Project no.: **027657**

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

Project Acronym: **PACO-PLUS**

Deliverable no.: D5.1.2

Title of the deliverable: Extensions to PKS for support of low-level continuous control systems

Contractual Date of Delivery to the CEC:	31 July 2008	
Actual Date of Delivery to the CEC:	31 July 2008	
Organisation name of lead contractor for this deliverable:	UEDIN	
Author(s): Ronald Petrick, Christopher Geib, and Mark Steedman		
Participant(s): UEDIN		
Work package contributing to the deliverable:	WP4, WP5	
Nature:	R	
Version:	Final	
Total number of pages:	42	
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:

The core focus of WP5.1 is the generalisation of the basic symbolic representation of OACs and ancillary planning apparatus to communication and language. WP5.1 builds on WP3 to extend the representation of OACs to communicative acts and is tightly integrated with WP4, in particular the theoretical and practical components of WP4.3, to provide the foundational infrastructure needed to support high-level linguistic concepts. This deliverable focuses on one component of the WP4 architecture, the PKS planner, and describes a set of extensions required for PKS to support low-level continuous control systems, such as the SDU robot/vision system in WP4.1. Since the problem of planning dialogue acts can be viewed as an instance of the general problem of planning with incomplete information and sensing, the mechanisms needed to support ordinary action planning are also relevant to dialogue planning. The associated deliverable D5.2.2 describes the computational problem of natural language acquisition in greater detail.

Keyword list: Planning with Knowledge and Sensing (PKS), three-level control and communication architecture, integration of low-level robot/vision with high-level planning, dialogue planning infrastructure

Table of Contents

1. Executive Summary

The core focus of WP5.1 is the generalisation of the basic symbolic representation of OACs and ancillary planning apparatus to communication and language. More specifically, this work builds on WP3 to extend the representation of OACs to communicative acts (Task 5.1.1) and is tightly integrated with WP4, in particular the theoretical and practical components developed as part of WP4.3, to provide the foundational infrastructure needed to support high-level linguistic concepts (Task 5.1.2). This deliverable focuses on the latter task (5.1.2) and one essential component of our architecture, namely the PKS ("Planning with Knowledge and Sensing") planner [8, 9]. In particular, we describe the extensions required for PKS to support low-level continuous control systems, such as the SDU robot/vision system in WP4.1.

PKS is a state-of-the-art knowledge-level planner that is able to construct plans in the presence of incomplete information. Unlike traditional planners, PKS constructs plans at a more abstract level, by representing and reasoning about the planner's incomplete knowledge state. The representation used by PKS is based on an extended version of STRIPS [4] that explicitly models particular types of knowledge. Actions in PKS describe how they change the planner's knowledge state, rather than the world state. PKS can construct conditional plans with sensing actions, and supports numerical reasoning, run-time variables [3], 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. On the other hand, low-level robot systems typically use representations based on vectors of continuous values. Thus the task of integrating low-level robot systems with high-level planners requires overcoming the representational differences that exists between these two levels. This deliverable primarily focuses on the components we have currently developed to bridge this representational divide.

We also propose to use PKS as our planning framework for natural language and communication. Our first step is dialogue planning in PKS based on speech acts; a description of the theory behind our approach is outlined in [10]. In particular, we view the problem of planning dialogue acts as an instance of the general problem of planning with incomplete information and sensing. As a result, the mechanisms supporting ordinary action planning in PKS will also be relevant to dialogue planning. The associated deliverable D5.2.2 describes the computational problem of natural language acquisition in greater detail.

We have attached a number of additional documents to this deliverable that highlight the central role that PKS plays in the three-level architecture we have developed as part of WP4. These documents describe the additional support mechanisms that enable PKS to generate plans that can be executed in low-level continuous control spaces. More generally, these same components provide the infrastructure needed to support the longer term objectives of language and communication in WP5. Here we briefly sketch the relation of each paper to this workpackage and deliverable, and make links to the specific contributions of each paper.

[A] *(Internal PACO-PLUS Technical Report)* This document is an evolving specification of the interfaces required to integrate the low-level SDU robot/vision system, the mid-level UL memory component, and the high-level UEDIN planner and plan execution monitor. This document outlines a robot manipulation scenario that is realised on all three levels of representation. In particular, the high-level action representation used by PKS is described as an abstraction of the properties and capabilities available at the lower system levels. A control architecture and communication protocol are defined, illustrating how high-level plans can be generated by PKS for execution in the low-level continuous control space, and how high-level plan execution monitoring can be used to control replanning and resensing activities. We also demonstrate how PKS's ability to reason about incomplete information and sensing actions can be interpreted and executed by the lower system levels. This document describes the integration of components from WP4 and provides the needed technical basis for future work in WP5.

[B] *(Presented at CogSys-2008 in Karlsruhe, Germany)* This paper describes a mechanism for learning STRIPS and ADL [7] style actions effects from world state snapshots of the form produced by the control architecture in [A] above. The actual learning mechanism is based on kernel perceptrons [1, 5] which provide efficient learning and produce high quality models with low error rates. This mechanism was tested using data simulated from the same robot manipulation scenario described in [A] and shown to be effective at learning action effects in this domain. This work 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.

[C] *(Presented at CogRob-2008 in Patras, Greece)* This paper describes the integration of the SDU robot/vision system with the UEDIN planner and plan execution monitor in [A], together with the action effect learning mechanism in [B], from a representational point of view using the instantiated state transition fragment (ISTF) and object-action complex (OAC) concepts [6] developed as part of WP4. This document highlights how certain representational notions, like objects, are identified in low-level control space and are introduced into the high-level planning representation. Moreover, this paper illustrates how high-level plans constructed by PKS—including conditional plans that reason about incomplete information and sensing actions—can be executed by the robot/vision system.

Together, these papers report a number of significant developments:

- The three-level control architecture and associated communication protocol provides the necessary abstraction layer needed to support high-level planning with PKS in continuous low-level robot domains like our robot manipulation scenario.
- The early integration of the SDU robot/vision system with the PKS planner has been completed, allowing both simple linear plans and conditional plans with sensing actions to be constructed by the planner and executed on the robot platform.
- A role for the UL mid-level memory component has been identified within the control architecture, offering the prospect of improved high-level plan specification through sub-symbolic manipulation and reasoning.
- The kernel perceptron learning component in [B] has been implemented and tested on data simulated from the shared integration scenario in [A], providing a mechanism for learning a set of high-level action effects, usable as part of PKS's action representation.
- The work described in this report provides a complete theoretical path from continuous low-level representations to high-level models suitable for planning and the support of language. The representational structures underlying our integration efforts make use of the ISTF and OAC concepts, previously defined as part of WP4. These structures, as well as the associated control mechanisms already implemented, provide the necessary infrastructure needed to support the longer term goals of WP5, such as dialogue planning.

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

- As a collaboration with UniKarl, UEDIN is starting to integrate its control, communication, and planning mechanisms on the ARMAR robot platform as part of WP1, and is in the process of extending the robot scenario in [A] to take into consideration ARMAR's extended action capabilities.
- The plan execution monitor is currently being built by UEDIN and has not yet been tested within the integrated SDU/UEDIN system.
- The action effect learning mechanism has not yet been tested on data generated by the real robot/vision system. Furthermore, we have not generated plans for execution on the robot using the learnt action models.

- 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 [11] which has been successfully applied by planners in the probabilistic track of the International Planning Competition [2].
- Although the theoretical work required to extend PKS to support dialogue planning of the form described in deliverable D5.2.2 and [10] is complete, the implementation of these extensions (Milestone 5.1.1) is only partially complete at the time of reporting.

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

2. Publications Associated with D5.1.2

[A] **A Scenario for Integrating Low-Level Robot/Vision, Mid-Level Memory, and High-Level Planning with Sensing**
Ronald Petrick, Christopher Geib, and Mark Steedman
Internal PACO-PLUS Technical Report, July 2008.

> **Abstract:** In this document we provide an overview of a proposed scenario for integrating the University of Southern Denmark (SDU)'s robot/vision system, the University of Leiden (UL)'s mid-level memory/reasoning component, and the University of Edinburgh (UEDIN)'s high-level planner and plan execution monitor. We also give an overview of our proposed extensions for incorporating dialogue planning into the scenario, using UEDIN's planning system. This document builds on discussions between project partners from SDU, UL, the University of Liège (ULg), and UEDIN at a meeting held in Leiden in September 2007, and captures some of our ideas from the point of view of the planning task and required high-level representation. Since our integration discussions are ongoing, this document should be viewed as a snapshot of our current thinking (and subject to change in the future). This document describes the integration of 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.

[B] **Using Kernel Perceptrons to Learn Action Effects for Planning**
Kira Mourão, Ronald Petrick, and Mark Steedman
Published at the International Conference on Cognitive Systems (CogSys), 2008.

> **Abstract:** We investigate the problem of learning action effects in STRIPS and ADL planning domains. Our approach is based on a kernel perceptron learning model, where action and state information is encoded in a compact vector representation as input to the learning mechanism, and resulting state changes are produced as output. Empirical results of our approach indicate efficient training and prediction times, with low average error rates ($< 3\%$) when tested on STRIPS and ADL versions of an object manipulation scenario. This work is part of a project to integrate machine learning techniques with a planning system, as part of a larger cognitive architecture linking a high-level reasoning component with a low-level robot/vision system.

[C] **Representation and Integration: Combining Robot Control, High-Level Planning, and Action Learning**
Ronald Petrick, Dirk Kraft, Kira Mourão, Christopher Geib, Nicolas Pugeault, Norbert Krüger, and Mark Steedman
Published at the 6th International Cognitive Robotics Workshop (CogRob), pp. 32–41, 2008.

Abstract: We describe an approach to integrated robot control, high-level planning, and action effect learning that attempts to overcome the representational difficulties that exist between these diverse areas. Our approach combines ideas from robot vision, knowledge-level planning, and connectionist machine learning, and focuses on the representational needs of these components. We also make use of a simple representational unit called an instantiated state transition fragment (ISTF) and a related structure called an object-action complex (OAC). The goal of this work is a general approach for inducing high-level action specifications, suitable for planning, from a robot's interactions with the world. We present a detailed overview of our approach and show how it supports the learning of certain aspects of a high-level representation from low-level world state information.

References

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

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

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

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

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

[6] Christopher Geib, Kira Mourão, Ron Petrick, Nico Pugeault, Mark Steedman, Norbert Krueger, and Florentin Wörgötter. Object action complexes as an interface for planning and robot control. In *Proc. of Humanoids-2006 Workshop: Towards Cognitive Humanoid Robots*, 2006.

[7] Edward Pednault. ADL: Exploring the middle ground between STRIPS and the situation calculus. In *Proceedings of the 1st International Conference on Principles of Knowledge Representation and Reasoning*, pages 324–332, Palo Alto CA, 1989. Morgan Kaufmann.

[8] 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.

[9] 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.

[10] 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.

[11] 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

A Scenario for Integrating Low-Level Robot/Vision, Mid-Level Memory, and High-Level Planning with Sensing

Ronald Petrick, Christopher Geib, Mark Steedman

20 July 2008 / Internal PACO-PLUS technical report

Abstract

In this document we provide an overview of a proposed scenario for integrating the University of Southern Denmark (SDU)'s robot/vision system, the University of Leiden (UL)'s mid-level memory/reasoning component, and the University of Edinburgh (UEDIN)'s high-level planner and plan execution monitor. We also give an overview of our proposed extensions for incorporating dialogue planning into the scenario, using UEDIN's planning system. This document builds on discussions between project partners from SDU, UL, the University of Liège (ULg), and UEDIN at a meeting held in Leiden in September 2007, and captures some of our ideas from the point of view of the planning task and required high-level representation. Since our integration discussions are ongoing, this document should be viewed as a snapshot of our current thinking (and subject to change in the future). This document describes the integration of components developed as part of workpackage 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.

