving. *Artificial Intelligence* 2:189–208.

Freund, Y., and Schapire, R. 1999. Large margin classification using the perceptron algorithm. *Machine Learning* 37:277–296.

Gil, Y. 1994. Learning by experimentation: Incremental refinement of incomplete planning domains. In *Proceedings of the International Conference on Machine Learning (ICML-94)*. MIT Press.

Graepel, T.; Herbrich, R.; and Williamson, R. C. 2000. From margin to sparsity. In *Advances in Neural Information Processing Systems (NIPS) 13*, 210–216.

Holmes, M., and Isbell, C. 2005. Schema learning: Experience-based construction of predictive action models. In *Advances in Neural Information Processing Systems (NIPS) 17*, 585–562.

McDermott, D.; Ghallab, M.; Howe, A.; Knoblock, C.; Ram, A.; Veloso, M.; Weld, D.; and Wilkins, D. 1998. PDDL - the planning domain definition language. Technical report, CVC TR-98-003/DCS TR-1165, Yale Center for Computational Vision and Control.

Metta, G., and Fitzpatrick, P. 2003. Early integration of vision and manipulation. *Adaptive Behavior* 11(2):109–128.

Minsky, M. L., and Papert, S. A. 1969. *Perceptrons*. The MIT Press.

Modayil, J., and Kuipers, B. 2008. The initial development of object knowledge by a learning robot. *Robotics and Autonomous Systems* 56(11):879–890.

Mourão, K.; Petrick, R. P.; and Steedman, M. 2008. Using kernel perceptrons to learn action effects for planning. In *Proceedings of the International Conference on Cognitive Systems (CogSys 2008)*.

Novikoff, A. B. 1963. On convergence proofs for perceptrons. In *Proceedings of the Symposium on the Mathematical Theory of Automata*, volume 12, 615–622.

Pasula, H. M.; Zettlemoyer, L. S.; and Kaelbling, L. P. 2007. Learning symbolic models of stochastic world domains. *Journal of Artificial Intelligence Research* 29:309–352.

Petrick, R. P. A., and Bacchus, F. 2004. Extending the knowledge-based approach to planning with incomplete information and sensing. In *Proceedings of Automated Planning and Scheduling (ICAPS-04)*, 2–11. AAAI Press.

Rosenblatt, F. 1958. The perceptron: a probabilistic model for information storage and organization in the brain. *Psychological Review* 65(6):386–408.

Satish, G., and Mukerjee, A. 2008. Acquiring linguistic argument structure from multimodal input using attentive focus. In *Development and Learning, 2008. ICDL 2008. 7th IEEE International Conference on*, 43–48.

Slaney, J., and Thiébaux, S. 2001. Blocks world revisited. *Artificial Intelligence* 125(1-2):119–153.

Surdeanu, M., and Ciaramita, M. 2007. Robust information extraction with perceptrons. In *Proceedings of the NIST 2007 Automatic Content Extraction Workshop (ACE07)*.

Wang, X. 1995. Learning by observation and practice: An incremental approach for planning operator acquisition. In *Proceedings of the International Conference on Machine Learning (ICML-95)*, 549–557.

Yang, Q.; Wu, K.; and Jiang, Y. 2007. Learning action models from plan examples using weighted MAX-SAT. *Artificial Intelligence* 171(2-3):107–143.

Appendix E

Combining Cognitive Vision, Knowledge-Level Planning with Sensing, and Execution Monitoring for Effective Robot Control

Ronald P. A. Petrick
School of Informatics
University of Edinburgh
Edinburgh EH8 9AB, Scotland, UK
rpetrick@inf.ed.ac.uk

Dirk Kraft **Norbert Krüger**
The Maersk Mc-Kinney Moller Institute
University of Southern Denmark
DK-5230 Odense M, Denmark
{kraft,norbert}@mmmi.sdu.dk

Mark Steedman
School of Informatics
University of Edinburgh
Edinburgh EH8 9AB, Scotland, UK
steedman@inf.ed.ac.uk

Abstract

We describe an approach to robot control in real-world environments that integrates a cognitive vision system with a knowledge-level planner and plan execution monitor. Our approach makes use of a formalism called an Object-Action Complex (OAC) to overcome some of the representational differences that arise between the low-level control mechanisms and high-level reasoning components of the system. We are particularly interested in using OACs as a formalism that enables us to induce certain aspects of the representation, suitable for planning, through the robot's interaction with the world. Although this work is at a preliminary stage, we have implemented our ideas in a framework that supports object discovery, planning with sensing, action execution, and failure recovery, with the long term goal of designing a system that can be transferred to other robot platforms and planners.

Introduction and Motivation

A robot operating in a real-world domain must typically rely on a range of mechanisms that combine both reactive and planned behaviour, and operate at different levels of representational abstraction. Building a system that can effectively perform these tasks requires overcoming a number of theoretical and practical challenges that arise from integrating such diverse components within a single framework.

One of the crucial aspects of the integration task is representation: the requirements of robot controllers differ from those of traditional planning systems, and neither representation is usually sufficient to accommodate the needs of an integrated system. For instance, robot systems often use real-valued representations to model features like 3D spatial coordinates and joint angles, allowing robot behaviours to be specified as continuous transforms of vectors over time (Murray, Li, and Sastry 1994). On the other hand, planning systems tend to use representations based on discrete, symbolic models of objects, properties, and actions, described in languages like STRIPS (Fikes and Nilsson 1971) or PDDL (McDermott 1998). Overcoming these differences is essential for building a system that can act in the real world.

In this paper we describe an approach that combines a *cognitive vision* system with a *knowledge-level planner* and *plan execution monitor*, on a robot platform that can manipulate objects in a restricted, but uncertain, environment. Our system uses a multi-level architecture that mixes a low-level robot/vision controller for object manipulation and scene interpretation, with high-level components for reasoning, planning, and action failure recovery. To overcome the modelling differences between the different system components, we use a representational unit called an *Object-Action Complex (OAC)* (Geib et al. 2006; Krüger et al. 2009), which arises naturally from the robot's interaction with the world. OACs provide an object/situation-oriented notion of affordance in a universal formalism for describing state change.

Although the idea of combining a robot/vision system with an automated planner is not new, the particular components we use each bring their own strengths to this work. For instance, the cognitive vision system (Krüger, Lappe, and Wörgötter 2004; Pugeault 2008) provides a powerful object discovery mechanism that lets us induce certain aspects of the representation, suitable for planning, from the robot's basic "reflex" actions. The high-level planner, PKS (Petrick and Bacchus 2002; 2004), is effective at constructing plans under conditions of incomplete information, with both ordinary physical actions and *sensing* actions. Moreover, OACs occur at all levels of the system and, we believe, provide a novel solution to some of the integration problems that arise in our architecture.

This paper reports on work currently in progress, centred around OACs and their role in object discovery, planning with sensing, action execution, and failure recovery in uncertain domains. This work also forms part of a larger project investigating perception, action, and cognition, and combines multiple robot platforms with symbolic representations and reasoning mechanisms. We have therefore approached this work with a great deal of generality, in order to facilitate the transfer of our ideas to robot platforms and planners with capabilities other than those we describe here.

Hardware Setup and Testing Domain

The hardware setup used in this work (see Figure 1) consists of a six-degree-of-freedom industrial robot arm (Stäubli RX60) with a force/torque (FT) sensor (Schunk FTACL 50-80) and a two-finger-parallel gripper (Schunk PG 70) attached. The FT sensor is mounted between the robot arm and gripper and is used to detect collisions which might occur due to limited knowledge about the objects in the world. In addition, a calibrated stereo camera system is mounted in a fixed position. The AVT Pike cameras have a resolution

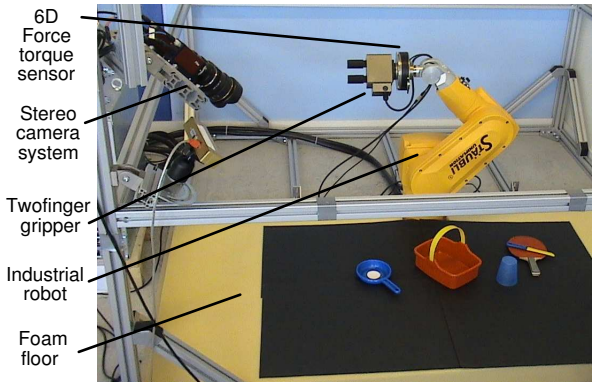


Figure 1: Hardware setup.

of up to 2048x2048 pixels and can produce high-resolution images for particular regions of interest.

To test our approach, we use a Blocksworld-like object manipulation scenario. This domain consists of a table with a number of objects on it and a "shelf" (a special region of the table). The robot can view the objects in the world but, initially, does not have any knowledge about those objects. Instead, world knowledge must be provided by the vision system, the robot's sensors, and the primitive actions built into the robot controller. The robot is given the task of clearing the objects from the table by placing them onto the shelf. The shelf has limited space so the objects must be stacked in order to successfully complete the task. For simplicity, each object has a radius which provides an estimate of its size. An object A can be stacked into an object B provided the radius of A is less than that of B, and B is "open." Unlike standard Blocksworld, the robot does not have complete information about the state of the world. Instead, we consider scenarios where the robot does not know whether an object is open or not and must perform a test to determine an object's "openness". The robot also has a choice of four different grasping types for manipulating objects in the world. Not all grasp types can be used on every object, and certain grasp types are further restricted by the position of an object relative to other objects in the world. Finally, actions can fail during execution and the robot's sensors may return noisy data.

Basic Representations and OACs

At the robot/vision level, the system has a set Σ of sensors, $\Sigma = \{\sigma_1, \sigma_2, \ldots, \sigma_n\}$, where each sensor σ_i returns an observation $obs(\sigma_i)$ about some feature of the world, represented as a real-valued vector. The execution of a robot-level action, called a *motor program*, may cause changes to the world which can be observed through subsequent sensing. Each motor program is typically executed with respect to particular *objects* in the world. We assume that initially the robot/vision system does not know about any objects and, therefore, can't execute many motor programs. Instead, the robot has a set of object-independent *basic reflex actions* which it can use in conjunction with the vision system for early exploration and object discovery.

