Project no.: 027657

Project full title: Perception, Action & Cognition through learning of Object-Action Complexes

Project Acronym: PACO-PLUS

Deliverable no.: D4.3.5

Title of the deliverable: Technical Report: Revised version of PACO-PLUS design documentation for integration of robot control and AI planning

Contractual Date of Delivery to the CEC:	31 January 2009
Actual Date of Delivery to the CEC:	31 January 2009
Organisation name of lead contractor for this deliverable:	UEDIN
Author(s): Ronald Petrick, Christopher Geib, and Mark Steedman	
Participant(s): UEDIN, SDU, UniKarl	
Work package contributing to the deliverable:	WP4, WP5
Nature:	R
Version:	Final
Total number of pages:	47
Start date of project:	1st Feb. 2006 **Duration:** 48 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:

One of the central contributions of WP4.3 is to provide the high-level action representation and planning apparatus needed to support plan generation and execution in low-level robotics domains and higher-level environments requiring language and communication. This deliverable focuses on the integration of UEDIN's high-level planning components with the low-level robot platforms of UniKarl (previously reported in WP1) and SDU (previously reported in WP4.1). A high-level planning representation is developed for each integration environment, along with a common message-passing and control architecture to facilitate communication in the integrated system. Related integration tasks, such as plan execution monitoring, high-level action learning, and dialogue planning are also discussed, providing links to WP5. This work forms part of the project-wide integration work reported in WP1, and has connections to WP8.

Keyword list: Integration of low-level robot/vision with high-level planning, message passing communication protocol, three-level control architecture, Planning with Knowledge and Sensing (PKS)

Table of Contents

1. EXECUTIVE SUMMARY 4
2. PAPERS ASSOCIATED WITH D4.3.5 5
REFERENCES 6
A. INTEGRATING LOW-LEVEL ROBOT/VISION WITH HIGH-LEVEL PLANNING AND SENSING IN PACO-PLUS 9

1. Executive Summary

A central contribution of WP4.3 is to provide the high-level action representation and planning apparatus needed to support plan generation and execution in low-level robotics domains (WP1 and WP4.1) and higher-level domains requiring language and communication (WP5). The attached UEDIN technical report, *Integrating Low-Level Robot/Vision with High-Level Planning and Sensing in PACO-PLUS*, describes ongoing integration work in support of this goal.

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 UniKarl kitchen domain using the ARMAR robot platform [3, 2], and an object stacking problem using SDU's robot/vision system [14]. A high-level action representation is developed for each integration scenario for the purpose of goal-directed planning, by abstracting the capabilities of the robot and its working environment. Although different high-level action specifications are required due to differences in the low-level functionality of the UniKarl and SDU robots, the core representations are nevertheless similar.

High-level planning is provided by the PKS planner [18, 19], a state-of-the-art knowledge-level planner which UEDIN is extending for use in robotic and linguistic domains (WP4 and WP5). Unlike traditional planners, PKS constructs plans at the "knowledge level", by representing and reasoning about how the planner's (incomplete) knowledge state changes during plan generation. Actions are specified in a STRIPS-like [9] manner in terms of their preconditions and effects. PKS is able to construct conditional plans with sensing actions, and supports numerical reasoning, run-time variables [8], and features like functions that arise in real-world planning scenarios. As such, the work reported in this deliverable centres around the design of high-level action representations usable by PKS.

A common message-passing and control architecture is also presented to facilitate communication between the various levels of the integrated system. Furthermore, we demonstrate how PKS's ability to reason about incomplete information and sensing actions can be interpreted and executed by the lower system levels. Finally, we have reserved a role for possible mid-level processes (WP4.2) which could be incorporated into our architecture in the future.

This document also briefly describes how a number of tasks being pursued by UEDIN as part of WP4 and WP5 (e.g., plan execution monitoring, high-level action learning, and dialogue planning) relate to this integration work.