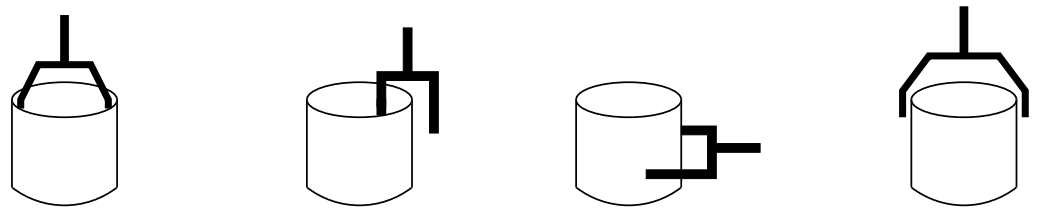
1 Object stacking with sensing

The domain we have chosen for our integration task is a simple object manipulation scenario.[1] We assume a *table* with a number of *objects* that are graspable by the robot. We consider situations with no more than 5 objects and, initially, only 1-2 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 determines whether or not we can *stack* objects inside other objects, provided the object sizes permit such stacking.

The goal of the scenario is to clear all open objects from the table, by removing them to some designated location (e.g., a box, a shelf, a hole, a corner of the table, etc.). The location may furthermore be restricted in such a 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 planning system 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. Object openness plays two important roles in this scenario: (i) as a goal condition that determines which objects should be removed from the table, and (ii) as a prerequisite for stacking operations.

This scenario is meant to provide a basis for integrating the robot/vision, mid-level memory, and high-level planning components of the system. The planner is responsible for constructing a plan that achieves the goal of clearing open objects from the table, by working with a high-level representation of the scenario. The job of the mid-level component in this case is to *refine* such plans with regard to the sensing actions contained in these plans. In particular, the robot/vision system will be able to ascertain whether an object is open or not by one of two means: (i) it can *poke* an object in order to verify its concavity, or (ii) it can *focus* the vision system on the object at a higher level of resolution. The mid-level memory system must make an informed choice between poking and focusing operations, update the plan as appropriate, and pass the

[1] For the purposes of distinguishing between the three levels in this document we will use the tags "robot", "memory", and "planner" to denote the SDU, UL, and UEDIN components, respectively.



(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

augmented plan on to the robot/vision system. Ultimately, the robot/vision system must be able to interpret, understand, and execute the plans generated and refined by the upper levels.

For the remainder of this document we will mainly focus on the top-down task, in particular, describing the high-level planning representation that is passed to the mid-level memory system, and the message passing protocol that supports the exchange of messages between the levels.

2 High-level planning representation

Given the above scenario description, we define a set of high-level actions and properties that allows the planner to operate in this domain, and provide some insights as to how these actions/properties relate to the memory and robot/vision levels.

2.1 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`.

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.

We have also encoded four actions for moving and manipulating objects when successfully grasped:

putInto-objectOnTable(?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.

putOnTable(?x) - Put object ?x onto the table.

putAway(?x) - Put object ?x away.

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 being put away onto a shelf. We also note that both putInto-objectOnTable and putInto-stack actions denote stacking operations which will have as a prerequisite the property that objects can only be stacked into open objects.

2.2 Sensing actions

The high-level planning representation also consists of a single sensing action:

sense-open(?x) - Determine whether object ?x is open or not.

At the planning level, this action is modelled an information gathering or *knowledge-producing action* that provides the planner with additional information about an object's state. At the robot/vision level, sense-open will ultimately be executed as either a *poke* operation which tests the object's concavity, or a *focus* operation which directs the vision system to study the object at a higher resolution. The mid-level memory system is responsible for refining high-level sense-open actions into robot/vision operations that are appropriate, given the context, and that can be understood by the lower level.

2.3 Properties

Our planning-level domain encoding makes use of a set of predicates and functions which we have agreed could reasonably be provided to the planner as a result of sensor information from the robot/vision level. These properties are subject to change as the domain model is refined through further discussions.

open(?x) - A predicate indicating that object ?x is open.

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.

ontable(?x) - A predicate indicating that object ?x is on the table.

onshelf(?x) - A predicate indicating that object ?x is on the shelf.

isin(?x,?y) - A predicate indicating that object ?x is stacked in object ?y.

clear(?x) - A predicate indicating that no object is stacked in ?x.

instack(?x,?y) - A predicate indicating that object ?x is in a stack with object ?y at its base.

radius(?x) = ?y - A function indicating that the radius of object ?x is ?y.

shelfspace = ?x - A function indicating that there are ?x empty shelf spaces.

reachableA(?x)
reachableB(?x)
reachableC(?x)
reachableD(?x) - Predicates indicating that object ?x is reachable by the gripper using a particular grasp.

graspAMinRadius = ?x
graspAMaxRadius = ?x
graspBMinRadius = ?x
graspCMaxRadius = ?x
graspDMaxRadius = ?x - Functions indicating the min./max. radius restrictions for each grasp type.

2.4 Domain encoding

Using the above properties we can write PKS [Petrick and Bacchus, 2002, 2004] operators for the actions we require. For simplicity, we have made the following restrictions in our domain 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 `putOnTable` action will initially be omitted (since there are no initial stacks).

Our current domain encoding is given in Table 1. 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 from future discussions.

PKS action description notation: the domain encoding in Table 1 is very much like a standard STRIPS [Fikes and Nilsson, 1971] encoding except that PKS, unlike STRIPS, 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 of PKS's databases. (K_f is very much like a standard STRIPS databases that stores the planner's knowledge of facts, and K_w is a specialized database for storing the effects of sensing actions.) As well, $\neg K_w$`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).

Action	Preconditions	Effects
sense-open(?x)	$\neg K_w$(open(?x)) ontable(?x)	add(K_w,open(?x))
graspA-fromTable(?x)	reachableA(?x) clear(?x) gripperempty ontable(?x) radius(?x) $\geq$ graspAMinRadius graspAMaxRadius $\geq$ radius(?x)	add(K_f,ingripper(?x)) add(K_f,$\neg$gripperempty) add(K_f,$\neg$ontable(?x))
graspA-fromTopOfStack(?x)	reachableA(?x) clear(?x) gripperempty radius(?x) $\geq$ graspAMinRadius graspAMaxRadius $\geq$ radius(?x) ($\exists$?z). instack(?x,?z) ontable(?z)	add(K_f,ingripper(?x)) add(K_f,$\neg$gripperempty) ($\forall$?y). isin(?x,?y) $\Rightarrow$ del(K_f,isin(?x,?y)) add(K_f,clear(?y)) ($\forall$?z). instack(?x,?y) $\Rightarrow$ del(K_f,instack(?x,?z))
graspB-fromTable(?x)	reachableB(?x) clear(?x) gripperempty ontable(?x) radius(?x) $\geq$ graspBMinRadius	add(K_f,ingripper(?x)) add(K_f,$\neg$gripperempty) add(K_f,$\neg$ontable(?x))
graspD-fromTable(?x)	reachableD(?x) gripperempty ontable(?x) graspDMaxRadius $\geq$ radius(?x)	add(K_f,ingripper(?x)) add(K_f,$\neg$gripperempty) add(K_f,$\neg$ontable(?x))
putInto-objectOnTable(?x,?y)	?x $\neq$?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)) ($\forall$?w). instack(?w,?x) $\Rightarrow$ del(K_f,instack(?w,?x)) add(K_f,instack(?w,?y))
putInto-stack(?x,?y)	?x $\neq$?y ingripper(?x) open(?y) clear(?y) radius(?y) > radius(?x) ($\exists$?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)) ($\forall$?z). instack(?y,?z) $\Rightarrow$ add(K_f, instack(?x,?z)) ($\forall$?w). instack(?w,?x) $\Rightarrow$ 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

Table 1: PKS-style action descriptions

3 Example plans

In this section we give three examples of planning problems we can solve using PKS and the above action descriptions. In each example we consider a scenario with 2 objects initially on the table. Each object also 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.) In each example we assume the following initial conditions:

- Objects: `obj1`, `obj2`
- `radius(obj1) = 1`, `radius(obj2) = 4`
- `shelfspace = 1`
- 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 the objects on a shelf which has limited space. In Example 1, the planner initially knows that both objects are open and, thus, does not need to include sensing actions in the plan. 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.

3.1 Example 1

The planner initially knows that `open(obj1)` and `open(obj2)` are true.

Plan:

```
graspA-fromTable(obj1)
putInto-objectOnTable(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.

3.2 Example 2

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 Example 3

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-objectOnTable(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.

4 Message passing protocol

In this section we describe a simple message passing protocol for exchanging information between the robot/vision, memory, and planning levels. We begin by defining the kinds of messages that can be passed between the system levels. We then describe a simple control architecture that is sufficient for our proposed integration task, and provide some details of a communication library (supplied by UEDIN) that implements this protocol.

4.1 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 its *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 complete set of messages is given in Table 2 and the send/receive message pairs are given in Table 3.[2]

4.2 Message passing control loop

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 4). 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, however, 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.

4.2.1 Robot-level control loop

At the robot level, the message-processing control loop follows a relatively simple repeating pattern where the robot essentially drives the message-passing process and the upper levels of the system respond to

[2]The message set is still subject to change and may be expanded or streamlined in the future.

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 2: Available message types 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 3: Send/receive message pairs

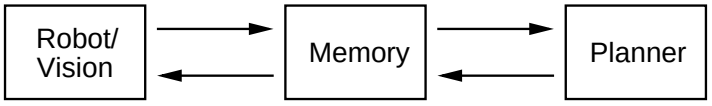


Table 4: Flow of messages between the three system levels

queries. The robot-level control loop defines a very simple synchronous cycle wherein 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, we see the interaction between the robot and the higher levels follow the *RobotLevel-ControlLoop* pseudo code given in Figure 2(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.

4.2.2 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.

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 notable exception is the receipt of `MSG_ACTION_SUBMIT` messages from the planner. Before forwarding such messages, the memory must inspect the message contents to check for sensing actions, which may need to be refined. In the context of the simple integration scenario described in this document, the memory must transform all `sense-open` actions into *poke* or *focus* operations before passing them on to the robot. (In the future, the memory may also be responsible for refining 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 help in its decision making concerning refinement activities.

The pseudo code for the memory-level control algorithm is given in Figure 2(b).

4.2.3 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 analysed in order to help direct future operations of the memory level, or refine future actions sent to the robot. Future versions of the communication protocol may also allow the planner to directly "push" such plan information to the memory, for instance as a result of replanning operations. The general planning-level control algorithm is given in Figure 2(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 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 with subsymbolic information between the robot and memory, and messages with symbolic information between the memory and planner). 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, we have designed our message set to support much more complex and flexible control architectures, which might arise in the future. For our initial integration tasks, however, the existing process we have outlined is more than sufficient.

Direct link between SDU and UEDIN: In terms of the initial integration efforts between SDU and UEDIN, the above protocol does not specify any major changes to existing work. Instead, the memory level can be viewed as a message-forwarding module that holds the place of the full mid-level memory component, in order to bring the existing SDU/UEDIN architecture in line with the protocol described here. Although this module will simply pass messages to the other system levels, its addition should facilitate the inclusion of a more fully-featured module into current integration efforts at a later date when a memory system is made available. The necessary code for the message-forwarding module is provided as part of UEDIN's supplied communication library.

4.3 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.[3] Currently, the content of the `MSG_STATE_UPDATE` message must be a list of instantiated properties from Section 2.3 that form the world state. Similarly, the action specification content of the `MSG_ACTION_SUBMIT` message is a single instantiated action from Sections 2.1 or 2.2. The content of the `MSG_PLAN_SUBMIT` message will be a plan similar to those in Section 3, but encoded as a Prolog-style list

[3]The current version of the communication library also defines messages 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.