At the planning level, the underlying representation is based on a set of *fluents*, $f_1, f_2, \ldots, f_m$: first-order predicates and functions that denote particular qualities of the world, robot, and objects. Fluents typically represent high-level versions of some of the world-level properties the robot is capable of sensing, where the value of a fluent is a function Γ_i of a set of observations returned by the sensor set, i.e., $f_i = \Gamma_i(\Sigma)$. However, in general, not every sensor need map to some fluent, and we allow for the possibility of fluents with no direct mapping to robot-level sensors.

Fluents may be parametrized and instantiated by high-level counterparts of the objects discovered at the robot level. In particular, for each robot-level object obj^r we denote a corresponding high-level object by obj^p. A *state* is a snapshot of the values of all instantiated fluents at some point during the execution of the system, i.e., $\{f_1, f_2, \ldots, f_m\}$. States represent an intersection between the low-level and high-level representations and are *induced* from the sensor observations (the Γ_i functions) and the object set.

The planning level representation also includes a set of high-level *actions*, $\alpha_1, \alpha_2, \ldots, \alpha_p$, which are viewed as abstract versions of some of the robot's motor programs. Since all actions must ultimately be executed by the robot, each action is decomposable to a fixed set of motor programs $\Pi(\alpha_i)$, where $\Pi(\alpha_i) = \{mp_1, mp_2, \ldots, mp_l\}$, and each mp_j is a motor program. As with fluents, not every robot-level motor program need map to a high-level action.

Although the robot/vision and planning levels use quite different representations (i.e., real-valued vectors versus logical fluents), the notions of "action" and "state change" are common among these components. To capture these similarities, we model our actions and motor programs using a structure called an *Object-Action Complex (OAC)* (Geib et al. 2006; Krüger et al. 2009). Formally, an OAC is a tuple $\langle I, T^S, M \rangle$, where I is an identifier label for the OAC, $T : S \rightarrow S$ is a transition function over a state space S, and M is a statistic measure of the accuracy of the transition. OACs provide a universal "container" for encapsulating the relationship between actions (operating over objects) and the changes they make to their state spaces. Each OAC also has an identical set of predefined operations (e.g., composition, update, etc.), providing a common interface to these structures. Since robot systems may have many components, OACs are meant to provide a standard language for describing action-like processes (including continuous processes) within these components, and to simplify the exchange of information between different components.

OACs exist at each level of our system. We encode each motor program on the robot/vision level and each action at the planning level as a separate OAC, with OACs at each level having a different underlying state space. By assigning an accuracy metric to each OAC we also capture the nondeterministic nature of our actions in the real world. Furthermore, since every interaction of the robot with the world provides the robot with an opportunity to observe a small portion of the world's state space (interpreted with respect to the state space of a particular OAC), we can make use of this information to refine or improve the accuracy of the OACs at all levels of our system.

Typically, we consider OACs that are formed from *partial* state descriptions, which may have low reliability. Such descriptions arise since the robot cannot always sense the status of all objects and properties in the world (e.g., occluded or undiscovered objects). Furthermore, the robot's sensors may be noisy and, thus, there is no guarantee that sensor observations are always correct. Certain sensors also have associated resource costs (e.g., time, energy, etc.) which limit their execution. For instance, our robot can perform a test to determine whether an object is open by "poking" the object to check its concavity. Such operations are only initiated on demand at the discretion of the high-level planning system.

Finally, our system includes a middle level component that mediates between the robot and planning levels. This component is responsible for mapping between OACs at different levels of the system (i.e., implementing the Γ_i and Π functions) in order to ensure that observation/state and motor program/action information passing between levels is translated into a form that the destination level understands.

In the remainder of this paper we will look at the main components of our system in greater detail, and describe the current (and future) role of OACs in our framework.

Vision-Based Object Discovery

The visual representation used by the lower level of our system is delivered by an early cognitive vision system (Krüger, Lappe, and Wörgötter 2004; Pugeault 2008) which creates sparse 2D and 3D features, so-called *multi-modal primitives*, along image contours from stereo images. 2D features represent a small image patch in terms of position, orientation, phase, colour and optical flow. These are matched across two stereo views, and pairs of corresponding 2D features permit the reconstruction of an equivalent 3D feature. 2D and 3D primitives are then organized into perceptual groups in 2D and 3D. The procedure to create visual representations is illustrated in Figure 2. We note that the resulting representation not only contains appearance information (e.g., colour and phase) but also geometrical information (i.e., 2D and 3D position and orientation).

Initially, the system lacks knowledge of the objects in a scene and so the visual representation is unsegmented: descriptors that belong to one object are not explicitly distinct from the ones that belong to other objects, or the background. To aid in the discovery of new objects, the robot is equipped with a basic reflex action (Aarno et al. 2007) that is elicited by specific visual feature combinations in the unsegmented world representation (e.g., see Figure 3(a)–(c)). The outcome of these reflexes allows the system to gather knowledge about the scene, which is used to segment the visual world into objects and identify basic affordances. We consider a reflex where the robot tries to grasp a planar surface in the scene. Each time the robot executes such a reflex, haptic information allows the system to evaluate the outcome: either the grasp was successful and the gripper is holding something, or it failed and the gripper simply closed.

With physical control, the system visually inspects an object from a variety of viewpoints and builds a 3D representation (Kraft et al. 2008). Features on the object are tracked over multiple frames, between which the object moves with a known motion. If features are constant over a series of frames they become included in the object's representation; otherwise they are assumed to not belong to the object. (See Figure 3(d)–(f) and (Kraft et al. 2008) for a more detailed explanation.) The final description is labelled and recorded as an identifier for a new object class, along with the successful reflex (now a motor program). Using this new knowledge, the system then reconsiders its interpretation of the scene: using a representation-specific pose estimation algorithm (Detry, Pugeault, and Piater 2009) all other instances of the same object class are identified and labelled. By repeating this process, the system constructs a representation of the world objects, as instances of symbolic classes that carry basic affordances, i.e., particular reflex actions that have been successfully applied to objects of this class.[1] This relationship can also be interpreted as a new low-level OAC.

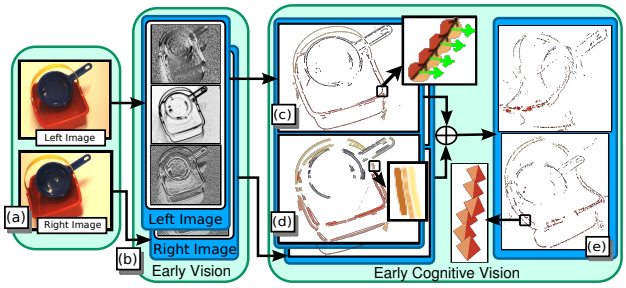


Figure 2: An overview of the visual representation. (a) Stereo image pair, (b) Filter responses, (c) 2D primitives, (d) 2D contours, (e) 3D primitives.

The object-centric nature of the robot's world exploration process has immediate consequences for the high-level representation. First, newly discovered objects are reported to the planning level and added to its representation. At this level, objects are simply labels that act as indices to the object information stored at the robot level. Such a representation means that the planner can avoid reasoning about certain types of real-valued information (e.g., 3D coordinates, orientation vectors, etc.) and instead refer to objects by their labels (e.g., $obj1^p$ may denote a particular red cup on the table). Second, the planner can immediately use such objects during plan generation. Since we assume that object names do not change over time, plans with object references will be understandable to the lower system levels. Finally, the identification of new objects will cause the robot/vision system to start sending regular updates about the state of objects and their properties to the planning level. In particular, low-level observations resulting from subsequent interactions with the world will contain state information about these objects, pro-

[1] We have recently completed the technical implementation of the pose estimation algorithm. Prior to this, a circle detection algorithm was developed (Başeski, Kraft, and Krüger 2009) to recognise cylindrical objects. Four grasp templates were used to define the primitive reflex actions in an object-centric way (where concrete grasps were generated based on the object pose). Although this approach negates the need for the general pose estimation algorithm, the conclusions drawn from experiments in this limited scenario are still easily transferable to the general case.

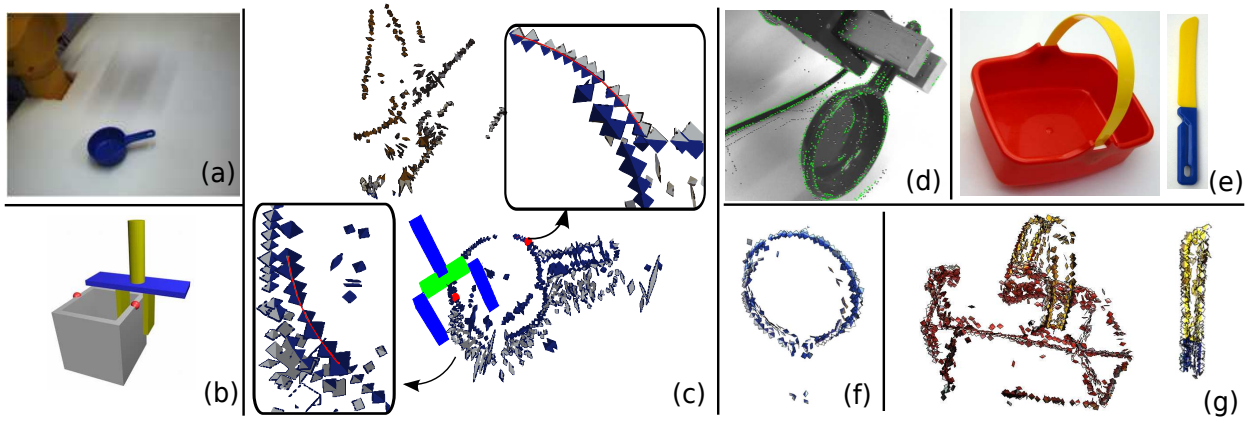


Figure 3: (a)–(c) Initial grasping behaviour: (a) A Scene, (b) Definition of a possible grasp based on two contours, (c) Representation of the scene with contours generating a grasp. (d)–(f) Accumulation process ("birth of the object"): (d) One step in the process. The dots on the image show the predicted structures. Both spurious primitives, parts of the background that are not confirmed by the image, and the confirmed predictions are shown, (e) Images of objects, (f),(g) Extracted models.

vided they can be sensed by the robot.