Overall, this deliverable reports a number of significant developments:

- In conjunction with UniKarl, UEDIN has developed a high-level action representation supporting some of the sophisticated capabilities of the ARMAR robot, including object manipulation with multiple grippers, movement between workspaces in the UniKarl kitchen, and relocation of objects awkwardly positioned for grasping. High-level plans can currently be built to advantage of such functionality in the UniKarl kitchen domain.
- The UEDIN message passing protocol and control architecture has been successfully transferred to the UniKarl robot platform, demonstrating the generality of our current approach. Work is ongoing to identify possible extensions to this architecture, to support additional features of the ARMAR system.
- As previously reported in WP4/WP5, the early integration of the SDU robot/vision system with the PKS planner has been completed, allowing simple linear plans and conditional plans with sensing actions to be generated by the planner and executed on the robot platform. More recently, work has progressed on the addition of a plan execution monitor into this architecture. This component will also be incorporated into the UniKarl system in the future.

- A role for a mid-level memory component has been identified within the control architecture, offering the prospect of improved high-level plan specification through mid-level plan refinement.
- This work provides a complete theoretical path from continuous low-level representations to high-level models suitable for planning and language (as required in WP5). The practical implementation of these systems currently supports plan generation using the full action representation, and plan execution using a restricted subset of the representation in both the SDU and UniKarl robot environments.

A number of tasks remain open at the time of this report and constitute ongoing and future work:

- The plan execution monitor currently being built by UEDIN has not yet been tested on the integrated UniKarl/UEDIN or SDU/UEDIN systems.
- 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 studying how best to utilise such information at the higher control levels. One technique we are experimenting with is the use of rapid replanning [23] (in conjunction with our plan execution monitor) which has been successfully applied by planners in the probabilistic track of the International Planning Competition [6].
- Previous work [15] reported in WP5 describes a mechanism for learning high-level STRIPS-style actions effects from world state snapshots of the form produced by our control architecture. In contrast to previous approaches (e.g., [22, 11, 20, 16, 7]) our approach uses kernel perceptrons [1, 10] combined with deictic referencing [16] to reduce the complexity of the learning task. (We believe this technique will also allow our approach to scale.) Using existing components described in this document we are now ready to complete the "learn-plan-execute" loop and integrate high-level action learning with PKS. The resulting system will let us build plans using learnt action models and execute generated plans in real robot environments. A preliminary description of this proposed work is given in [17].
- We have focused on robot-planner integration in this report, with an emphasis on standard action planning in PKS. Using an approach that applies modern planning techniques to problems in natural language generation (e.g., [12, 4, 13, 5]), we will generalize our existing apparatus for ordinary action planning to dialogue planning with speech acts. Although the theoretical work required to extend PKS to support dialogue planning (WP5) within our integration architecture is complete [21], the implementation of this approach is only partially complete.

Besides the connections to WP1, WP4, and WP5 mentioned above, this workpackage also has interactions with other workpackages including WP2, WP3, WP7, and WP8.

2. Papers Associated with D4.3.5

[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 2009.

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 UniKarl 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 is extending 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 of 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.

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] 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.

[3] 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.

[4] Luciana 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.

[5] Michael Brenner and Ivana 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.

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

[7] Mehmet R. Doğar, Maya Çakmak, Emre Uğur, and Erol Şahin. From primitive behaviors to goal directed behavior using affordances. In *Proc. of Intelligent Robots and Systems (IROS 2007)*, 2007.

[8] 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.

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

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

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

[12] 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.

[13] Alexander Koller and Ronald Petrick. Experiences with planning for natural language generation. In *Scheduling and Planning Applications woRKshop (SPARK 2008) at ICAPS 2008*, September 2008.

[14] 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.

[15] Kira Mourão, Ronald P. A. Petrick, and Mark Steedman. Using kernel perceptrons to learn action effects for planning. In *Proc. of CogSys 2008*, pages 45–50, 2008.

[16] Hanna M. Pasula, Luke S. Zettlemoyer, and Leslie P. Kaelbling. Learning symbolic models of stochastic domains. *Journal of Artificial Intelligence Research*, 29:309–352, 2007.