```
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 2: Message passing control algorithms

(see Section 5 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 provided with the socket library.)

For initial testing purposes we terminate 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 will then pass this message on 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 robot, memory, and planner components. These programs focus solely on the communication interface, with little additional functionality. (For instance, the memory level program simply forwards messages without "refining" sensing actions and always requests a complete plan after the first robot level request for an action.) It is hoped that these programs can serve as the basis for the development of more sophisticated modules that can simply be "plugged" into the communication architecture. A series of pregenerated plans are included with this software, however, to test the message exchanges between the three levels.

5 Message passing example

To better understand the flow of messages between the three system levels, we consider the scenario in Example 2, where the robot is tasked with the goal of clearing the open objects from a table. Figure 3 shows the messages sent by the three levels during the execution of the first action, `sense-open(obj2)`, in the conditional plan constructed for Example 2 (i.e., one 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 on initialization the robot/vision system will send such an "unusually complete" world description, as a bootstrapping action. From the perspective of the planning system such a message is no more than a particularly large state update, and requires no extra machinery.

Given an initial state description, the planner can construct 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 3 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 the system, the memory level acts as a forwarding service between the robot and the planner. (In the future the memory may 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 this message is a sensing action, `sense-open(obj2)`, the memory refines 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 3 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 in Example 2, encoded in a Prolog-style list format for transmission using the communication library.[4]

[4]The communication library provides a helper class for processing plans in the compact list 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 3. (In the sample code the memory sends a `MSG_PLAN_REQUEST` after receipt of the first `MSG_ACTION_SUBMIT` message from the planner.) Similarly, alternate message orderings—including messages sent in parallel from 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 our message passing protocol ensures that such ordering differences do not lead to problems like deadlock, however.

6 Plan execution and monitoring

Although we are able to construct plans for the proposed object manipulation scenario, and communicate those plans to the other system levels using the message passing protocol, we must still be concerned with how plan failure information should be exchanged between the planner and the other system levels.

In discussions with SDU we have identified the need for a high-level mechanism that would operate closely with the planner in order to monitor plan execution and control replanning/resensing activities. A *plan execution monitor*, currently being built by UEDIN, 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 state information will be used to decide between (i) continuing the execution of an existing 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 support of the resensing activities described in (ii), we have agreed in discussion with SDU that the plan monitor could initially provide the vision system (possibly through the memory level) with a list of the objects that were "relevant" to the execution of the action that is reported to have failed, as 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 and, possibly (as future work), an updated high-level action model.

In terms of the integration scenario described in this document, the plan execution monitor will have the added task of managing the execution of plans with conditional branches. Plan branches result from the inclusion of explicit sensing operations (like `sense-open`) into a plan. When a sensing action is ultimately executed at the robot level, the result of the sensing will be returned to the planner through the memory level, as part of the standard state update cycle. When faced with a conditional branch point in a plan, the plan execution monitor will then 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 above. 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 short term, in order to evaluate its effectiveness on plans being executed in the actual robot environment.

	ROBOT	MEMORY	PLANNER
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 3: Example of messages sent during the execution of the sense-open(obj2) action in Example 2

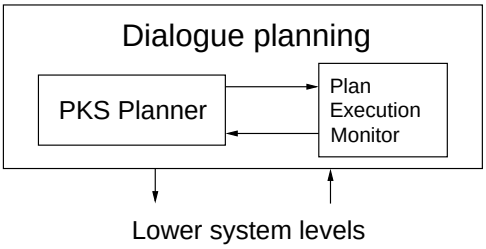


Figure 4: Dialogue planning as an instance of standard action planning

7 Towards language and communication through dialogue planning

In this document we have primarily focused on the robot/memory/planner integration task, with an emphasis towards standard action planning. As outlined in the objectives of workpackage WP5, the mechanisms supporting the symbolic representation of action and ancillary planning apparatus will be generalised to language and communication, enabling planning with communicative acts. Our first step towards this goal—dialogue planning based on speech acts—is outlined in [Steedman and Petrick, 2007] and describes 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].

One of the important insights in [Steedman and Petrick, 2007] is that the problem of planning dialogue moves can be viewed as an instance of the more general AI problem of planning with incomplete information and sensing. By incorporating ideas from PKS to the representation of dialogue acts in LDEC (our high-level symbolic representation of OACs), we are able to 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 support dialogue planning. 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 (see Figure 4). However, our extensions will also 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. More details about the dialogue planning component will be described in a future version of this document.

8 Discussion

- 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 components of the grasp that failed.

- In this document we only consider the refinement of `sense-open` actions at the memory level. In the future, we could also accommodate the refinement of grasp actions. For instance, the planner could generate plans with abstract actions like `grasp(obj1)`. It would then be the task of the memory level to make a decision as to the choice of grasping option and transform such actions into more specific robot-level actions like `graspA(obj1)` or `graspD(obj1)`.

- As future work, we believe that a more complex interaction between the robot, memory, and planning levels might be desirable. For instance, we may want the planning level to have the ability to terminate an action during its execution if it is having an undesirable outcome, or alert the memory about a replanning operation. This would require a more asynchronous ("push") architecture, including state update messages from the robot during action execution, as well as the ability to issue halt commands from the planning level. Moreover, 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. We believe that such extensions will not require a significant reworking of the message passing protocol, but only slight extensions to the message set and control algorithm.

- We also envision a more significant extension to the message passing protocol to support the addition of new objects, new properties, and new 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 at the code level of the socket library, however, future versions of the message passing protocol will more fully specify the operation of these message types.

- There are a number of places where incompleteness of information in the world state update can come 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 believe that we must seriously examine the limitations of the system's capability for providing complete state updates, as well as the traditional AI assumption that we have a complete model of the state changes that result from executed actions. This is a very 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.

- We have begun working with the University of Karlsruhe (UniKarl) in an effort to incorporate the UEDIN communication architecture and planning components onto the ARMAR robot platform. Initial integration work has started and we are in the process of defining a new scenario that takes into consideration ARMAR's capabilities (e.g., multiple grippers, different grasping options, etc.). While this new scenario will extend the SDU/UL/UEDIN scenario described in this document, the core scenario (and action representation) will be essentially unchanged. Moreover, we believe that the communication and planning architecture will transfer to the ARMAR platform without change, since these components provide an abstraction layer above the robot level.

References

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

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

R. P. A. Petrick and F. Bacchus. A knowledge-based approach to planning with incomplete information and sensing. In *Proceedings 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.

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 *Proceedings of the 8th SIGdial Workshop on Discourse and Dialogue (SIGdial 2007)*, pages 265–272, Antwerp, Belgium, Sept. 2007.

Appendix B

Using Kernel Perceptrons to Learn Action Effects for Planning

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

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

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

***Abstract*— We investigate the problem of learning action effects in STRIPS and ADL planning domains. Our approach is based on a kernel perceptron learning model, where action and state information is encoded in a compact vector representation as input to the learning mechanism, and resulting state changes are produced as output. Empirical results of our approach indicate efficient training and prediction times, with low average error rates ($< 3\%$) when tested on STRIPS and ADL versions of an object manipulation scenario. This work is part of a project to integrate machine learning techniques with a planning system, as part of a larger cognitive architecture linking a high-level reasoning component with a low-level robot/vision system.**

I. Introduction

Artificial intelligence planning systems provide a powerful tool for controlling a cognitive agent's actions in both real-world and artificial domains. A drawback of such approaches is that they require a model of the dynamics of the domain in which the agent will operate. In real-world domains, such models may not be readily available, or may not properly account for unexpected subtleties that arise in the world when a model is constructed *a priori* by hand. An alternative, more desirable approach is to endow the agent with the ability to *learn* from its environment in order to induce a world model, and the effects of its actions, from its experiences.

Using machine learning techniques to learn action models is not a new idea. Prior approaches have applied inductive learning [1] and directed experimentation [2] techniques to data represented in first-order logic, without noise or non-determinism. Other approaches have used schema learning to learn probabilistic action rules operating on discrete-valued sensor data [3]. Also, k-means clustering of equivalence classes, followed by extraction of sensor data features, has been used to train support vector machines (SVMs) to predict deterministic action effects in a given context [4]. Recently, attention has shifted to methods which exploit relational structure in order to improve speed and generalisation performance. For instance, [5] generates and refines rules using heuristic search guided by maximum likelihood, and shows that relational deictic rules are learnt more effectively than propositional or purely relational rules. Similarly, [6] uses a logical inference algorithm to efficiently learn rules in relational environments.

Our approach is based on a connectionist learning model, namely *kernel perceptron learning* [7], [8]. Such methods are particularly useful since they can be shown to provide good performance, in terms of both the training time and the quality of the learnt models. We focus on one aspect of the learning problem in this paper, namely learning the effects of an agent's actions, given a set of actions and their preconditions. Currently, our learning method assumes a fully observable world for training purposes (i.e., complete world state descriptions), however, it can be made much more general. For instance, our approach can be extended to handle noisy data [9], and we believe it can also be used to learn action preconditions and more complex representations.

Since we would like to apply our techniques to real planning systems, we will focus on two different types of action representations commonly used in the planning community: STRIPS actions [10] and ADL actions with conditional effects [11]. We consider deterministic domains with actions that affect a subset of the properties (predicates) that make up the world state. In our approach, we use a representation that makes efficient use of predicates, and follow the approach of [5] where deictic referencing is used to reduce the complexity of the representation. We demonstrate that kernel perceptrons can be used successfully to learn the dynamics of an object manipulation domain, in a manner that is independent of the number of objects in the world, making it suitable for large planning scenarios.

This paper is organized as follows. In Section II we discuss the planning representations we are interested in learning. In Section III we describe kernel perceptrons and how we use them to learn action effects. In Section IV we present the results of our learning experiments. In Sections V and VI we discuss our results and our plans to incorporate these techniques into a cognitive architecture linking a high-level planning system to a low-level robot/vision system.

II. Action representations for planning

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

TABLE I

STRIPS ACTIONS FROM AN OBJECT MANIPULATION DOMAIN

Action	Preconditions	Effects
graspA-table(x)	*clear*(x) *gripperempty* *ontable*(x)	*add*(*ingripper*(x)) *del*(*gripperempty*) *del*(*ontable*(x))
graspA-stack(x, y, z)	*clear*(x) *gripperempty* *isin*(x, y) *instack*(x, z)	*add*(*ingripper*(x)) *add*(*clear*(y)) *del*(*gripperempty*) *del*(*isin*(x, y)) *del*(*instack*(x, z))

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

Each action $a \in \mathcal{A}$ is defined by a set of *preconditions*, Pre_a, and a set of *effects*, Eff_a. Pre_a can be any set of fluents and negated fluents. We consider two different kinds of action effects, both of which are commonly found in planning domains. In STRIPS actions [10], each effect $e \in Eff_a$ has the form *add*(p) or *del*(p), where p is any fluent. In ADL actions [11], each effect $e \in Eff_a$ is either a standard STRIPS effect, or a *conditional effect* of the form $C_e \Rightarrow$ *add*(p) or $C_e \Rightarrow$ *del*(p). Here, C_e is any set of fluents and negated fluents, and is referred to as the *secondary preconditions* of effect e. Action preconditions and effects can also be parametrized. An action with all of its parameters replaced with objects from $\mathcal{O}$ is said to be an *action instance*.

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

In this paper we will focus on a specific object manipulation scenario, represented as a simple planning domain, where a robot has the ability to grasp, stack, and remove objects from a table environment.[1] For instance, the domain includes actions like *graspA-table*(x) ("grasp object x from the table using grasp type A"), *graspA-stack*(x, y, z) ("grasp object x from object y in a stack with z at its base using grasp type A"), and *putAway*(x) ("put object x away on a shelf"). The domain also includes properties like *gripperempty* ("the robot's gripper is empty"), *clear*(x) ("object x has no objects on top of it"), *ontable*(x) ("object x is on the table"), and *isin*(x, y) ("object x is in object y"). As we'll see in Section V, this domain is motivated by work that aims to link a robot/vision system with a planner, within an architecture where the robot can learn and act [12], [13].