Knowledge-Level Planning with Sensing

The high-level planner constructs plans that direct the behaviour of the robot to achieve a set of goals. Plans are built using PKS ("Planning with Knowledge and Sensing") (Petrick and Bacchus 2002; 2004), a conditional planner that can operate with incomplete information and sensing actions. Like other symbolic planners, PKS requires a goal, a description of the initial state, and a list of the available actions. Unlike classical planners, PKS operates at the *knowledge level* by explicitly modelling what the planner knows and does not know about the state of the world. PKS can reason efficiently about certain restricted types of knowledge, and make effective use of features like functions, which often arise in real-world scenarios.

PKS is based on a generalization of STRIPS (Fikes and Nilsson 1971). In STRIPS, a single database represents the world state; actions update this database in a way that corresponds to their effects on the world. In PKS, the planner's knowledge state is represented by five databases, each of which stores a particular type of knowledge. Actions describe the changes they make to the database set and, thus, to the underlying knowledge state. PKS also supports ADL-style conditional action effects (Pednault 1989), numerical reasoning, and a set of program-like control structures.

Table 1 shows an example of some of the PKS actions available in the testing domain. As in standard planning representations, like PDDL, actions in PKS are described by their preconditions and effects. Actions may be parametrized (e.g., $graspA(x)$), with an action's parameters replaced with references to specific world objects when an action is instantiated in a plan. As we described above, objects at the planning level are labels to actual objects identified by the robot/vision system.

Preconditions and effects are specified in terms of a set of high-level predicates and functions, i.e., fluents that model particular qualities of the world, robot, and objects. For instance, the actions in Table 1 include references to fluents:

- $open(x)$: object x is open,
- *gripperEmpty*: the robot's gripper is empty,
- $onTable(x)$: object x is on the table,
- $isIn(x, y)$: object x is stacked in object y,
- $radius(x) = y$: the radius of object x is y, and
- $reachableX(x)$: object x is reachable using grasp type X,

among others. While most high-level properties abstract the information returned by the robot-level sensors (e.g., *onTable* requires data from a set of visual sensors concerning object positions), some properties correspond more closely to individual sensors (e.g., *gripperEmpty* closely models a low-level sensor that detects whether the robot's gripper can be closed without contact).

One significant difference between PKS and other planners is that all actions in PKS are modelled at the knowledge level: preconditions denote conditions that must be true of the planner's knowledge state while effects describe changes to what the planner knows. For instance, precondition expressions of the form $K(\phi)$ denote a knowledge-level query that asks "does the planner know ϕ to be true?" while an expression like $K_w(\phi)$ asks "does the planner know whether ϕ is true or not?" Effect expressions of the form $add(D, \phi)$ assert that ϕ should be added to database D, while $del(D, \phi)$ means that ϕ should be removed from database D. In Table 1, K_f refers to a database that models the planner's definite knowledge of facts, while K_w is a specialized database that stores the results of sensing actions that return binary information.

In our robot scenario, high-level actions represent counterparts to some of the motor programs available at the robot level. For instance, the planner has access to actions like:

- $graspA(x)$: grasp x from the table using grasp type A,
- $graspD(x)$: grasp x from the table using grasp type D,
- $putInto(x, y)$: put x into y on the table,

Action	Preconditions	Effects
$graspA(x)$	$K(reachableA(x))$	$add(K_f, inGripper(x))$
	$K(gripperEmpty)$	$add(K_f, \neg gripperEmpty)$
	$K(onTable(x))$	$add(K_f, \neg onTable(x))$
	$K(clear(x))$	
	$K(radius(x) \geq minA)$	
	$K(radius(x) \leq maxA)$	
$graspD(x)$	$K(reachableD(x))$	$add(K_f, inGripper(x))$
	$K(gripperEmpty)$	$add(K_f, \neg gripperEmpty)$
	$K(onTable(x))$	$add(K_f, \neg onTable(x))$
	$K(radius(x) \leq maxD)$	
$putInto(x, y)$	$K(x \neq y)$	$add(K_f, gripperEmpty)$
	$K(inGripper(x))$	$add(K_f, isIn(x, y))$
	$K(open(y))$	$add(K_f, clear(y))$
	$K(clear(y))$	$add(K_f, \neg inGripper(x))$
	$K(onTable(y))$	
	$K(radius(y) > radius(x))$	
$putAway(x)$	$K(inGripper(x))$	$add(K_f, onShelf(x))$
	$K(shelfSpace > 0)$	$add(K_f, gripperEmpty)$
		$add(K_f, \neg inGripper(x))$
		$add(K_f, shelfSpace -= 1)$
$findout\text{-}open(x)$	$\neg K_w(open(x))$	$add(K_w, open(x))$
	$K(onTable(x))$	

Table 1: PKS actions in the testing domain.

- *putAway*(x): put x away onto a shelf space, and
- *findout-open*(x): determine whether x is open or not,

among others. Some actions like "grasp" are divided into multiple actions (e.g., *graspA*, *graspD*, plus actions for grasp types B and C). The object-centric nature of these actions means they do not require 3D coordinates, joint angles, or similar real values but, instead, include parameters that can be instantiated with specific objects. Actions like *putInto* and *putAway* account for different object/location configurations, although the motor programs that implement these actions do not necessarily make such distinctions. (The complete action list has a larger set of such actions.) The *findout-open* action is an example of a high-level *sensing action* that directs the robot to gather information about the world state that is not normally provided as part of its regular sensing cycle. From the planner's point of view, an action's sensory effects are assumed to only change the planner's knowledge state, while leaving the world state unchanged.

Each planning level action is treated as an individual OAC with its own identifier and transition function corresponding to the action's preconditions and effects. All planning level OACs share a common state space consisting of the high-level predicates and functions. Each OAC also maintains a measure, M, of its reliability, which is updated by the plan execution monitor (see below). Currently, PKS does not use this information (or any probabilistic measures) during plan generation, but instead relies on its ability to reason about incomplete information and replan from action failure.

As an example, consider the situation in the testing domain where two unstacked and open objects $obj1^p$ and $obj2^p$ are on a table, the planner can construct the following plan for clearing all *open* objects from the table:

$$\begin{array}{l} graspD(obj2^p), \\ putInto(obj2^p, obj1^p), \\ graspD(obj1^p), \\ putAway(obj1^p). \end{array}$$

In this plan, $obj2^p$ is grasped from the table using grasp type D (an overhand grasp) and put into $obj1^p$, before the stacked objects are grasped and removed to the shelf.

The planner can also build more complex plans using sensing actions. For instance, if the planner is given the goal of removing the *open* objects from the table in the example scenario, but does not know whether object $obj3^p$ is open or not, then it might construct the conditional plan:

$$\begin{array}{l} findout\text{-}open(obj3^p), \\ branch(open(obj3^p)) \\ K^+ : \\ \quad graspA(obj3^p), \\ \quad putAway(obj3^p) \\ K^- : \\ \quad nil. \end{array}$$

This plan senses the truth value of the predicate $open(obj3^p)$ using *findout-open* and reasons about the possible outcome of this action. As a result, two branches are included in the plan denoting potential execution paths: if $open(obj3^p)$ is true (the K^+ branch) then $obj3^p$ is grasped and put away; if $open(obj3^p)$ is false (the K^- branch) then no action is taken.

State Generation and OAC Interaction

From an integration point of view, the robot/vision system is linked to the planning level through a component which mediates between the state spaces and OACs used by the two levels of the system. Since the planner is not able to handle raw sensor data as a state description, or directly control the robot, the low-level observations generated by the robot/vision system must be abstracted into a language the planner understands, and planned actions must be converted into appropriate robot-level motor programs.

For state space information, sensor data is "wrapped" and reported to the planner in the form of a fluent-based symbolic state representation that includes predicates and functions. Currently, the mappings between certain sensor combinations and the corresponding high-level fluents (i.e., the Γ_i functions) are simply hardcoded. For example:

- *inGripper*, *gripperEmpty*: Initially the gripper is empty and the predicate *gripperEmpty* is formed. As soon as the robot grasps an object ($objX^r$), and confirms that the grasp is successful by means of the gripper not closing up to mechanical limits, the system knows that it has the object in its hand and can form a predicate $inGripper(objX^p)$. Releasing the object returns the gripper to an empty state.
- *reachableX*: Based on the position of a circle forming the top of a cylindrical object in the scene we can compute possible grasp positions (for the different grasp types) for each object. Using standard robotics path planning methods we then compute whether or not there is a collision-free path between the start position and the gripper pose needed to reach the object for a particular grasp.

- *open*: Objects are not assumed to be "open." Unlike the above properties which are determined directly from ordinary sensor data, the robot must perform an explicit test to determine an object's openness. In this case, the robot attempts to use its gripper to "poke" inside the potential opening of an object. If the robot encounters a collision (determined by the FT sensor), the object is assumed to be closed. Otherwise, we assume the object is open.

To compute these predicates, the mediator interacts with the robot/vision system to maintain a snapshot of the current world state which, besides the state information necessary for the planner, also contains information needed for consistency and action computations. In particular, object positions are represented here. To cope with sensor noise (especially the vision-based information about the number and location of circles) a simple mechanism to avoid spurious object disappearance and appearance is employed.