[17] Ronald Petrick, Dirk Kraft, Kira Mourão, Nicolas Pugeault, Norbert Krüger, and Mark Steedman. Representation and integration: Combining robot control, high-level planning, and action learning. In *Proc. of CogRob 2008 at ECAI 2008*, pages 32–41, 2008.

[18] 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.

[19] 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.

[20] Dafna Shahaf and Eyal Amir. Learning partially observable action schemas. In *Proceedings of the National Conference on Artificial Intelligence (AAAI-06)*. AAAI Press, 2006.

[21] 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.

[22] Xuemei 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.

[23] 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
2009-01-29

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 UniKarl 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 is extending 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 of 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

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

1 Introduction **4**

2 Object Manipulation in a Kitchen Domain (UniKarl/UEDIN Integration) **5**

2.1 High-level domain description 5

2.2 Representing actions for planning 8

2.3 Example plans 11

2.3.1 Example 1 11

2.3.2 Example 2 12

2.3.3 Example 3 12

3 Object Stacking with Sensing (SDU/UEDIN Integration) **13**

3.1 High-level domain description 14

3.2 Representing actions for planning 15

3.3 Example plans 18

3.3.1 Example 1 19

3.3.2 Example 2 19

3.3.3 Example 3 20

4 Experimental Extensions to the Integration Domains **21**

4.1 Pulling and relocating actions 21

4.2 Example plans 22

4.2.1 Example 1 23

4.2.2 Example 2 23

4.2.3 Example 3 24

5 Message Passing Protocol and Control Architecture **25**

5.1 Message definitions 25

5.2 Message passing control algorithms 25

5.2.1 Robot-level control loop 27

5.2.2 Memory-level control loop 27

5.2.3 Planning-level control loop 28

5.3 Socket communication library and sample code 28

5.4 Message passing example 30

6 Related High-Level Integration Work **32**
6.1 Plan execution monitoring 32
6.2 High-level action learning in robot domains 33
6.3 Towards language and communication with dialogue planning 34

7 Discussion **35**

List of Figures

1 Robot grasp types available to the planner 14
2 Flow of messages between the three system levels 25
3 Message passing control algorithms 29
4 Example of messages passed during the execution of `sense-open(obj2)` 32
5 A sample dialogue in the kitchen domain 35

List of Tables

1 High-level actions and properties in the kitchen domain 7
2 Representation of high-level actions in the kitchen domain 10
3 High-level actions and properties in the object stacking domain 16
4 Representation of high-level actions in the object stacking domain . . . 17
5 Additional high-level actions and properties 21
6 Representation of additional high-level actions 22
7 Message types defined in the message passing protocol 26
8 Send/receive message pairs 27

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 D1.1.2, 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 UniKarl 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 UniKarl/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 efforts have 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 UniKarl's ARMAR robot platform. Although differences between the UniKarl 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 incorporated into our architecture in the future.

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 UniKarl. 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 (UniKarl/UEDIN Integration)

In this section we describe the state of ongoing integration work to link UEDIN's high-level planning components with UniKarl's *ARMAR robot platform* [Asfour et al., 2006, 2008]. We primarily focus on the action representation used to support planning in the UniKarl 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 *UniKarl 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 UniKarl to properly define such actions.

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

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 $\neq$?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.

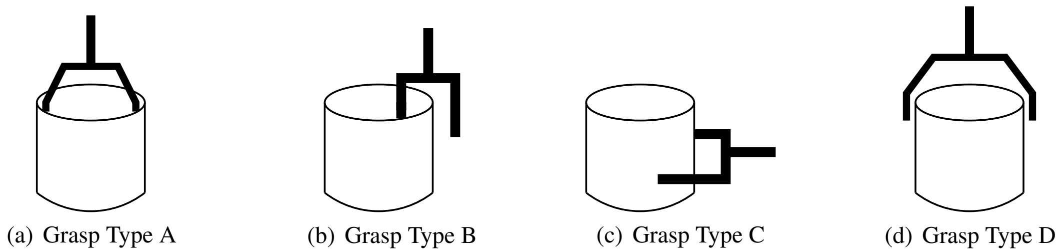
3 Object Stacking with Sensing (SDU/UEDIN Integration)

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 UniKarl 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 UniKarl 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 UniKarl 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 UniKarl scenario (e.g., the UniKarl 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 part of our current integration domains, but may be added to either domain at some point in the future. These extensions are still preliminary and are subject to change.