[1]This domain is similar to Blocksworld, but has been extended to include more complex grasping actions and management of limited resources.

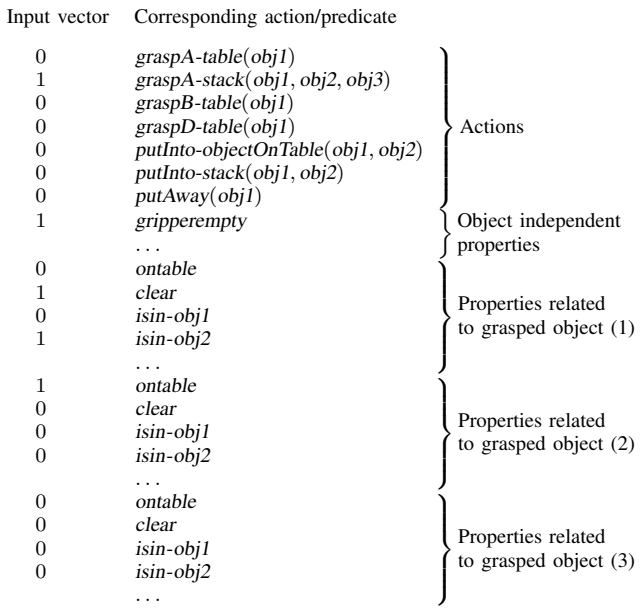


Fig. 1. Representation of an action and state as a binary input vector

Table I shows the encoding of two STRIPS actions in our object manipulation domain, *graspA-table* and *graspA-stack*. Actions such as these provide a straightforward representation of the manipulation tasks that can be performed. For instance, if we have a state s defined by the set {*clear*(*obj1*), *clear*(*obj2*), *gripperempty*, *ontable*(*obj1*), *ontable*(*obj2*)}, then the action instance *graspA-table*(*obj1*) is applicable at s. Applying the effects of this action instance to s produces the new state $s' =$ {*clear*(*obj1*), *clear*(*obj2*), *ingripper*(*obj1*), *ontable*(*obj2*)}. We will also consider an ADL version of this domain in our testing.

III. KERNEL PERCEPTRON ACTION LEARNING

The planning actions in the previous section give rise to a simple state transition system, where the application of an action at a state produces a new state. In this model, an action's effects determine the *changes* made to a state during execution. Since a state is simply a set of fluents, the transition between states is simply the difference between two sets of fluents. Our goal in this section is to develop an approach that *learns* these differences between states.

Learning the complete dynamics of a planning domain requires the ability to learn both the preconditions needed to perform an action, and the effects of the action at a particular state. In this paper, we will only focus on the effects problem, and will simply assume that the action set and action preconditions are supplied to our learning mechanism as part of the input. (We also believe our approach extends to the problem of learning action preconditions; see Section V).

The specific learning method we will use is a connectionist machine learning model based on *kernel perceptrons* [7], [8]. Kernel perceptrons obtain reasonable accuracy at acceptable training and prediction speeds, allowing us to use this approach in practical planning applications. Alternative non-

linear classifiers, such as SVMs, can be substantially slower [14] while performance is not guaranteed to be better [15].

In order to effectively use kernel perceptrons, we must consider how best to encode our learning problem in terms of the inputs and outputs of the learning mechanism. We consider each of these problems in turn, as well as the overall operation of our learning approach.

A. Input representation

The input to our learning mechanism uses a vector representation that encodes a description of the action being performed and the state at which the action is applied. For each action in the domain, the vector includes an element that is set to 1 if the action is to be performed, or 0 otherwise. For states, we consider object-independent and object-dependent properties separately. In the case of object-independent properties (e.g., *gripperempty*), the vector includes a single element for each property of the domain, representing the truth value of that property (fluent) at the state being considered: the element is set to 1 if the fluent is true at the state, or 0 if the fluent is false at the state.

For object-dependent properties, we avoid representing all possible fluents, which could lead to very large input vectors. Instead, we consider each property on a per object basis, by representing only those properties of the objects directly involved in the action being applied, and the objects related in some way to those objects. Additionally, a form of deictic representation is used (similar to [5]), where objects are specified in terms of their roles in the action, or their roles in a predicate with another object. For example, in Table I the only object involved in *graspA-table* is the "grasped object" x. In *graspA-stack* the objects include the "grasped object" x, and the related objects "object containing the grasped object" y and "object at base of grasped object's stack" z.

Rather than maintaining a "slot" in the input vector for each possible role, roles are allowed to overlap. The only constraint is that two objects with the same role in the same action in two different instances of the action must always be represented at the same slot in the input vector. Thus, each object is represented by a set of inputs, one for each object-specific predicate (such as *ingripper*), and each relation with another object (such as *isin*). To bind relations to the correct objects, extra predicates are used which relate the current object to one or more other objects, identified by their slot (e.g., *isin-obj1*, *isin-obj2*, etc.). This representation significantly reduces the number of inputs to the learning mechanism, and is dependent on the complexity of the actions and relations between objects, rather than the number of objects in the domain.

The final input vector has the form: $\langle$*actions, object-independent properties, object slot 1 predicates, object slot 2 predicates, ..., object slot n predicates*$\rangle$. Fig. 1 shows an example of an input vector for an action-state pair. In this case, the action performed is *graspA-stack*. The "grasped object" properties are represented in the object *obj1* slot, while the "object below the grasped object" properties are represented in the object *obj2* slot. Here, *clear*$(obj1)$, *isin*$(obj1, obj2)$ and *ontable*$(obj2)$ are shown to be true. No further object properties are included in the state in this example, and so all the remaining bits are set to 0.

B. Output representation

The output of the learning mechanism is a prediction of the set of domain properties that will change if the given action is performed at the given state. As with the input, this is encoded as a binary vector, with each output representing a state property: the output value is 1 if the property changes and 0 if it does not. As with the input vector, object-independent properties are represented by single elements, while object-specific properties are again represented on a per-object basis in slots. Thus, the output vector has the form: $\langle$*object-independent properties, object slot 1 predicates, object slot 2 predicates, ..., object slot n predicates*$\rangle$.

C. Learning

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

A simple, fast, binary classifier that can be used to address our particular learning problem is the *perceptron* [16]. The perceptron maintains a weight vector $\mathbf{w}$ which is adjusted at each training step. The i-th input vector $\mathbf{x}_i \in \{0,1\}^n$ in a class $y \in \{-1, 1\}$ is classified by the perceptron using the decision function $f(\mathbf{x}_i) = sgn(\langle \mathbf{w} \cdot \mathbf{x}_i \rangle)$. If $f(\mathbf{x}_i)$ is not the correct class then $\mathbf{w}$ is set to $\mathbf{w} + y\mathbf{x}$; if $f(\mathbf{x}_i)$ is correct then $\mathbf{w}$ is left unchanged. Provided the data is linearly separable, the perceptron algorithm is guaranteed to converge on a solution in a finite number of steps [17], [18]. If the data is not linearly separable then the algorithm oscillates, changing $\mathbf{w}$ at each misclassified input vector.

One solution for non-linearly separable data is to map the input feature space into a higher-dimensional space where the data is linearly separable. However, an explicit mapping leads to a massive expansion in the number of features which may make the classification problem computationally infeasible. Instead, an implicit mapping can be achieved by applying the *kernel trick* to the perceptron algorithm [8]. The kernel trick is applied by noting that the decision function can be written in terms of the dot product of the input vectors:

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

where α_j is the number of times the j-th example has been misclassified by the perceptron. By replacing the dot product

with a *kernel function* $k(\mathbf{x}_i, \mathbf{x}_j)$ which calculates $\langle \phi(\mathbf{x}_i) \cdot \phi(\mathbf{x}_j) \rangle$ for some mapping ϕ, the perceptron algorithm can be run in higher dimensional space without ever requiring the mapping to be explicitly calculated.

Since in general the problem of learning action effects is not linearly separable, the kernel perceptron is an appropriate choice for this problem. An ideal kernel is one which allows the perceptron algorithm to run over the feature space of all conjunctions of features in the original input space, as this would allow an accurate representation of the exact conjunction of features (action and preconditions) corresponding to a particular effect. Such a kernel is $k(x, y) = 2^{same(x,y)}$, where $same(x, y)$ is the number of bits with the same value in both x and y [19], [20]. As we'll see in Section V, we also believe our approach can be made much more general, letting us relax some of our earlier restrictions.

IV. Empirical results

In this section we describe the results of testing our learning procedure on the object manipulation domain described in Section II. Data was simulated from the domain description, for both a purely STRIPS version of the actions and an ADL version with context-dependent action effects. (For example, the two STRIPS actions in Table I were merged into a single ADL action, along with other changes.) Each case was generated by randomly selecting an action, and setting the inputs for the preconditions required for the action to 1. The action input was set to 1, and all other action inputs to 0. The remaining irrelevant inputs were used to create separate training and testing input data sets. For the training data, half of the inputs in each instance were randomly set to 0 or 1, with the other half all set to 0 (vice versa for the testing data). Outputs were set to 1 if the feature changed as a result of the action and 0 if not. Overall, 3000 training and 500 testing examples were generated with the (strong) assumptions that (i) no noise was present in the training/testing set, and (ii) no irrelevant output data was included in the training examples (i.e., only relevant changes were provided). To determine an error bound on our results, 10 runs with different randomly generated training and testing sets were used. Our testing environment was a 2.4 GHz quad-core system with 6 Gb of RAM. All times were measured for Matlab 7.2.0.294.

The results of our testing are shown in Fig. 2. Overall, the kernel perceptron learnt the training data and performed well on the testing data with a low error rate. Fig. 2(a) shows the error rate for the learnt STRIPS actions, while Fig. 2(b) shows the error rate for the ADL actions. In both cases, the average error dropped to less than 3% after 700 training examples. The standard perceptron error rate, included for comparison, shows significantly worse performance: over 5% error after 3000 training examples. Fig. 2(c) shows the training time for both STRIPS and ADL actions (for 1 bit of the effect vector), while Fig. 2(d) shows the prediction time (for 1 bit of 1 prediction). In both cases, the kernel perceptron method is quite efficient. Perhaps the most surprising result is that there is little difference between the training and prediction times of STRIPS actions, compared with those for ADL actions, at least for our particular testing domain. In general, performance on ADL domains will always take longer than STRIPS domains, particularly when the conditional effect training examples are very dissimilar to the other training examples available. (The ADL problem is slightly more difficult, even in our domain.) Additional testing is needed on more complex domains, to determine to what extent the STRIPS and ADL results remain similar.