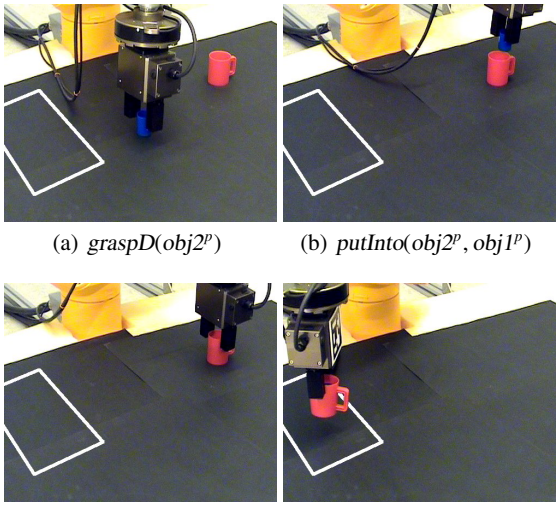
From the planner's point of view, it begins operation without any information about the state of the world. After an initial exploration of the environment, the robot/vision system begins to gather observations and generate (partial) state reports about the current set of objects it believes to be in the world, along with the properties it senses for those objects. This observation set (converted into a fluent-based representation) is then sent to the planner and used as its initial (incomplete) knowledge state: the predicate and function instances are treated as *known* state information, with all other state information considered to be unknown. Subsequent state reports are interpreted by the plan monitor (see below) and used to update the reliability of high-level OACs.

High-level planning actions, in the form of OACs, must also be mapped to their appropriate low-level counterparts, for execution by the robot system in the real world. We currently assume that the set of action schema is supplied to the planner as part of its input, as are the mappings from planning actions to robot motor programs (the Π function).

For instance, the high-level OAC *graspD* is realised on the lowest level as a mapping to an object-independent OAC, $graspD^r$.[2] This low-level OAC requires the object position (retrieved using the object label as an index) as an input to computing suitable grasping positions. The preconditions of this OAC require that there be a grasping position on the brim of the object for which a collision free path from the current position to the grasp position exists. The motor program associated with this OAC is a motion sequence that first completely opens the gripper's fingers, followed by a movement of the arm along the joint trajectory and, lastly, closes the fingers and lifts the arm. After the motor program has been executed the expected outcome state expresses that the fingers should no longer be totally open nor totally closed. In this case, closed fingers indicate that the action failed and no object has been grasped.

Plan Execution and Failure Recovery

Once a plan is generated, the planning level interacts with the robot/vision level (through the mid-level mediator) to execute the plan. Actions are sent to the robot one at a time, where they are converted into motor programs and executed in the world. A stream of observations is also generated, arising from the executed motor programs, and processed into high-level state information. Upon action completion the robot/vision level returns this information to the higher reasoning levels, along with an indication of the success or failure of the action which are used to update the reliability measure M of the high-level OACs. The execution cycle then continues. For instance, Figure 4 shows the execution of the four step plan described above for clearing a table.



(a) $graspD(obj2^p)$ (b) $putInto(obj2^p, obj1^p)$

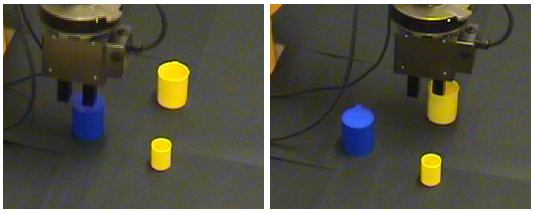
(c) $graspD(obj1^p)$ (d) $putAway(obj1^p)$

Figure 4: Executing a high-level plan to clear a table.

An essential component in this process is the *plan execution monitor*, which assesses action failure and unexpected state information resulting from feedback provided to the planner from the execution of planned actions at the robot level. The execution monitor operates in conjunction with the planner and mid-level mediator, and is responsible for controlling replanning and resensing activities in the system. In particular, the difference between predicted and observed states are used to decide between (i) continuing the execution of an existing plan, (ii) asking the vision system to resense a portion of a scene at a higher resolution in the hope of producing a more detailed state report, and (iii) replanning from an unexpected state using the current state report as a new initial planning state. The plan execution monitor also has the important task of managing the execution of plans with conditional branches, resulting from the inclusion of high-level sensing actions. In each case, the decision of the monitor depends on the type of action being processed and the state information returned by the robot.

Continuing a plan's execution During plan execution, actions are delivered to the lower control levels for execution on the robot. After the execution of each action, a state report representing the *observed* state of the world is returned to the plan monitor and compared against the planner's *predicted* state as constructed during planning, to determine if plan execution should continue or resensing/replanning

[2] In general, a high-level OAC may be realised by multiple robot-level OACs.



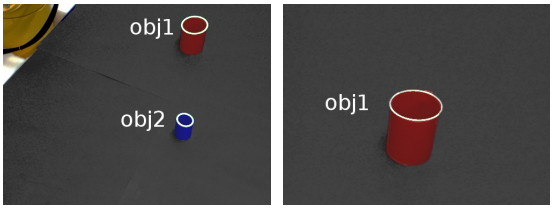
(a) Object is not open (b) Object is open

Figure 5: Testing the openness of an object.

should be activated. Since states in our testing domain tend to be partial, we currently use a limited horizon lookahead method, that attempts to verify that the preconditions for the next n actions in the plan are satisfied in the current (partial) state, and the states that follow when the predicted effects of those actions are applied. (In our testing domain $n = 1$ is often sufficient to ensure good performance.) This means that it is possible for an action to only achieve some of its effects and for the plan to continue, provided the action did not report that it outright failed, and the state is sufficiently correct to ensure the execution of the next action in the plan. (Thus, we defer possible replanning over plan continuation if possible.) If a state match is successful, the monitor then proceeds with the current plan. Otherwise, resensing is considered as a secondary test before replanning (see below).

Sensing actions and conditional plan execution The plan execution monitor also has the added task of managing the execution of plans with sensing actions and associated conditional plan branches. When a high-level sensing action is encountered in a plan it is sent to the robot/vision level like any other action and executed on the robot (as determined by the Π mappings). The actual execution of a sensing action is left to the lower control level which can make more informed decisions about motor program execution. For instance, the *findout-open* action in our example domain is executed at the robot level as a combination of "physical" action (e.g., "poking" an object to determine its openness) and "observational" action (i.e., observing the result); as far as the planner is concerned, the action is executed under the assumption that it is *knowledge producing* and will return an expected piece of information. (Figure 5 shows the execution of *findout-open* by the robot in the case where (a) an object is not open and (b) an object is open.) The sensing result will subsequently be observed by the robot system and returned to the planner as part of the state update cycle.

Plans may also have conditional branch points resulting from sensing actions. When faced with a branch in a plan, the plan execution monitor makes a decision as to the correct plan branch it should execute, based on its current knowledge state. If only partial state information is available, but the required information needed for branch determination is missing (e.g., due to a failure at the robot/vision level), resensing or replanning is triggered. For instance, the example conditional plan given above includes the branch point $branch(open(obj3^p))$, i.e., branch on the truth of the fluent $open(obj3^p)$. If $open(obj3^p)$ is true according to the planner's knowledge state then the "positive" (K^+) branch of the plan is followed and the next action is considered; if



(a) (b)

Figure 6: Resensing the scene using the region of interest capabilities of the high resolution cameras.

$\neg open(obj3^p)$ is true then the "negative" (K^-) branch is followed. If the planner has no information about $open(obj3^p)$ then replanning or resensing is activated. It is important to note that the robot/vision system will never be aware of the conditional nature of a plan, and will never receive a "branch" action. From the point of view of the robot, it will only receive a sequential stream of actions.

Resensing at the monitoring level Sensing also plays a role during plan monitoring as a strategy for improving the monitor's accuracy. When the monitor has determined an action's predicted effects do not match the observed state, resensing is considered. At this point, the accuracy of the action's predictions are checked by comparing the M component of the high-level OAC, weighted together with the M components of the OACs of the underlying motor programs which implement this action (the Π mapping), against a threshold value. If the accuracy measure falls below the threshold (i.e., the predictions are considered too spurious), then replanning is activated; otherwise, resensing is performed.

When resensing is required, the plan monitor provides the vision system with a list of the objects considered relevant to the execution of the action that is reported to have failed, based on the parameters in the high-level action description. This information lets the vision system use its high resolution camera to target particular regions of interest in the scene with greater resolution, to reevaluate the sensors that provide information about these objects. New state information returned by this operation may help the monitor decide between continuing a plan's execution and replanning.

For instance, Figure 6(a) shows the state of the world before the $graspD(obj2^p)$ action in our example plan for clearing a table is executed and $obj2^p$ is grasped; both objects in the scene are correctly detected and identified. After executing $graspD(obj2^p)$, however, it is possible that $obj1^p$ may no longer be detected, leading the monitor to resense both $obj1^p$ and $obj2^p$ since the next action in the plan, $putInto(obj2^p, obj1^p)$, depends on these two objects. In Figure 6(b), the old position of $obj1^p$ is resensed, leading to a rediscovery of the object. The old position of $obj2^p$ is also resensed to confirm that it is no longer on the table. In this case, the conditions in the state are sufficient for the monitor to decide that the next action in the plan can be executed.

Replanning When the monitor determines that an action has failed based on the available (resensed) state information, a new plan is constructed for the given goal using the current state as the planner's new initial knowledge state. We use rapid replanning techniques, rather than plan repair, due

to the success of planners like FF-Replan (Yoon, Fern, and Givan 2007). This technique also provides a way of overcoming PKS's inability to work with probabilistic representations: if a plan fails we direct PKS to construct an alternate plan for achieving the goal. So far this technique has proven to be effective during testing in our example domain.