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

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

pullCloser-usingObject does not take into consideration how the "gripper extension" 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 UniKarl/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.)

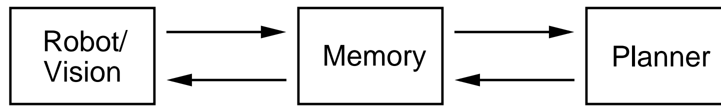


Figure 2: Flow of messages between the three system levels

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 described in this document currently use our message passing protocol and architecture which we believe is sufficiently general to support future domains we may define.

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. These lists may be expanded or streamlined in the future.

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 in the future 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 future bottom-up role

for the memory, 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, future changes to the communication protocol involving 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 initial 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. We are still in the process of extending the message passing protocol to include these new message types and, thus, we have not included a discussion of these messages here. Such additions will appear in a future version of this document.

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)```

```
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.

	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 is currently building a *plan execution monitor* that will be 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 will be 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 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 implementation of (ii), the plan execution monitor will initially provide 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 could then 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. We will initially focus on implementing (i) and (iii), with (ii) considered as future work.

The plan execution monitor will also have the added 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 will make 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 will be triggered as above. 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" operation like those shown in the example plans. From the point of view of the robot, it will only receive a sequential stream of actions. This will also be the case for the memory level, except when a complete plan is requested. In such situations a fully-specified conditional plan will be transmitted to the memory level.

Initially, we expect that most plans will fail early, and often, and that most monitoring operations will trigger replanning activities. Our goal is to implement the basic framework for the plan monitor in the near future, in order to evaluate its effectiveness on plans being executed in the actual robot environment.

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.

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. (We believe this technique will also allow our approach to scale.) 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.

Our current focus is on completing the "learn-plan-execute" loop to integrate the high-level action learner with the PKS planner, and to build plans using our learnt action models. Rather than testing with simulated data, we would like to use real state data generated from the SDU/UEDIN domain and execute plans in the actual robot environment, to investigate the quality and effectiveness of the learnt action models in real-world domains. A preliminary description of this proposed work is given in [Petrick et al., 2008] (and was previously included in D5.1.2).

6.3 Towards language and communication with dialogue planning

We have primarily focused on robot-planner integration in this document, with an emphasis on standard action 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. Our approach to achieving this goal is based on applying ordinary action planning techniques to *dialogue planning* with speech acts.

The problem of planning conversational moves can be viewed as a problem of planning with sensing or knowledge-producing actions (see, e.g., [Stone, 2000]). However, early approaches to dialogue planning (e.g., [Perrault and Allen, 1980, Appelt, 1985]) suffered as a result of inefficient planning techniques available at the time. Other approaches to this problem have tended to segregate standard action planning and discourse planning, using specialized techniques to address the latter (e.g., [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]). More recently, there has been a renewed interest in applying planning to problems in natural language generation, including dialogue (e.g., [Koller and Stone, 2007, Benotti, 2008, Koller and Petrick, 2008, Brenner and Kruijff-Korbayová, 2008]) to take advantage of modern planning techniques.

Our work also builds on the idea that the task of planning dialogue moves can be treated as an instance of planning with incomplete information and sensing. In prior work [Steedman and Petrick, 2007], we describe a set of extensions needed to adapt the Linear Dynamic Event Calculus (LDEC) [Steedman, 1997, 2002] to represent and reason about dialogue, using insights from the PKS planner and a representational unit called a *knowledge fluent* [Demolombe and Pozos Parra, 2000]. By incorporating ideas from PKS to the representation of dialogue acts in LDEC (our high-level symbolic representation of OACs), we can demonstrate how our existing formalisms and system components can be applied to the problem of planning mixed-initiative collaborative discourse.