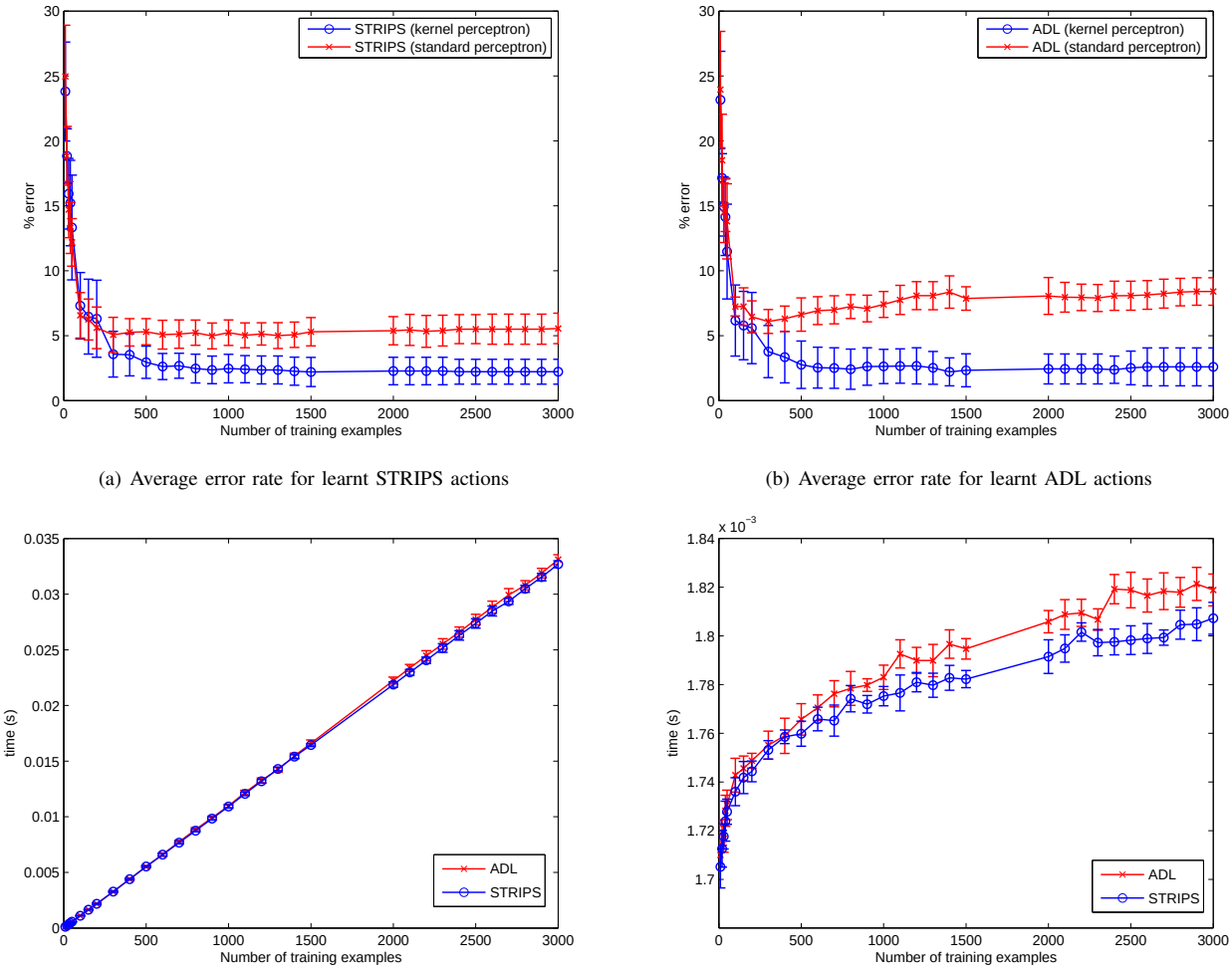
In our implementation, performing training or prediction consists of two steps: calculating the kernel matrix, followed by either the perceptron algorithm loop or the decision function calculation, respectively. The kernel matrix calculation does not vary with the difficulty of the problem. Calculating the kernel matrix for training is $O(n^2)$, where n is the number of training examples. However, only around 700 examples are required to achieve sufficient generalisation for planning in the test domain, corresponding to under 0.25 seconds to calculate the kernel matrix. Similarly for prediction, calculating the kernel matrix is $O(mn)$, where n is the number of training examples and m the number of testing examples. For 700 training examples and 500 testing examples the computation time is also below 0.25 seconds.

For the second step, estimates of $O(n^2)$ for training and $O(n)$ for prediction are valid for the worst case, where the kernel matrix entries for every pair of training examples have to be used in the perceptron algorithm loop (so every training example contributes to the weight vector), and where kernel matrix entries for every training example paired with every test example have to be used for the decision function calculation (again, when every training example contributes to the weight vector). The overall time is also affected by such factors as the number of bits in the input vector (which affects every calculations of the kernel matrix entries), the number of iterations the perceptron algorithm has to make (which affects training), and the number of training vectors which contribute to the weight vector (which affects both training and prediction). For the testing domain, however, we have almost linear training time and constant prediction time in the number of training examples.

V. Discussion

The results of our experiments show that kernel perceptrons are able to learn the dynamics of a planning domain. In order to test the feasibility of our approach under real planning conditions, we are currently integrating our learning mechanism with the PKS (Planning with Knowledge and Sensing) planner [21], [22]. In this case, the planner uses the network as part of its action model, by querying the network during plan construction to determine action effects at particular states. Since planning systems traditionally use efficient rule-based action models, we must still evaluate the extra overhead resulting from our network-based approach.

We also envision a more incremental approach to train and use our learning mechanism, as a component of a cognitive architecture of the kind reported in [12], where a low-level robot/vision system is linked to a high-level planning system.



(a) Average error rate for learnt STRIPS actions

(b) Average error rate for learnt ADL actions

(c) Training time for STRIPS and ADL actions

(d) Prediction time for STRIPS and ADL actions

Fig. 2. Results from experiments in the object manipulation domain

Rather than employing a completely offline training phase for our network, we foresee a more interactive approach, where online training data is generated from the robot's initial experiences in its environment, and the planner is able to use early action models to generate plans (albeit, lower quality plans). As more training data becomes available, plan quality should also increase, allowing the robot to learn how better to act. An existing integration project that uses the object manipulation scenario described in this paper, may provide a useful testbed for our learning techniques [13].

Testing in a robot/planning environment will also allow us to investigate the effectiveness of our learning method in practice. For instance, although the error rates for our learnt actions are low ($< 3\%$), it is unclear what effect this will have on the quality of plans constructed for this domain. Since replanning often has to take place in real-world robot domains, having a perfect plan is not always necessary. On the other hand, our method also assumes deterministic outcomes of actions, whereas some actions might better be modelled with probabilistic outcomes in this environment. In this case, we may again be able to use replanning techniques to some extent, but may also have to consider more substantial changes to our approach.

Currently, our approach makes certain assumptions that are not always realistic, especially when data is provided by real-world systems. For instance, we assume that there is no noise in the input or output data, and no irrelevant data in the training outcomes. We can relax the first assumption about noisy data, by using a noise-tolerant variant of the perceptron algorithm, such as adding a margin term [9]. We also believe such techniques can be used to handle irrelevant output data, since by definition such changes behave like noise. (In particular, irrelevant outputs correspond to irrelevant state changes in the action effects.) Otherwise, we run the risk of having perceptrons that fail to converge, or produce error-prone output when trying to predict such cases.

Representationally, there is also an issue with predicting changes to domain properties that are dependent on more primitive properties, as such changes can be wider-reaching than changes to the purely primitive properties themselves.

For example, our object manipulation scenario allows for situations where the robot can combine two existing stacks of objects into one large tower, by gripping the base object of one stack and releasing it at the top of another stack. In our initial planning representation, a predicate $\mathit{instack}(x, y)$ is used to indicate that a block x is in a stack with y at its base. Thus, after combining two stacks, *instack* must be updated for all objects in the "gripped" stack, to reflect the new base object of the single tower. In order to update *instack* correctly using our approach, the output would have to represent all of the objects in both stacks. Currently, only objects directly acted on, or related to directly acted on objects, are represented. Rather than attempt to represent all the objects required, it is easier to treat *instack* as a derived predicate defined in terms of more primitive properties (e.g., *isin* and *ontable*). This is also the approach taken in [5], where derived "concepts" are used in the rule antecedents but only primitive properties are used in the outcomes.

Additional testing is needed to determine the scalability of our approach on more difficult planning domains, although we expect that it should scale well with the number of predicates and actions. We also plan to compare our results to those of other classifiers, such as SVMs. We are currently investigating extensions to our approach to learn more comprehensive, and more complex, action models. For instance, we believe that our kernel perceptron approach could also be used to learn action preconditions, provided it is possible to only represent a small number of objects in the state at a time. Such an extension would require a means of choosing which objects to consider, and may ultimately need to be learnt. An attentional mechanism of some sort may be of help in this task [23], [5]. We also believe that our approach can be adapted to learn more sophisticated action representations, such as those used by PKS to describe knowledge and sensing. Since PKS's representation is based on an extended version of STRIPS/ADL, many of its features are similar to those that can already be learnt by our methods.

VI. Conclusions

In this paper we presented a mechanism based on kernel perceptrons, to address the problem of learning STRIPS and ADL action effects for planning domains. Overall, our approach demonstrated efficient performance on our testing sets, with low average error rates ($< 3\%$) for the learnt action effects. This work is also part of a larger cognitive architecture linking a high-level reasoning component with a low-level robot/vision system. We are currently in the process of integrating our learning method with the PKS planning system, and testing our approach on more complex planning domains including a real-world robot environment [13].

Acknowledgements

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

References

[1] 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)*, 1995, pp. 549–557.

[2] 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.

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

[4] M. R. Doğar, M. Çakmak, E. Uğur, and E. Şahin, "From primitive behaviors to goal directed behavior using affordances," in *Proc. of Intelligent Robots and Systems (IROS 2007)*, 2007.

[5] H. M. Pasula, L. S. Zettlemoyer, and L. P. Kaelbling, "Learning symbolic models of stochastic world domains," *Journal of Artificial Intelligence Research*, vol. 29, pp. 309–352, 2007.

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

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

[8] Y. Freund and R. Schapire, "Large margin classification using the perceptron algorithm," *Machine Learning*, vol. 37, pp. 277–296, 1999.

[9] R. Khardon and G. M. Wachman, "Noise tolerant variants of the perceptron algorithm," *Journal of Machine Learning Research*, vol. 8, pp. 227–248, 2007.

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

[11] E. P. D. Pednault, "ADL: Exploring the middle ground between STRIPS and the situation calculus," in *Proc. of Principles of Knowledge Representation and Reasoning (KR-89)*. Morgan Kaufmann Publishers, 1989, pp. 324–332.

[12] C. Geib, K. Mourão, R. Petrick, N. Pugeault, M. Steedman, N. Krueger, and F. Wörgötter, "Object action complexes as an interface for planning and robot control," in *Proceedings of the Humanoids-06 Workshop: Towards Cognitive Humanoid Robots*, Genoa, Italy, 2006.

[13] D. Kraft, E. Başeski, M. Popović, A. M. Batog, A. Kjær-Nielsen, N. Krüger, R. Petrick, C. Geib, N. Pugeault, M. Steedman, T. Asfour, R. Dillmann, S. Kalkan, F. Wörgötter, B. Hommel, R. Detry, and J. Piater, "Exploration and planning in a three-level cognitive architecture," in *Int. Conference on Cognitive Systems (CogSys 2008)*, 2008.

[14] M. Surdeanu and M. Ciaramita, "Robust information extraction with perceptrons," in *Proceedings of the NIST 2007 Automatic Content Extraction Workshop (ACE07)*, Mar. 2007.

[15] T. Graepel, R. Herbrich, and R. C. Williamson, "From margin to sparsity," *Advances in Neural Information Processing Systems*, vol. 13, pp. 210–216, 2000.

[16] F. Rosenblatt, "The perceptron: a probabilistic model for information storage and organization in the brain," *Psychological Review*, vol. 65, no. 6, pp. 386–408, Nov. 1958.

[17] A. B. Novikoff, "On convergence proofs for perceptrons," in *Proceedings of the Symposium on the Mathematical Theory of Automata*, vol. 12, 1963, pp. 615–622.

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

[19] K. Sadohara, "Learning of boolean functions using support vector machines," in *Proc. of Algorithmic Learning Theory,* Lecture Notes in Artificial Intelligence, vol. 2225. Springer, 2001, pp. 106–118.

[20] R. Khardon, D. Roth, and R. A. Servedio, "Efficiency versus convergence of boolean kernels for on-line learning algorithms," *Journal of Artificial Intelligence Research*, vol. 24, pp. 341–356, 2005.