Discussion and Conclusions

We believe OACs provide a useful tool for overcoming some of the challenges surrounding the representation of affordances, actions, and state change in real-world robot systems: OACs facilitate the description of different system components in terms of a common representation and common set of interfaces. Although we have grounded many of our system components in terms of the OAC concept, and can describe processes like object discovery and action execution in terms of OACs, our work is preliminary and we have not used this representation to its full potential.

For instance, while our OACs maintain a measure of reliability (i.e., the M measure), this property is not significantly used in our system. We are currently exploring how to improve the reliability of lower-level OACs based on state observations, which could in turn "refine" related higher-level OACs. Closely related to OAC update is the idea of learning completely new OACs. To this end, we are investigating how high-level action schema (i.e., planning level OACs) can be learned directly from (partial) state snapshots provided by the robot level (Mourão, Petrick, and Steedman 2008). Furthermore, we would also like to automatically induce the mapping between OACs at different levels. Thus, the OACs in this paper are not as fully featured as those of (Krüger et al. 2009) and implementing the full set of OAC properties remains a future goal of this work.

The robot/vision components of our system are also being improved. After a recent significant increase in the frequency at which the robot/vision level can provide state updates, we are exploring a more sophisticated mechanism to cope with the sensor noise using multiple consecutive updates. In the future we will also investigate whether a probabilistic framework can increase the reliability of the information provided to the planning level. More work is also needed to properly compare our approach to other existing architectures in the literature.

Although this work is preliminary, we have implemented a framework with all the control mechanisms described here. This has enabled us to test our system in a domain similar to the one described in the paper, but with more actions, more objects, and more complex plans. While the results of our initial experiments look promising, we are also in the process of transferring some of our ideas to a humanoid robot that can operate in a real-world kitchen with real-world objects and appliances. This will provide us with a challenging environment to test the scalability of our system and, in particular, our approach to planning and plan execution.

Acknowledgements

This work was partly funded by the European Commission through the PACO-PLUS project (FP6-2004-IST-4-27657).

References

Aarno, D.; Sommerfeld, J.; Kragic, D.; Pugeault, N.; Kalkan, S.; Wörgötter, F.; Kraft, D.; and Krüger, N. 2007. Early reactive grasping with second order 3D feature relations. In *The IEEE International Conference on Advanced Robotics*.

Başeski, E.; Kraft, D.; and Krüger, N. 2009. A hierarchical 3d circle detection algorithm applied in a grasping scenario. In *Proc. of VISAPP-09*, 496–502.

Detry, R.; Pugeault, N.; and Piater, J. 2009. A probabilistic framework for 3D visual object representation. *IEEE Transactions on Pattern Analysis and Machine Intelligence*. To appear.

Fikes, R. E., and Nilsson, N. J. 1971. STRIPS: A new approach to the application of theorem proving to problem solving. *Artificial Intelligence* 2:189–208.

Geib, C.; Mourão, K.; Petrick, R.; Pugeault, N.; Steedman, M.; Krueger, N.; and Wörgötter, F. 2006. Object action complexes as an interface for planning and robot control. In *IEEE-RAS Humanoids-06 Workshop: Towards Cognitive Humanoid Robots*.

Kraft, D.; Pugeault, N.; Başeski, E.; Popović, M.; Kragic, D.; Kalkan, S.; Wörgötter, F.; and Krüger, N. 2008. Birth of the Object: Detection of Objectness and Extraction of Object Shape through Object Action Complexes. *Special Issue on "Cognitive Humanoid Robots" of the International Journal of Humanoid Robotics* 5:247–265.

Krüger, N.; Piater, J.; Wörgötter, F.; Geib, C.; Petrick, R.; Steedman, M.; Ude, A.; Asfour, T.; Kraft, D.; Omrčen, D.; Hommel, B.; Agostini, A.; Kragic, D.; Eklundh, J.-O.; Krüger, V.; Torras, C.; and Dillmann, R. 2009. A formal definition of object-action complexes and examples at different levels of the processing hierarchy. PACO-PLUS Technical Report, available from `http://www.paco-plus.org/`.

Krüger, N.; Lappe, M.; and Wörgötter, F. 2004. Biologically Motivated Multi-modal Processing of Visual Primitives. *The Interdisciplinary Journal of Artificial Intelligence and the Simulation of Behaviour* 1(5):417–428.

McDermott, D. 1998. PDDL – The Planning Domain Definition Language. Technical Report CVC TR-98-003/DCS TR-1165, Yale Center for Computational Vision and Control.

Mourão, K.; Petrick, R. P. A.; and Steedman, M. 2008. Using kernel perceptrons to learn action effects for planning. In *Proc. of CogSys 2008*, 45–50.

Murray, R.; Li, Z.; and Sastry, S. 1994. *A mathematical introduction to Robotic Manipulation*. CRC Press.

Pednault, E. P. D. 1989. ADL: Exploring the middle ground between STRIPS and the situation calculus. In *Proc. of KR-89*, 324–332. Morgan Kaufmann.

Petrick, R. P. A., and Bacchus, F. 2002. A knowledge-based approach to planning with incomplete information and sensing. In *Proc. of AIPS-2002*, 212–221.

Petrick, R. P. A., and Bacchus, F. 2004. Extending the knowledge-based approach to planning with incomplete information and sensing. In *Proc. of ICAPS-04*, 2–11.

Pugeault, N. 2008. *Early Cognitive Vision: Feedback Mechanisms for the Disambiguation of Early Visual Representation*. Ph.D. Dissertation, Informatics Institute, University of Göttingen.

Yoon, S.; Fern, A.; and Givan, R. 2007. FF-Replan: A baseline for probabilistic planning. In *Proc. of ICAPS-07*, 352–359.

Appendix F

P^2: A Baseline Approach to Planning with Control Structures and Programs

Ronald P. A. Petrick
School of Informatics
University of Edinburgh
Edinburgh EH8 9AB, Scotland, UK
rpetrick@inf.ed.ac.uk

Abstract

Many planners model planning domains with "primitive actions," where action preconditions are represented by sets of simple tests about the state of domain fluents, and action effects are described as updates to these fluents. Queries and updates are typically combined in only very limited ways, for instance using logical operators and quantification. By comparison, formalisms like Golog permit "complex actions," with control structures like `if-else` blocks and `while` loops, and view actions as programs. In this paper we explore the idea of planning directly with complex actions and programs. We describe the structure of a simple planner based on undirected search, that generates plans by simulating the execution of action programs before they are added to a plan. An initial evaluation compares this approach against a classical heuristic planner using a domain whose program structures have been compiled into ordinary PDDL actions. Initial results illustrate that in certain domains, planning directly with programs can lead to a significant performance improvement. This work offers a baseline planner to compare against alternate approaches to planning with programs.

Introduction and Motivation

A recent trend in modern planning research has focused on the problem of planning with complex expressions, control structures, and programs—representations that are more complicated compared with traditional formalisms based on PDDL (McDermott 1998), the standard language for modelling planning domains. While recent additions to PDDL (e.g., constraints, preferences, durative actions, and numerical fluents) have extended its expressiveness, PDDL remains inherently STRIPS-like (Fikes and Nilsson 1971) in its structure. *Primitive actions* form the basis of a domain specification: action preconditions are defined by simple tests about the state of domain fluents, and action effects capture the (conditional) changes made to these fluents. Fluent tests and updates are often combined in very limited ways, using standard logical connectives and quantification.

By comparison, attempts to plan with *complex actions* admit actions with control flow blocks (e.g., sequence, iteration, and conditionals) and other procedural operators inspired by imperative programming languages. In practice, complex actions operate more like *programs* and are often distinct from primitive actions, with the latter defining the fluent-level state changes and the former acting as a wrapper around sets of primitive actions. While complex actions add more flexibility to the expressiveness of the representation language, most planners cannot directly construct plans with such actions. In this paper we present a simple planner that is capable of manipulating such structures.

The idea of mixing procedural constructs with planning is not new. For instance, much work has addressed the problem of automatically constructing *macro operators*, which combine useful sequences of actions in an attempt to improve plan generation efficiency (e.g., (Botea, Müller, and Schaeffer 2007; Coles and Smith 2007)). *HTN planning* (e.g., (Sacerdoti 1975; Nau et al. 2003)) also has a procedural flavour: HTN domains abstract the action space into high-level tasks and methods for decomposing those tasks into more primitive subtasks, with the lowest-level subtasks corresponding to ordinary planning operators. More formally, Levesque (1996) generalizes the planning problem in terms of a universal programming language $\mathcal{R}$, which includes sequence, branch, and loop constructs operating over actions. Levesque (2005) also uses a variant of $\mathcal{R}$ to investigate the problem of automatically generating plans with loops.

More closely related to the focus of this paper, one of the most popular approaches to planning with programs has been to *compile* complex actions into primitive actions, written in ordinary PDDL, which can then be used in conjunction with ordinary off-the-shelf planners. For instance, McIlraith and Fadel (2002) formalize an approach that transforms certain classes of programs written in Golog (Levesque et al. 1997)—a high-level programming language based on the situation calculus (McCarthy and Hayes 1969; Reiter 2001)—into PDDL. These programs allow procedural structures like action sequencing, `if-else` blocks, and a bounded `while` loop, among others. Baier and McIlraith (2006) build on this work by considering Golog programs with sensing actions (i.e., knowledge-producing actions that observe the state of the world without necessarily changing it) and translate these domains into a form usable by planners that support sensing actions, but not complex actions. Similarly, Baier, Fritz, and McIlraith (2007) compile procedural domain control knowledge into PDDL domains, modelled in a language based on Golog.

There are two potential drawbacks of the compilation approaches. First, new fluents and actions are generally intro-

duced into the resulting planning domain as a consequence of the compilation process, thereby increasing the size of the state space. Second, the rich control knowledge explicitly represented in structures like loops is discarded during compilation. Instead, the behaviour of such structures must be "rediscovered" through search, by appropriately guiding the planner's search through the resulting primitive actions, to mimic the effects of the original complex actions. While modern planners can often cope with the first drawback, the second is more problematic. For instance, the number of states a planner must visit can quickly become large when loops are permitted. As we will see, even the best heuristic planners do not always work well with compiled domains.