We are currently in the process of implementing a series of extensions to PKS to enable dialogue planning with communicative acts. Our existing planning and plan execution mechanisms will remain fundamentally unchanged, meaning dialogue planning can be viewed as an instance of the same basic mechanisms used for standard action planning. Our extensions, however, will enable the planner to reason (in a limited sense) about the beliefs of multiple agents, model certain linguistic notions like common ground, and represent speech acts like "asking" and "telling", all of which are required for successful multi-agent discourse. For instance, Figure 5 shows an example of a possible two-agent dialogue in the kitchen domain described in Section 2, where speech acts are encoded as general STRIPS-style rules; we hope to generate dialogues similar to this with our

Agent	Dialogue	STRIPS-encoded speech act
Robot1:	Let's make breakfast.	[goal-propose(breakfast)]
Robot2:	I don't know how to make breakfast.	[assert(¬know(breakfast))
Robot1:	To make breakfast we must bring the cereal and the milk to the sideboard.	[explain(breakfast :- loc(cereal,sideboard) ∧ loc(milk,sideboard))]
Robot2:	Is the cereal at the sideboard?	[ask(loc(cereal,sideboard))]
Robot1:	No.	[tell(no)]
Robot2:	Where is the cereal?	[ask(loc(cereal, X))]
Robot1:	The cereal is in the cupboard.	[tell(loc(cereal,cupboard))]
Robot2:	Is the milk at the sideboard?	[ask(loc(milk,sideboard))]
Robot1:	No.	[tell(no)]
Robot2:	Where is the milk?	[ask(loc(milk, X))]
Robot1:	The milk is in the fridge.	[tell(loc(milk,fridge))]
Robot2:	Okay. I suggest I go to the cupboard, pickup the cereal, bring it to the sideboard, then go the fridge, pickup the milk, and bring it to the sideboard.	[assert-plan(move(sideboard,cupboard),...]

Figure 5: A sample dialogue in the kitchen domain

planner once our extensions are complete. More details about the dialogue planning component will be described in a future version of this document.

7 Discussion

In this document we described two high-level action representations enabling goal-directed planning in low-level robot domains. While additional actions may be added to these specifications in the future, based on the needs of particular robot platforms, we believe that the basic action representations will remain relatively unchanged. A number of important issues remain, however.

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 in the world. 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 a number of places where incomplete world state information can be introduced into the system. Some of these are endemic to the interaction of a resource bounded agent working in a real world setting. As a result, we must examine the limitations of the system's capabilities, as well as the traditional AI assumption that we have complete models of the state changes resulting from executed actions. This is an interesting area for future work and something we are committed to looking at in detail. Initially, however, we will simply ensure that our action models and state updates are complete and correct.

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 will more fully specify these new message types.

6. Although the focus of our integration efforts is shifting towards the UniKarl kitchen domain, which supports a more complex robot platform and real-world planning environment, we remain committed to ensuring our action descriptions, message passing protocol, and communication library continue to support the SDU robot/vision platform as needed for our ongoing integration tasks.

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 ongoing discussions with UniKarl. Special thanks go to Nils Adermann for his help with the ARMAR robot platform and UniKarl 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. 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.

R. Demolombe and M. P. Pozos Parra. A simple and tractable extension of situation calculus to epistemic logic. In *Proc. of ISMIS-2000*, pages 515–524, 2000.

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.

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

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.

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.

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.

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, D. Kraft, K. Mourão, N. Pugeault, N. Krüger, and M. Steedman. Representation and integration: Combining robot control, high-level planning, and action learning. In *Proc. of CogRob 2008 at ECAI 2008*, pages 32–41, 2008.

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.

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. Plans, affordances, and combinatory grammar. *Linguistics and Philosophy*, 25:723–753, 2002.

M. Steedman. Temporality. In J. van Benthem and A. ter Meulen, editors, *Handbook of Logic and Language*, pages 895–938. North Holland/Elsevier, Amsterdam, 1997.

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

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.