[21] R. P. A. Petrick and F. Bacchus, "A knowledge-based approach to planning with incomplete information and sensing," in *Proc. of Artificial Intelligence Planning and Scheduling (AIPS-2002)*. AAAI Press, 2002, pp. 212–221.

[22] ——, "Extending the knowledge-based approach to planning with incomplete information and sensing," in *Proc. of Automated Planning and Scheduling (ICAPS-04)*. AAAI Press, 2004, pp. 2–11.

[23] D. Kragic, M. Björkman, H. I. Christensen, and J.-O. Eklundh, "Vision for robotic object manipulation in domestic settings," *Robotics and Autonomous Systems*, vol. 52, no. 1, pp. 85–100, July 2005.

Appendix C

Representation and Integration: Combining Robot Control, High-Level Planning, and Action Learning

Ronald Petrick[1], **Dirk Kraft**[2], **Kira Mourão**[1], **Christopher Geib**[1], **Nicolas Pugeault**[1,2], **Norbert Krüger**[2], and **Mark Steedman**[1]

Abstract. We describe an approach to integrated robot control, high-level planning, and action effect learning that attempts to overcome the representational difficulties that exist between these diverse areas. Our approach combines ideas from robot vision, knowledge-level planning, and connectionist machine learning, and focuses on the representational needs of these components. We also make use of a simple representational unit called an instantiated state transition fragment (ISTF) and a related structure called an object-action complex (OAC). The goal of this work is a general approach for inducing high-level action specifications, suitable for planning, from a robot's interactions with the world. We present a detailed overview of our approach and show how it supports the learning of certain aspects of a high-level representation from low-level world state information.

1 INTRODUCTION AND MOTIVATION

The problem of integrating low-level robot systems with high-level symbolic planners introduces significant representational difficulties that must first be overcome. Since the requirements for robot-level control and vision tend to be different from that of traditional planning, neither representation is usually sufficient to accommodate the needs of an integrated system. Overcoming these representational differences is a necessary challenge, however, since both levels seem to be required to produce human-like behaviour.

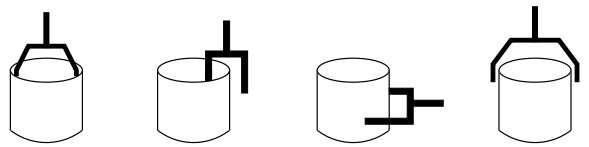
In general, robot systems tend to use representations based on vectors of continuous values, which denote 3D spatial coordinates, joint angles, force vectors, or other world-level features that require real-valued parameters [20]. Such representations allow system builders to succinctly specify robot behaviour since most of the computations required for low-level robot control are effectively captured as continuous transforms of continuous vectors over time. On the other hand, high-level planning systems typically use representations based on discrete, symbolic models of objects, properties, and actions, described in logical languages (e.g., [5, 23, 16, 27, 31]). Instead of modelling low-level continuous processes, these representations capture the dynamics of the world or the agent's knowledge at a more abstract level, for instance by characterizing the state changes that result from deliberate, planned action.

In this paper we describe an approach for integrating a robot/vision system with a high-level planner, that attempts to overcome the representational challenges described above. In particular, our approach gives rise to a system that is capable of automatically inducing certain aspects of a high-level representation suitable for planning, from the robot's interactions with the real world using basic "reflex" actions. This paper describes work currently in progress. As such, we do not address the entire problem of learning action representations, but instead focus on two important parts: object learning and action effect learning. Our approach uses a simple representational unit called an *instantiated state transition fragment (ISTF)* and a related structure called an *object-action complex (OAC)* [7], both of which arise naturally from the robot's interaction with the world—and world objects in particular. These notions also help us address certain control problems, for instance the relationship between high-level sensing actions and their execution by the robot, and representational issues that arise at different levels of our system. Although we only consider a portion of a larger learning problem, we are also interested in implementing these ideas within a framework that includes the lowest-level control mechanisms right up to the high-level reasoning components. Finally, we believe our approach is general and that these ideas can be successfully transferred to other robot platforms and planners, with capabilities other than those we describe here.

To illustrate our approach, we will consider a simple robot manipulation scenario throughout this paper. This domain consists of a robot with a gripper, a table with a number of objects on it, and a "shelf" (a special region of the table). The robot has a camera to view the objects in the world but does not initially have knowledge of those objects. Instead, world knowledge must be provided by the vision system, the robot's sensors, and the basic action reflexes built into the robot controller. The robot is given the task of clearing the objects from the table by placing them on the shelf. The shelf has limited space so the objects must be stacked in order for the robot to successfully complete the task. For simplicity, each object is assumed to be roughly cylindrical in shape and has a "radius" which provides an estimate of its size. An object A can be stacked into an object B provided the radius of A is less than that of B, and B is an "open" object. Unlike a standard blocks-world scenario, the robot will not have complete information about the state of the world. In particular, we will often consider scenarios where the robot does not know whether an object is open or not and must perform a test to determine an object's "openness". The robot will also have a choice of four different grasping types for manipulating objects in the world. Not all grasp types can be used on every object, and certain grasp types are further restricted by the position of an object relative to other objects in the world. The set of available grasp types is shown in Figure 1. Finally, actions in this domain can fail during execution and the robot's sensors may return noisy data.

The rest of the paper presents an overview of our approach from a representational point of view, and discusses the main components

[1] School of Informatics, University of Edinburgh,Edinburgh EH8 9LW, Scotland, United Kingdom, contact e-mail: *rpetrick@inf.ed.ac.uk*.
[2] The Maersk Mc-Kinney Moller Institute, University of Southern Denmark, DK-5230 Odense M, Denmark.



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

Figure 1. Available grasping types in the robot manipulation scenario

of our system. In Section 2 we describe the basic representations used in the paper. In Section 3 we discuss how object information is discovered from the robot/vision system's initial experiences in the world. In Section 4 we describe the high-level planner and plan execution monitor. In Section 5 we introduce a mechanism for learning the effects of actions from state descriptions. In Section 6 we discuss the current state of implementation in our system and some early empirical results. Finally, in Section 7 we discuss the advantages of our approach from a representation point of view, and describe some areas of future work.

2 BASIC REPRESENTATIONS

At the robot/vision level, the system has a set Σ of sensors, $\Sigma = \{\sigma_1, \sigma_2, \ldots, \sigma_n\}$, where each sensor σ_i returns an observation $obs(\sigma_i)$ about some aspect of the world. The execution of a robot-level *motor program* or robot action may cause changes to the world which can be observed through subsequent sensing. Each motor program is typically executed with respect to particular *objects* in the world. We assume that the robot/vision system does not initially know about any objects and, thus, can't execute many motor programs. Instead, the robot has a set of *basic reflex actions* that aren't dependent on particular objects and can be used for exploring the world initially.

The planning level representation is based on a set of *fluents*, $f_1, f_2, \ldots, f_m$: first-order predicates and functions that denote particular qualities of the world, robot, and objects. Fluents represent high-level (possibly abstract) counterparts of some of the properties the robot is capable of sensing. In particular, the value of a fluent is a function of the observations returned by the sensor set, i.e., $f_i = \Gamma_i(\Sigma)$. Typically, each fluent depends on a subset of the sensor observations and not every sensor need map to a fluent (some sensors are only used at the lower control level). Fluents can also be parameterized by high-level versions of the objects known at the robot level.

A *state* is a snapshot of the values of all instantiated fluents at some point during the execution of the system. States represent a point of intersection between the low-level and high-level representations, since states are *induced* from a set of sensor observations and the corresponding sensor/fluent mappings (i.e., the functions Γ_i). High-level *actions* represent abstract versions of some of the robot's motor programs. Since all actions must ultimately be executed by the robot, each action is decomposable to a fixed set of motor programs.[3] Thus, the robot's interaction with the world can be viewed as a simple state transition system: the robot's sensor observations give rise to a state description; executing an action brings about changes in the world that can be observed through subsequent sensing. More importantly, every interaction of this form provides the robot with an opportunity to observe a small portion of the world's state space, which we refer to as an *instantiated state transition fragment (ISTF)* [7].

Formally, an ISTF is a tuple $\langle s_i, mp_j, obj_{mp_j}, s_{i+1} \rangle$, where s_i is the state that is sensed before applying the motor program instance mp_j, obj_{mp_j} is the object that the motor program is defined relative to, and s_{i+1} is the state sensed after executing the motor program. Thus, an ISTF is a situated pairing of an object and an action that captures a small fragment of the world's state transition function. The states s_i and s_{i+1} contain snapshots of the robot's information about these states, some of which may be irrelevant to the action being applied.

We will also consider a second representational structure that results from generalising over instances of ISTFs. Such structures are referred to as *object-action complexes (OACs)* [7], and are similar to ISTFs, but contain only the relevant instantiated state information needed to predict the applicability of an action and its effects, with all irrelevant information pruned away. An OAC is defined by a tuple of the form $\langle S_i, MP_j, Obj_k, S_{i+1} \rangle$, where S_i and S_{i+1} are two states, MP_j is a set of motor programs, and Obj_k is a class of objects. In this case, S_i only describes those properties of the world state that are required to execute any of the motor programs in MP_j, when acting on an object of class Obj_k. S_{i+1} describes a world state which captures the properties changed by the motor program.

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

3 VISION-BASED OBJECT DISCOVERY

The robot system includes a vision component based on an *early cognitive vision* framework [15] which provides a scene representation composed of local 3D edge descriptors that outline the visible contours [26]. Because the system initially lacks knowledge of the objects that make up the scene, the visual representation is *unsegmented*: descriptors that belong to one object are not explicitly distinct from the ones that belong to other objects, or the background.

To aid in the discovery of new objects, the robot is equipped with a basic reflex action [1] that is elicited by specific visual feature combinations in the unsegmented world representation. The outcome of these reflexes allows the system to gather knowledge about the scene, which is used to segment the visual world into objects and identify basic affordances. We consider a reflex where the robot tries to grasp a planar surface in the scene. Each time the robot executes such a reflex, haptic information allows the system to evaluate the outcome: either the grasp was successful and the gripper is holding something, or it failed and the gripper simply closed. A failed attempt forces the system to reconsider its original assumptions, whereas a successful attempt confirms the feasibility of the reflex. Once a successful grasp is performed, the robot gains physical control over this part of the scene. If we assume that the full kinematics of the robot's arm are known (which is true for industrial robots), then it is possible to segment the grasped object from the rest of the visual world as it is the only part that moves synchronously with the robot's arm.

With physical control, the system visually inspects an object from

[3] We do not focus here on the problem of learning high-level action *schema* (i.e., the set of action names and their parameters) or the action/motor program mappings. Instead, we assume that the action schema are provided with the corresponding mappings to robot-level motor programs.

a variety of viewpoints and builds a 3D representation [13]. Features on the object are tracked over multiple frames, between which the object moves with a known motion. If features are constant over a series of frames they become included in the object's representation, otherwise they are assumed not to belong to the object. (A more detailed description of the accumulation process can be found in [13].) The final description is labelled and recorded as an identifier for a new object class, along with the successful reflex (now a motor program). Using this new knowledge, the system then reconsiders its interpretation of the scene: using a representation-specific pose estimation algorithm [3] all other instances of the same object class are identified and labelled. By repeating this process, the system constructs a representation of the world objects, as instances of symbolic classes that carry basic affordances, i.e., particular reflex actions that have been successfully applied to grasp objects of this class.

The technical implementation of the pose estimation algorithm has only recently become available. Prior to this, a circle detection algorithm was developed to recognise cylindrical objects, to which the domain was restricted for this work. Four grasp templates were used to define the primitive reflex actions in an object-centric way (where concrete grasps were generated based on the object pose). Although this approach negates the need for the general pose estimation algorithm, the conclusions drawn from experiments in this limited scenario are still easily transferable to the general case.