As an alternative to the compilation approaches, we explore the notion of planning directly with complex actions and programs, by simulating their execution within action blocks. We describe the implementation of a simple planner that supports a set of procedural constructs, including `if-else` blocks and unbounded `while` loops. During plan construction our planner simulates the application of an action by "running" its precondition or effects program, in a manner not unlike Golog. While we do not aim to be competitive with off-the-shelf planners in terms of speed (e.g., our initial implementation uses blind search), our planner nevertheless shows good performance compared against Metric-FF (Hoffmann 2003) on a toy domain, and provides a useful baseline to compare against alternative approaches. Overall, this work is a first step in a research agenda aimed at designing new planners that can search and plan directly with procedural control structures.

Example: Compiling `while` Loops into PDDL

As motivation for this work, consider the toy action in Figure 1. This action is similar in form to a primitive action, but includes a `while` loop. The intent here is to "loop while `i` is less than or equal to the value of the function `size(?d)`," adding `i` to the value of `count` and 1 to `i` each time through the loop. Although PDDL does not directly support actions with `while` loops, we can transform this action into a valid PDDL form that achieves a similar effect.[1]

Figure 2 shows three PDDL actions that encode the behaviour of the action in Figure 1: `processDataset` models the effects of the original action up to the start of the `while` loop, `processDataset-inLoop` simulates one iteration through the loop, and `processDataset-endLoop` encodes the effects following the loop. The first action contains the preconditions of the original action. The new predicate `context-loop` acts as a guard, controlling access to the body of the compiled loop. A second new predicate, `context-loop-params`, tracks the parameters of the original action. (If the domain contained additional actions their preconditions would also be updated with references to `context-loop` to prohibit their application during the execution of this loop.)

[1]We have implemented a compiler for transforming actions with simple program structures into PDDL, in order to compare our approach against such compilation methods. The example in Figure 2 was generated by our compiler and is characteristic of the kinds of actions we can produce. In general, we use the ADL subset of PDDL but our example here also requires numerical fluents.

```
action processDataset(?d)
   precondition:
      dataset(?d) and
      not(processedDataset(?d))
   effect:
      i = 1 ;
      while (i <= size(?d))
         count = count + i ;
         i = i + 1
      endWhile ;
      processedDataset(?d)
endAction
```

Figure 1: A simple action with a `while` loop

In this case, the correct behaviour of the compiled actions results from the planner's ability to order these actions appropriately during its search. For instance, once `processDataset` has been applied, the only action subsequently permitted according to its preconditions is `processDataset-inLoop`, which can be continually applied until the loop conditions are false. At this point the only permissible action is `processDataset-endLoop`, which completes the execution of the original action.

Although this example is extremely simple, we note two potential drawbacks. First, two actions and two predicates are added to the domain, increasing the size of the state space. Second, and more worrying, is the prospect that each iteration of the `while` loop is now an action instance. Thus, a loop with 100 iterations requires a sequence of 102 actions, and the rich control knowledge explicitly represented by the original `while` loop must be implicitly rediscovered by the backend PDDL planner during preprocessing and search.

Representing Actions as Programs

As an alternative to the compilation approach, we describe the structure of a simple planner called *ProgPlan* (abbreviated P^2), which supports actions with program constructs, and simulates their execution during plan search.

Symbols We assume a planning scenario whose symbols are defined as in an ordinary PDDL planning problem. Thus, we include a set of *fluent* symbols representing the properties of the domain that can change as a result of action, including both predicates and functions. (We also allow equality and standard numerical relations like $<$.) A set of *constants* denoting the objects in the domain is also defined.

The representation language used by P^2 is built around the notion of an *expression* and a *program*.

Expressions An *expression* in our representation is similar to the form of the preconditions used by ordinary classical, deterministic planners (e.g., the preconditions in Figure 2). Expressions can use the connectives `and`, `or`, `not`, `exists`, and `forall`, plus arithmetic expressions and fluent tests about the value of relations and functions.

We define a complex *expression* as follows:

```
(:action processDataset
  :parameters (?d)
  :precondition
    (and (not (context-loop))
         (dataset ?d)
         (not (processedDataset ?d)))
  :effect
    (and (assign (i) 1)
         (context-loop)
         (context-loop-params ?d)))

(:action processDataset-inLoop
  :parameters (?d)
  :precondition
    (and (context-loop)
         (context-loop-params ?d)
         (<= (i) (size ?d)))
  :effect
    (and (increase (count) (i))
         (increase (i) 1)))

(:action processDataset-endLoop
  :parameters (?d)
  :precondition
    (and (context-loop)
         (context-loop-params ?d)
         (not (<= (i) (size ?d))))
  :effect
    (and (processedDataset ?d)
         (not (context-loop))
         (not (context-loop-params ?d))))
```

Figure 2: Compiled PDDL actions simulating a `while` loop

expression ::= *expression* `and` *expression* |
expression `or` *expression* |
`not` (*expression*) | (*expression*) |
`forall` (*parameters*) *expression* |
`exists` (*parameters*) *expression* |
arithmetic-expression |
fluent-test.

We note that expressions "ground out" with ordinary arithmetic expressions (which include a large set of expressions from the C programming language) and fluent queries.

Programs A *program* is a set of control structures which operate over fluent updates and expressions.

program ::= *program* **;** *program* |
`if` *expression* `then`
program `else` *program* `endIF` |
`while` *expression* `do`
program `endWhile` |
`forall`(*parameters*)
program `endForall` |
`exists`(*parameters*) *expression* `then`
program `else`
program `endExists` |
arithmetic-assignment |
fluent-update | *nil*.

We follow ordinary program syntax in using **;** as the standard sequence operator for chaining program statements together. The `if-else` block is a standard conditional test which allows a choice as to which program should be executed, depending on the outcome of the test (the first program on success, the second on failure). Similarly, `while` is a standard while loop that repeats the execution of a program as long as the test expression is true. The `forall` and `exists` control structures introduce a special type of "quantified" program statement. `forall` is a loop that repeatedly executes a program; each time through the loop a new binding from the set of domain constants is chosen and assigned to the specified parameters. `exists` is a conditional nondeterministic choice statement that attempts to find a binding for the specified parameters so that the test expression evaluates as true. If found, the first program block is executed; otherwise, the second program block is executed. In both types of quantified structures, the "bound" parameters may be used in the body of the control block. Finally, a program can also be an empty program *nil*, an ordinary fluent update, or an arithmetic assignment statement. For arithmetic assignments, we not only allow simple calculations whose results are assigned to functions but also a rich selection of C-style numerical expressions.

Actions Actions are structured in a similar way to ordinary actions, with names, parameters, preconditions, and effects. Parameters are ordinary action variables which are bound to produce action instances. (Such variables may occur in an action's preconditions or effects.) In our case, preconditions are defined to be expressions and effects are programs, i.e.,

`action` *A* (*parameters*)
`preconditions:` *expression*
`effects:` *program*
`endAction`

Action preconditions and effects have the same intuitive meaning as ordinary planning actions: during plan construction an action's preconditions must be true before it's effects can be applied. In particular, we do not distinguish between "primitive" and "complex" actions in our representation.[2]

For instance, Table 3 shows a set of actions taken from an e-mail application domain, which give a flavour of the types of actions we can model with our representation. The *read*(m) action marks a particular message m as "read", provided it is in the user's inbox. In this case there are two effects: a fluent update marking m as read, and a second update increasing the count of the function *numread* which tracks the number of messages marked as read. The *markAllRead* action has the effect of marking all known messages in the user's inbox as read. In this case, the effects are modelled with an outer `forall` block and an inner `if-then` block, which tests each message and ensures only those messages in the user's inbox are appropriately marked. The functions *numread* and *numunread* denote the number of read and unread messages, respectively. The *findUnread* action uses the `exists` structure to find a message in the user's inbox which has not been read and sets the function *current* as this message. In the case no such message exists, *current* is set to a special constant *none*. Finally, the

[2]We are currently adding a "procedure call" to our representation, allowing one action to execute another action. This construct will let us specify actions with more complex control flow.

```
action markRead(?m)
   precondition: inbox(?m)
   effect:
      read(?m) ;
      numread = numread + 1
endAction

action markAllRead
   precondition: true
   effect:
      numread = 0 ;
      numunread = 0 ;
      forall(?m)
         if inbox(?m) then
            read(?m) ;
            numread = numread + 1
         endIf
      endForall
endAction

action findUnread
   precondition: true
   effect:
      exists(?m)
         (inbox(?m) and not(read(?m)))
         then current = ?m
         else current = none
      endExists
endAction

action countRange
   precondition: from <= to
   effect:
      count = 0 ; skipped = 0 ;
      i = from ;
      while i <= to do
         if read(msg(i)) then
            count = count + 1
         else
            skipped = skipped + 1
         endIf ;
         i = i + 1
      endWhile
endAction
```

Figure 3: Actions from an e-mail application domain

countRange action is used to count the number of messages in a particular range that are marked as read. The function *msg*(i) maps a message number i to a particular message. The `while` loop ensures we only consider the range defined by the functions *from* and *to*. The function *count* tracks the number of messages that are counted in the range, while *skipped* denotes the number of messages in the range that we ignore. The expression *read*(*msg*(i)) illustrates a permissible fluent test, with a nested function as an argument.

Planning by Simulating Program Execution

We now turn our attention to evaluating expressions and programs with respect to our representation.

Expression evaluation A *state* is a snapshot of the values of all fluents defined in a domain. For expressions, we define a procedure *EvalExpr*(e, S) which evaluates whether a compound expression e is true at a state S by recursively unwinding the expression down to its component parts (i.e., fluent tests), which are then evaluated at S. A special function *EvalArithExpr*(e, S) evaluates arithmetic expressions by reducing all arithmetic expressions (which may contain functions) to a number. Following C programming style, an arithmetic expression is "true" if it evaluates to a non-zero value. We have the following evaluation function.

Definition 1 Let S be a state let e, e_1, and e_2 be expressions. *EvalExpr*$(e, S) =$ **true** if

1. e has the form "e_1 `and` e_2" and *EvalExpr*$(e_1, S) =$ **true** and *EvalExpr*$(e_2, S) =$ **true**,
2. e has the form "e_1 `or` e_2" and *EvalExpr*$(e_1, S) =$ **true** or *EvalExpr*$(e_2, S) =$ **true**,
3. e has the form "`not(` e_1)" and *EvalExpr*$(e_1, S) =$ **false**,
4. e has the form "(e_1)" and *EvalExpr*$(e_1, S) =$ **true**,
5. e has the form "`forall(`$\vec{x}$`)` e_1" and *EvalExpr*$(e_1(\vec{x}/\vec{c}), S) =$ **true** for every substitution $\vec{c}$ of $\vec{x}$ in e_1,
6. e has the form "`exists(`$\vec{x}$`)` e_1" and *EvalExpr*$(e_1(\vec{x}/\vec{c}), S) =$ **true** for some substitution $\vec{c}$ of $\vec{x}$ in e_1,
7. e is an arithmetic expression and *EvalArithExpr*$(e, S) \neq 0$,
8. e is a fluent query and $\mathbf{IA}(e, S) =$ **true**.

Otherwise, *EvalExpr*$(e, S) =$ **false**.

EvalExpr recursively deconstructs a complex expression into simpler components. In (1) – (4), the standard `and`, `or`, and `not` connectives, plus expression precedence, are evaluated in a straightforward way. In (5) and (6), *EvalExpr* considers possible substitutions of the quantified parameters. The notation $e_1(\vec{x}/\vec{c})$ indicates that all occurrences of $\vec{x}$ in e_1 should be syntactically replaced with $\vec{c}$, where $\vec{c}$ is taken from the set of defined constants. (I.e., the expression is rewritten before it is recursively evaluated.) In (7), the special function *EvalArithExpr* evaluates an arithmetic expression against a state S, by attempting to reduce the expression to a number. (Space prohibits us from describing this process in detail.) We follow C programming style here and consider an arithmetic expression to be "true" at S if it evaluates to a non-zero value. In (8), the truth of a fluent query e is determined by a function called **IA** which checks the fluent's value in state S. **IA** is also responsible for evaluating queries with references to nested functional fluents.

Program simulation In traditional planning, a set of ordinary fluent updates, when applied to a state S, transforms S to produce a new state S'. We extend this notion to programs by *simulating* the run of a given program at a state S. All fluent updates that arise during program execution are applied to the current state, generating a sequence of new states. (Each fluent update could produce a new state.) Upon program termination, we disregard any "intermediate" states and return the final resulting state S'.

A procedure called *RunProg*(p, S) simulates the execution of a program p starting in a state S, and returns a state S' on completion of the program run. In general,

proc *ProgPlan*$(S, G, \mathcal{A}, P)$
  if *EvalExpr*$(G, S) =$ **true then return** P
  else if
    choose$(a \in \mathcal{A})$: *EvalExpr*$(pre(a), S) =$ **true then**
    $S' =$ *RunProg*$(eff(a), S)$ **;**
    return *ProgPlan*$(S', G, \mathcal{A}, P + a)$
  else return *fail*
  endIf
endProc

Figure 4: Pseudocode for the P^2 planning algorithm

RunProg operates as a program interpreter, stepping its way through a given program. A *program counter* tracks the current program statement being executed, which is updated after its completion. Depending on the type of procedural construct under evaluation, the interpreter runs a small control program to evaluate its outcome. For instance, evaluating a sequence construct involves running two programs in turn, with the second program executing from the state resulting from the execution of the first program, i.e., *RunProg*$(p_1$ `and` $p_2, S) :=$ *RunProg*$(p_2,$ *RunProg*$(p_1, S))$. For a `while` loop, the interpreter runs the control program

RunProg(`while` e `do` p `endWhile`, S) :=
  while *EvalExpr*$(e, S) =$ **true** do
    $S =$ *RunProg*(p, S)
  endWhile ; return S.

Here, *EvalExpr* evaluates the truth of expression e in each iteration of the loop. (The underlined control structures are part of the interpreter's control program for simulating the `while` loop.) S is updated each time through the loop and the final S is returned on completion. One important danger of this approach is that programs aren't guaranteed to terminate: since we simulate actual programs, we also inherit the problems of ordinary program design, including the possibility of infinite loops. Similar control programs are defined for the other control structures in our representation language. When *RunProg* encounters a fluent update, it applies it to the existing state an an ordinary update.

Planning A planning problem is specified by a set of actions $\mathcal{A}$, an initial state S, and a set of goal conditions G. The initial state can be any state (as in ordinary PDDL) and a goal is any expression. Figure 4 shows the pseudocode describing the main operation of our program planner, P^2. Plans are built in a simple forward-chaining manner, starting from the initial state. The planning algorithm attempts to grow a plan by searching over the space of applicable actions and choosing a ground action instance a whose preconditions *pre*(a) (an expression) are satisfied in the current state S according to *EvalExpr*. If such an action exists, its effects *eff*(a) (a program) are applied to S by *RunProg* to produce a new state S'. Action a is concatenated to the end of the current plan and planning continues until a state is reached where the goal is satisfied, or the plan cannot be extended.

Initial Evaluation

We have implemented an initial version of our planner in C++ as a simple forward-chaining planner using undirected depth-first search and breadth-first search.[3] Our expression evaluator implements the expressions described above and a large subset of the numerical expressions available in C.

`size(d1)`	Metric-FF	P^2
100	0.01	0.01
1000	0.33	0.01
2500	2.03	0.01
5000	8.07	0.01
10000	32.41	0.01
25000	202.62	0.02
50000	>3000.00	0.05

Table 1: Running time in seconds on the example domain

n	Test-1	Test-2	Test-3	Test-4
100	0.01	0.02	0.02	0.03
1000	0.07	0.11	0.14	0.21
10000	0.71	0.90	1.20	1.94
100000	7.02	8.82	13.08	19.30

Table 2: Running time in seconds of benchmark tests on `while` loop programs of n iterations and length 100 plans

Although our planner has not been optimized in any substantial way, we have applied it to a series of experiments in some small planning domains. In the first set of experiments, we compare P^2 using the action in Figure 1 against Metric-FF (Hoffmann 2003) using the compiled PDDL actions in Figure 2. In each case we consider a problem with the goal of processing a single dataset `d1` of varying size `size(d1)`. The results of this experiment are shown in Table 1. (All tests were performed on a Linux system with a single CPU running at 1.86 GHz and 2Gb of RAM.) Our prototype planner performs significantly better than Metric-FF. This is not surprising since Metric-FF must build a plan of length $n + 2$ using the compiled domain, for each `while` loop with n iterations. (It is also not altogether bad, and a tribute to modern search heuristics, that Metric-FF can build a plan with 2500 steps in 2 seconds.) By comparison, simulating the execution of the `while` loop means that P^2 solves each problem instance with a plan of length 1.

We also ran a number of benchmark experiments designed to test the efficiency of the program simulator running at the core of our planner. In these tests, we construct a planning domain with a single action that does not have any preconditions. This action's effects consist of a `while` loop of n iterations, forming the outermost control block. We then vary the contents of the `while` loop in each test case to evaluate the performance of different program structures. In Test-1, a single fluent update is added within the `while` loop. In Test-2, an `if-then` statement is added which conditionally performs a fluent update. In Test-3, a `forall` statement is added which ranges over a domain of 50 objects, performing a fluent update each iteration through the loop. Finally, in Test-4, a `forall` statement ranges over

[3]The source code for P^2 is available from `http://homepages.inf.ed.ac.uk/rpetrick/research/p2`.

100 objects. Each task has the common goal of chaining 100 actions together into a plan. The results of the four tests are shown in Table 2. Our initial experiments are encouraging, at least as far as program simulation is concerned. For instance, the $n = 10000$ case in Test-1 means that the program simulator is running 1 million loop iterations and fluent updates in under 1 second. However, these experiments are also quite simple and more work is needed to improve the planner's search procedure: blind search is only effective in small domains and there are many instances where off-the-shelf heuristic planners using compiled program actions will outperform our current implementation.

Discussion

Our approach differs from the complex-to-primitive action compilation methods since we're primarily interested in working with program structures directly at the planning level. However, for some types of control structures we can also make use of the compiled form, especially when it is well understood how to plan with such structures. (For instance, `if-else` blocks are a special case of ADL-style context-dependent effects (Pednault 1989).) For more complex structures, such as loops, we want to develop techniques for searching the state spaces arising from such structures, and use the rich procedural control information these structures provide. As a first step, we are interested in adapting the state relaxation technique used by FF during its preprocessing phase, as a distance estimate from a state to the goal, for instance by simulating program execution while ignoring delete lists. We are initially focusing on subsets of our representation for which this technique can be easily applied, to assess its effect on performance. In general, more study is needed since complications can make this method more difficult to apply (e.g., the continuation/exit conditions of a `while` loop might depend on the deletion of a fluent from the current state; failure to do so could result in poor reachability estimates or non-terminating loops).