Figure 2 illustrates the "birth of an object." In (a), the dots on the image show the predicted structures. Both spurious primitives, parts of the background that are not confirmed by the image, and the confirmed predictions are shown. In (b), the shape model learned from the object in (a) is shown. In (c) and (d), two additional objects are shown along with their learned shape models. The "gap" in the shape models corresponds to where the robot's gripper held the objects.

The object-centric nature of the robot's world exploration process has immediate consequences for the high-level representation. First, newly discovered objects are reported to the planning level and added to its representation. At the planning level, objects are simply labels while the real-world object information is stored at the robot level. Such a representation means that we can avoid certain types of real-valued information at the high level (e.g., 3D location coordinates, object orientation vectors, etc.) and instead refer to objects by their labels (e.g., *obj1* may denote a particular red cup on the table). With the addition of new objects, the planning system can immediately start using such objects in its reasoning and plan construction. Since we assume that object names do not change over time, high-level object labels act as indices into the low-level object representation. Thus, plans with object references will be understandable to the robot/vision system. Finally, the successful identification of new objects will cause the robot/vision system to start sending regular state updates to the planning level about these objects and their properties. In particular, the ISTFs that result from subsequent interactions with the world will contain state information about these objects, provided they can be sensed by the robot. The planning level can then use this information for plan construction and plan execution monitoring. Additional details about the link between the robot/vision and planning systems are given in Section 6.

4 PLAN GENERATION AND MONITORING

The high-level planner is responsible for constructing plans that direct the behaviour of the robot in order to achieve a set of goals. For instance, in our example domain a plan might be constructed to clear all "open" objects from the table. Plans are built using PKS ("Planning with Knowledge and Sensing") [24, 25], a knowledge-level conditional planner that can operate with incomplete information and high-level sensing actions. Like other symbolic planners, PKS requires a goal, a description of the initial state, and a list of the available actions before it can construct plans. Unlike traditional approaches, PKS operates at the *knowledge level* by modelling the agent's knowledge state, rather than the world state. By doing so, PKS can reason efficiently about certain types of knowledge, and make effective use of non-propositional features, like functions, which often arise in real-world scenarios.

PKS is based on a generalization of STRIPS [5]. In STRIPS, a single database represents the world state. Actions update the database in a way that corresponds to their effects on the world. In PKS, the planner's knowledge state is represented by five databases, each of which stores a particular type of knowledge. Actions are described by the changes they make to the database set and, thus, to the planner's knowledge state. PKS also supports ADL-style conditional action effects [23].

Using PKS's representation language, we can formally model the example robot scenario by describing the objects, properties, and actions that make up the planning level domain. As we described above, objects at the planning level are simply labels that denote actual objects in the world identified by the robot/vision system.

High-level domain properties are defined by sets of logical fluents, i.e., predicates and functions that denote particular qualities of the world, robot, and objects. For instance, to model the example object manipulation scenario we include fluents such as:

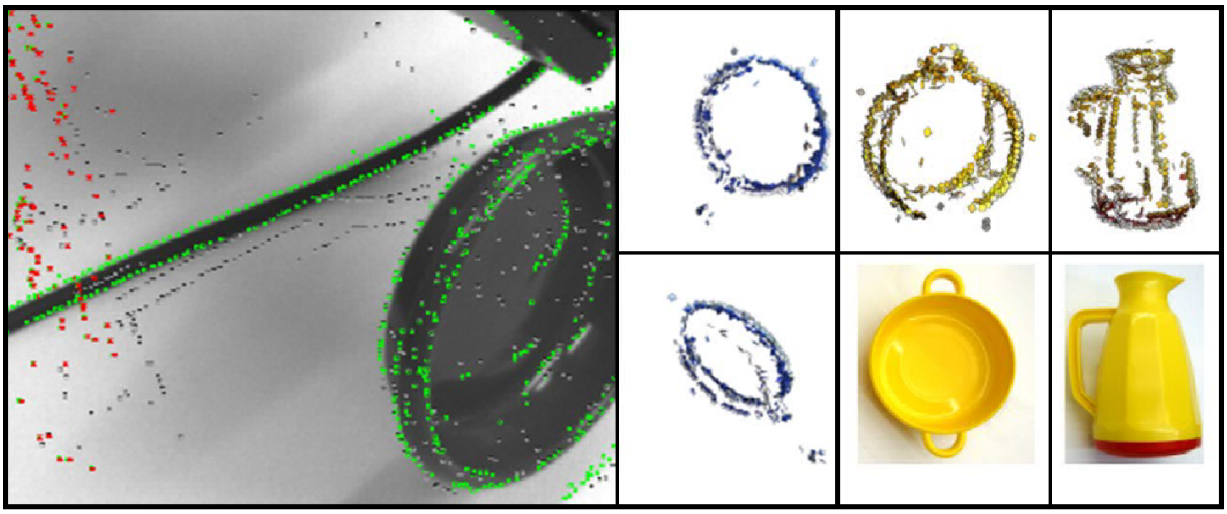
- *open*(x): object x is open,
- *gripperempty*: the robot's gripper is empty,
- *ingripper*(x): object x is in the gripper,
- *ontable*(x): object x is on the table,
- *isin*(x, y): object x is stacked in object y,
- *reachableX*(x): object x is reachable using grasp type X, and
- *radius*(x) = y: the radius of object x is y,

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

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

- *graspA-stack*(x): grasp object x from a stack using grasp type "A",
- *graspA-table*(x): grasp x from the table using grasp A,
- *putInto-object*(x, y): put object x into an object y on the table,
- *putAway*(x): put x away on the shelf, and
- *findout-open*(x): determine whether x is open or not,

among others. Some actions like "grasp A" are divided into two actions to account for different object configurations, however, the motor programs that implement these actions do not necessarily make such distinctions. Furthermore, the object-centric nature of the planning actions means that they do not require 3D coordinates, joint angles, or similar real values but, instead, include parameters that can be instantiated with specific objects. Actions also exist for other grasping options (B, C, and D) available at the robot level. Actions like *findout-open* are high-level *sensing actions* that direct the robot to gather information about the world state that is not normally provided to the planner as part of its regular state updates.



(a) (b) (c) (d)

Figure 2. Birth of the object

Table 1. Examples of PKS actions in the object manipulation domain

Action	Preconditions	Effects
graspA-table(x)	K(*clear*(x)) K(*gripperempty*) K(*ontable*(x)) K(*reachableA*(x)) K(*radius*(x) $\geq$ *minA*) K(*radius*(x) $\leq$ *maxA*)	*add*(K_f, *ingripper*(x)) *add*(K_f, $\neg$*gripperempty*) *add*(K_f, $\neg$*ontable*(x))
findout-open(x)	$\neg K_w$(*open*(x)) K(*ontable*(x))	*add*(K_w, *open*(x))

Actions in PKS are described by their preconditions and effects. An action's preconditions specify the domain properties that must hold for an action to be applied, while an action's effects encode the changes made to the domain properties as a result of executing the action. Table 1 shows two PKS actions from the example domain. Here, K_f refers to a database that models the planner's knowledge of simple facts, while K_w is a specialized database that stores the results of sensing actions that return binary information. An expression like $K(\phi)$ denotes a knowledge-level query that intuitively asks "does the planner know ϕ to be true?"

Given a goal, initial state description, and action list, the planner can build plans that are executable on the robot platform. We currently provide an interface that allows a human user to specify a high-level goal directly to the planning system. Initially, the planner does not know anything about the state of the world. After the robot/vision system performs its early exploration process and begins to produce ISTFs, an initial state description is generated and supplied to the planner automatically with information about newly discovered objects and their sensed properties, described in terms of the high-level fluents. Since PKS can model an agent's incomplete knowledge, the predicate and function instances in the initial state are treated as *known* state information, with all other state information considered to be unknown. We currently assume that the action schema are supplied to the planner as input, as are the mappings from high-level actions to low-level robot motor programs. (In Section 5 we consider how high-level action effects can be learned directly from state information.)

For instance, if we consider the situation in the example domain where two unstacked and open objects *obj1* and *obj2* are on a table, the planner can construct a simple plan using the above domain encoding to achieve the goal of clearing the table:

$$\begin{array}{l}[\textit{graspD-table}(\textit{obj1}),\\ \ \textit{putInto-object}(\textit{obj1}, \textit{obj2}),\\ \ \textit{graspB-table}(\textit{obj2}),\\ \ \textit{putAway}(\textit{obj2})\ ].\end{array} \quad (1)$$

In this plan, *obj1* is grasped from the table and put it into *obj2*, before the stacked objects are grasped and removed to the shelf.

The planner can also build more complex plans by including sensing actions. For instance, if the planner is given the goal of removing the "open" objects from the table, but does not know whether *obj1* is open or not, then it can construct the conditional plan:

$$\begin{array}{l}[\textit{findout-open}(\textit{obj1}),\\ \ \textit{branch}(\textit{open}(\textit{obj1}))\\ \ K^+ :\\ \qquad \textit{graspA-table}(\textit{obj1}),\\ \qquad \textit{putAway}(\textit{obj1})\\ \ K^- :\\ \qquad \textit{nil}\ ].\end{array}$$

This plan senses the truth value of *open*(*obj1*) and reasons about the possible outcomes of this action by including branches in the plan: if *open*(*obj1*) is true (the K^+ branch) then *obj1* is grasped and put away; if *open*(*obj1*) is false (the K^- branch) then no further action is taken.

To execute plans, the planning level interacts with the robot/vision system. Actions are fed to the robot one at a time, where they are converted into motor programs and executed in the world. A stream of ISTFs is also generated, arising from the motor programs being executed. Upon action completion the robot/vision level informs the planner as to any world state changes (the final state of the last ISTF).

An essential component in this architecture is the *plan execution monitor*, which assesses action failure and unexpected state information to control replanning and resensing activities. In particular, the difference between predicted and actual state information is used to decide between (i) continuing the execution of an existing plan, (ii)

asking the vision system to resense a portion of a scene at a higher resolution (in the hope of producing a more detailed state report), and (iii) replanning from the unexpected state using the current state report as a new initial planning state. The plan execution monitor also has the important task of managing the execution of plans with conditional branches, resulting from the inclusion of sensing actions.

When a high-level sensing action is executed at the robot level, the results of the sensing are made available to the robot/vision system in a subsequent ISTF, and passed to the planner as part of a state update. In our example domain, sensing actions like *findout-open* allow the robot to use its lower-level object information to make more informed decisions as to how such actions should best be executed (e.g., for *findout-open* the robot could "poke" an object to determine its openness). The plan execution monitor uses the returned information to decide which branch of a plan it should follow, and feeds the correct sequence of actions to the lower levels. If such information is unavailable, resensing or replanning is triggered as above.

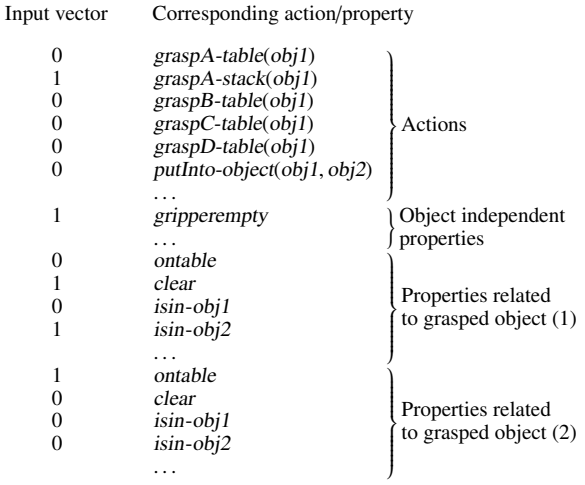


Figure 3. A binary input vector to the learning mechanism

5 LEARNING ACTION REPRESENTATIONS

The planner is capable of constructing plans that direct the robot's actions, in contrast to the reflex-based exploration of the world that the robot must initially perform. This shift from undirected to directed behaviour relies on an action specification that encodes the dynamics of the world in which the robot operates. While we have described how the robot/vision system is capable of generating ISTFs, the state information encoded in such fragments contains information that is both relevant and irrelevant to an action specification. The domain information required for planning actions, however, is more like the information found in a set of OACs, i.e., a generalization of the information in a set of ISTFs. Thus, presented with enough examples of such state transitions, a learning procedure should be able to filter out the irrelevant information and identify the necessary state information required for OACs and planning operators.