While our approach to simulating program execution is similar to that of Golog, we differ from those approaches aimed at integrating Golog with off-the-shelf planners. For instance, Röger, Helmert, and Nebel (2008) compare the expressiveness of Golog and ADL (Pednault 1989), and identify a maximal subset of the situation calculus that can be equivalently expressed in ADL. Claßen et al. (2007) separate certain procedural parts of Golog from the classical planning task, by using FF as a blackbox planner which is invoked when certain "achieve" statements are encountered in a Golog program. In contrast, we take a more tightly coupled view and treat program constructs as part of the planning problem. (In this way we are much closer to (McIlraith and Fadel 2002) than (Claßen et al. 2007).) However, one of Golog's strengths is its clean semantics, built on the situation calculus—an approach we are sympathetic with. (For instance, our informal procedural semantics could be redefined more formally in terms of Golog programs.) In future work we plan to evaluate our approach against (Claßen et al. 2007), as well as related approaches like (Baier and McIlraith 2006), which uses sensing actions.

Our current planner is not meant to be competitive with current off-the-shelf planners. Instead, it is a first step in an ongoing research programme aimed at developing practical planners that can operate in more complex state spaces. As such, we offer our present planner to the community as a baseline tool for evaluating alternative approaches and advancing research into planning with programs.

Acknowledgements

This work was partly funded by the European Commission through the PACO-PLUS project (FP6-2004-IST-4-27657).

References

Baier, J. A., and McIlraith, S. A. 2006. On planning with programs that sense. In *Proc. of KR-06*, 492–502.

Baier, J. A.; Fritz, C.; and McIlraith, S. A. 2007. Exploiting procedural domain control knowledge in state-of-the-art planners. In *Proc. of ICAPS-07*, 26–33.

Botea, A.; Müller, M.; and Schaeffer, J. 2007. Fast planning with iterative macros. In *Proc. of IJCAI-07*, 1828–1833.

Claßen, J.; Eyerich, P.; Lakemeyer, G.; and Nebel, B. 2007. Towards an integration of golog and planning. In *Proc. of IJCAI-07*, 1846–1851.

Coles, A., and Smith, A. 2007. Marvin: A heuristic search planner with online macro-action learning. *J. Artificial Intelligence Research* 28:119–156.

Fikes, R. E., and Nilsson, N. J. 1971. STRIPS: A new approach to the application of theorem proving to problem solving. *Artificial Intelligence* 2:189–208.

Hoffmann, J. 2003. The Metric-FF planning system: Translating "ignoring delete lists" to numeric state variables. *J. Artificial Intelligence Research* 20:291–341.

Levesque, H. J.; Reiter, R.; Lespérance, Y.; Lin, F.; and Scherl, R. B. 1997. GOLOG: A logic programming language for dynamic domains. *Journal of Logic Programming* 31(1–3):59–83.

Levesque, H. J. 1996. What is planning in the presence of sensing? In *Proc. of AAAI-96*, 1139–1146.

Levesque, H. J. 2005. Planning with loops. In *Proc. of IJCAI-05*, 509–515.

McCarthy, J., and Hayes, P. J. 1969. Some philosophical problems from the standpoint of artificial intelligence. *Machine Intelligence* 4:463–502.

McDermott, D. 1998. PDDL – The Planning Domain Definition Language (Version 1.2). Technical Report CVC TR-98-003/DCS TR-1165, Yale Center for Computational Vision and Control.

McIlraith, S., and Fadel, R. 2002. Planning with complex actions. In *Proc. of NMR-02*, 356–364.

Nau, D.; Au, T.-C.; Ilghami, O.; Kuter, U.; Murdock, J. W.; Wu, D.; and Yaman, F. 2003. SHOP2: An HTN planning system. *J. Artificial Intelligence Research* 20:379–404.

Pednault, E. P. D. 1989. ADL: Exploring the middle ground between STRIPS and the situation calculus. In *Proc. of KR-89*, 324–332.

Reiter, R. 2001. *Knowledge In Action: Logical Foundations for Specifying and Implementing Dynamical Systems*. MIT Press.

Röger, G.; Helmert, M.; and Nebel, B. 2008. On the relative expressiveness of ADL and Golog: The last piece of the puzzle. In *Proc. of KR-08*, 544–550.

Sacerdoti, E. D. 1975. The nonlinear nature of plans. In *Proc. of IJCAI-75*, 206–214.

Appendix G

paco plus — Connecting knowledge-level planning and task execution on a humanoid robot using Object-Action Complexes

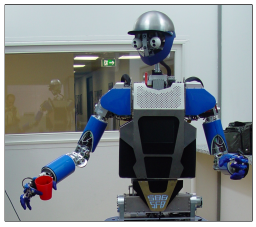
Ronald Petrick, Nils Adermann†, Tamim Asfour†, Mark Steedman*, and Rüdiger Dillmann†*
**University of Edinburgh, United Kingdom and †Karlsruhe Institute of Technology, Germany*

Motivation

A humanoid robot operating in a real-world domain typically requires a collection of decision making and control mechanisms, combining low-level sensorimotor systems with high-level action/reasoning engines. Building such systems requires overcoming the theoretical and practical challenges that arise from integrating such diverse components in a single framework.

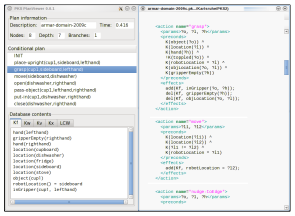
ARMAR humanoid robot platform

Our system uses the ARMAR humanoid robot platform [1] featuring a 7-degree-of-freedom (DOF) head with foveated vision, a 3-DOF torso, two 7-DOF arms, and two 5-finger hands, each with tactile sensors and 8 DOFs. ARMAR also includes a number of sensorimotor processes that enable it to act autonomously in complex environments.

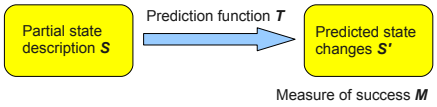


Planning with Knowledge and Sensing (PKS)

High-level plans are built using PKS [3], a conditional planner that operates with incomplete information and sensing actions. PKS operates at the "knowledge level" by explicitly modelling what the planner knows, and does not know, during plan generation.

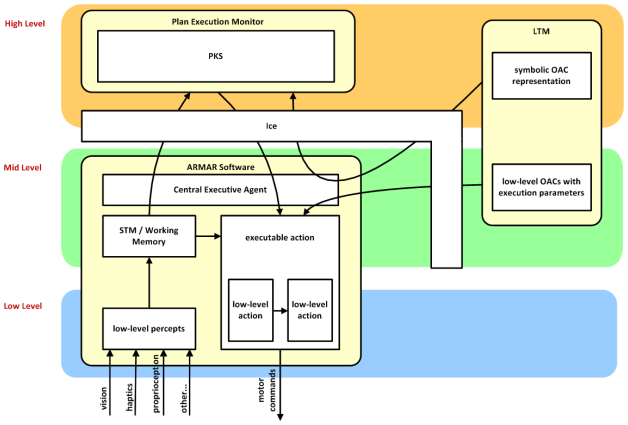


Object-Action Complexes (OACs)



Task planning and execution are connected using Object-Action Complexes (OACs) [2], a universal representation usable at all levels of a cognitive architecture. OACs combine ideas from STRIPS, the object/situation-oriented concept of affordance, and logical formalisms like the event calculus. Planning-level operators and robot-level tasks/skills are modelled using OACs.

System architecture and component interaction



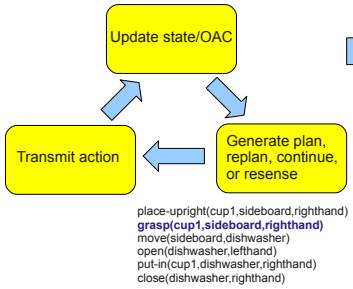
The Ice (Internet Communications Engine) middleware facilitates the exchange of information between system levels/components.

Using Object-Action Complexes for task planning and execution

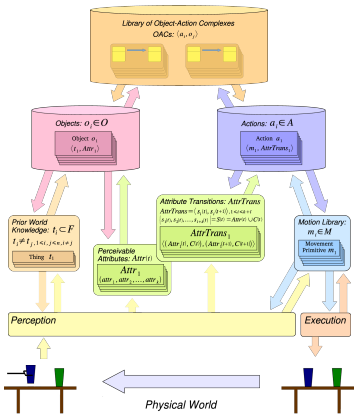
High level

Preconditions	Effects
object(?o) location(?l) hand(?h) ¬toppled(?o) robotLocation=?l objLocation(?o,?l) gripperEmpty(?h)	ingripper(?o,?h) ¬gripperEmpty(?h) ¬objLocation(?o,?l)
Success	0.90

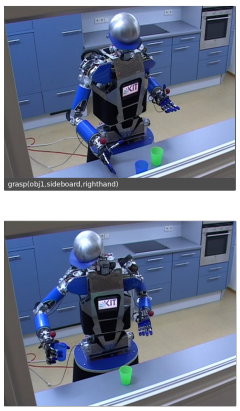
High-level OAC representation of **grasp(?o,?l,?h)**



Low level



Execution



References

[1] T. Asfour, K. Regenstein, P. Azad, J. Schröder, A. Bierbaum, N. Vahrenkamp, and R. Dillmann. ARMAR-III: An integrated humanoid platform for sensory-motor control, IEEE-RAS International Conference on Humanoid Robots (Humanoids 2006), pages 169-175, 2006. See **http://wwwiaim.ira.uka.de/** for more information about ARMAR.

[2] N. Krüger, J. Piater, C. Geib, R. Petrick, M. Steedman, F. Wörgötter, A. Ude, T. Asfour, D. Kraft, D. Omrcen, A. Agostini, R. Dillmann. Object-Action Complexes: Grounded abstractions of sensorimotor processes, submitted to *Robotics and Autonomous Systems*, 2009. See **http://www.paco-plus.org/** for a technical report about OACs.

[3] R. Petrick and F. Bacchus. Extending the knowledge-based approach to planning with incomplete information and sensing, *Proceedings of the International Conference on Automated Planning and Scheduling (ICAPS-04)*, pages 2-11, 2004. See **http://homepages.inf.ed.ac.uk/rpetrick/** for more information about PKS.