Using machine learning techniques to learn action specifications is not a new idea, and prior approaches have addressed this problem using a variety of techniques. For instance, inductive learning [32] and directed experimentation [8] have been applied to data represented in first-order logic, without noise or non-determinism. Other approaches have used schema learning to learn probabilistic action rules operating on discrete-valued sensor data [9]. Also, k-means clustering of equivalence classes, followed by extraction of sensor data features, has been used to train support vector machines (SVMs) to predict deterministic action effects in a given context [4]. [18] proposes a method of modelling actions by learning control laws that change individual perceptual features of the robot's world. Recently, attention has shifted to methods which exploit relational structure in order to improve speed and generalisation performance. [22] generates and refines rules using heuristic search, and shows that relational deictic rules are learnt more effectively than propositional or purely relational rules. [30] uses a logical inference algorithm to efficiently learn rules in relational environments.

Our approach is based on a connectionist machine learning model, namely *kernel perceptron learning* [2, 6]. This approach is particularly useful for our task since it can be shown to provide good performance, both in terms of training time and the quality of the models that are learnt, making it an attractive choice for practical systems.

Learning the complete dynamics of a planning domain requires the ability to learn both action preconditions and effects. Currently, our learning mechanism only addresses the problem of learning action effects, and the action schema and preconditions are supplied as input. Since an action's effects determine the changes made to a state during execution, the problem reduces to learning particular mappings between states. Furthermore, our current mechanism can only learn standard STRIPS and ADL action effects, and is restricted to relational state properties (i.e., no sensing actions or functions).

The input to the learning mechanism uses a vector representation that encodes a description of the action being performed and the state at which the action is applied. For each available action the vector includes an element that is set to 1 if the action is to be performed, or 0 otherwise. For states, we consider object-independent and object-dependent properties separately. In the case of object-independent properties (e.g., *gripperempty*), the vector includes an element for each property, representing its truth value at the state being considered (1 = true, 0 = false). For object-dependent properties we consider each property on a per object basis, and represent only those properties of the objects directly involved in the action being applied, and the objects related in some way to those objects. A form of deictic representation is used (similar to [22]), where objects are specified in terms of their roles in the action, or their roles in a property. Instead of maintaining a "slot" in the input vector for each possible role, roles are allowed to overlap. Thus, each object is represented by a set of inputs, one for each object-specific predicate (e.g., *ingripper*), and each relation with another object (e.g., *isin*). To bind relations to the correct objects, extra predicates are used *isin-obj1*, *isin-obj2*, etc.). This representation significantly reduces the number of inputs since its size is dependent on the actions and relations between objects, rather than the absolute number of objects in the world.

Overall, the input vector has the form: ⟨*actions, object-independent properties, object slot 1 predicates, object slot 2 predicates, ..., object slot n predicates*⟩. Figure 3 shows one such input vector for an action-state pair. In this case, the action performed is *graspA-stack*. The "grasped object" properties are represented in the object *obj1* slot, while the "object below the grasped object" properties are represented in the object *obj2* slot. Here, *gripperempty*, *clear*(*obj1*), *isin*(*obj1*, *obj2*) and *ontable*(*obj2*) are true in the state, since the corresponding bits are set to 1; all other bits are set to 0.

The output of the learning mechanism is a prediction of the properties that will change when the action is performed. The output is also encoded as a binary vector, with each bit representing one property of the state: the output value is 1 if the property changes and 0 if it does not. As with the input vector, object-independent properties

are represented by single elements, and object-specific properties are represented on a per-object basis in slots. Overall, the output vector has the form: ⟨*object-independent properties, object slot 1 predicates, object slot 2 predicates, . . . , object slot n predicates*⟩.

Using the above representation, the learning mechanism is tasked with finding the association between action-precondition pairs and their effects, i.e., rules of the form $\langle A, Pre_A \rangle \rightarrow Eff_A$. Currently, we have focused on learning the effects of standard STRIPS and ADL planning actions. Thus, all action effects involve either conjunctions of predicates (in the case of STRIPS) or conjunctions of predicates conditioned on other conjunctions of predicates (in the case of ADL). As a result, it is sufficient to learn a rule for each effect predicate separately and we can treat the learning problem as a set of binary classification problems, one for each (conditional) effect predicate.

A classifier that is both simple and fast is the *perceptron* [28]. The perceptron maintains a weight vector $\mathbf{w}$ which is adjusted at each training step. The i-th input vector $\mathbf{x}_i \in \{0,1\}^n$ in a class $y \in \{-1, 1\}$ is classified by the perceptron using the decision function $f(\mathbf{x}_i) = sgn(\langle \mathbf{w} \cdot \mathbf{x}_i \rangle)$. If $f(\mathbf{x}_i)$ is not the correct class then $\mathbf{w}$ is set to $\mathbf{w} + y\mathbf{x}$; if $f(\mathbf{x}_i)$ is correct then $\mathbf{w}$ is left unchanged. Provided the data is linearly separable, the perceptron algorithm is guaranteed to converge on a solution in a finite number of steps [21, 17]. Otherwise, the algorithm oscillates, changing $\mathbf{w}$ at each misclassified input vector.

Since the problem of learning action effects is not linearly separable in general, we adapt the perceptron algorithm by applying the *kernel trick* [6]. By doing so, we implicitly map the input feature space into a higher-dimensional space where the data is linearly separable. Since the mapping is implicit, we avoid a massive expansion in the number of features, which may make the problem computationally infeasible. The kernel trick is applied by rewriting the decision function in terms of the dot product of the input vectors:

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

where α_j is the number of times the j-th example has been misclassified by the perceptron. By replacing the dot product with a *kernel function* $k(\mathbf{x}_i, \mathbf{x}_j)$ which calculates $\langle \phi(\mathbf{x}_i) \cdot \phi(\mathbf{x}_j) \rangle$ for some mapping ϕ, the perceptron algorithm can be run in higher dimensional space without requiring the mapping to be explicitly calculated. An ideal kernel is one which allows the perceptron algorithm to run over the feature space of all conjunctions of features in the original input space, allowing an accurate representation of the exact conjunction of features (action and preconditions) corresponding to a particular effect. In our case, the kernel $k(x, y) = 2^{same(x,y)}$ is used, where $same(x, y)$ is the number of bits with the same value in both x and y [29, 10]. (See [19] for a more detailed discussion of this approach.)

6 INTEGRATION AND EMPIRICAL RESULTS

In this section we consider two separate interactions between the components described above, forming part of the larger system we are in the process of implementing (see Figure 4). In Section 6.1 we consider the link between the planning level and the robot/vision level, and the execution of high-level plans on the robot platform. In Section 6.2 we focus on the learning mechanism and the actions that arise from the object manipulation scenario. Certain aspects of our system, such as the plan execution monitor and the inclusion of the learning mechanism within the larger system, are currently under development and have not yet been fully implemented. The robot/vision system forming the basis of our implementation consists of an industrial 6 degrees of freedom robot with a two finger grasper, a high resolution stereo camera system, and a haptic force-torque sensor mounted between the robot and grasper, providing the measurement of forces at the wrist.

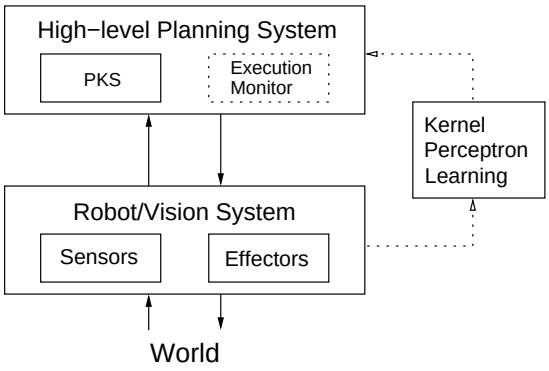


Figure 4. System components and proposed/current interactions

6.1 Linking high-level plan generation with robot/vision-level plan execution

From an integration point of view, the robot/vision system is currently linked directly to the planning level and we are experimenting with plan generation and execution. Since the planner is not able to handle raw sensor data as a state description, the low-level ISTFs generated by the robot/vision system must be abstracted into a language that is understandable by the planner. As a result, sensor data is "wrapped" and reported to the planner in the form of "symbolic" ISTFs with state representations that include predicates and functions. Since our present focus is on object and action learning, we have simply hard-coded the mappings between certain sensor combinations and the corresponding high-level properties.

For instance, some of the predicates used in the example manipulation domain are computed as follows:

- *ingripper*, *gripperempty*: Initially the gripper is empty and the predicate *gripperempty* is formed. As soon as the robot grasps an object, and confirms that the grasp is successful by means of the gripper not closing up to mechanical limits, the system knows that it has the object in its hand and can form a predicate *ingripper*(*objX*), using its visual information about discovered objects to identify the label *objX* corresponding to the object in the gripper. A negated predicate ¬*gripperempty* is also generated, as are negated *ingripper* instances for objects not in the gripper. Releasing the object returns the gripper to an empty state again.
- *reachableX*: Based on the position of a circle forming the top of a cylindrical object in the scene, as returned by the circle detection algorithm, we can compute possible grasp positions (for the different grasp types) for each object. Using standard robotics path planning methods we can then compute if there is a collision-free path between the start position and the pose the gripper needs to reach the object for a particular grasp.
- *isin*, *clear*, *instack*: These three predicates are computed based on geometric reasoning. Since the object height is not known we can only use the x, y-plane information. Furthermore the fact that objects with a bigger radius are lower in the stack is assumed. Objects whose centres (in the x, y-plane) are closer than 40mm are selected as stack candidates. The sorted stack candidates can then be checked for real inclusion using the circle centres and radii.

- *open*: We do not assume that all objects in the world are "open." Unlike the previous properties which can be determined directly from ordinary sensor data, the robot must first perform an explicit test in order to determine an object's openness. In this case, the robot gripper is used to "poke" inside the potential opening of an object. If the robot encounters a collision where forces acting upon the gripper are above a certain threshold, then the object is assumed to be closed. Otherwise, we assume the object is open. (We also envision a second, purely visual test for openness using dense stereo, but this approach has not yet been implemented.)

After an initial exploration of its environment, the robot/vision system provides the planner with a report of the current set of objects it believes to be in the world, along with a (possibly partial) state report of the sensed properties of those objects. Using this information as its initial knowledge state, and the high-level action specification described in Section 4, the planner attempts to construct a plan to achieve the goal of clearing a set of objects from the table.

Once a plan has been generated, it is passed to the plan execution monitor which sends actions to the robot/vision level one step at a time. At the robot level, a high-level action is decomposed into a set of motor programs which are then executed by the robot in the world. Currently, the mapping of actions to motor programs is preprogrammed and supplied as part of the input to the system. During the execution of low-level motor programs, a stream of ISTFs is generated and recorded by the robot/vision system. After an action has been executed its success or failure is reported back to the plan execution monitor, along with a new report on the state of the world (the final state of the last ISTF). In our current implementation, the plan execution monitor simply terminates the execution of a plan if it encounters an unexpected state property, or a reported failure of an action. Otherwise, it sends the next action to the robot for execution. (No replanning or focused resensing is performed.) For instance, Figure 5 shows the robot executing the four-step plan described in (1) of Section 4 for clearing the table. (The "shelf" in this case is a special area at the side of the table.)

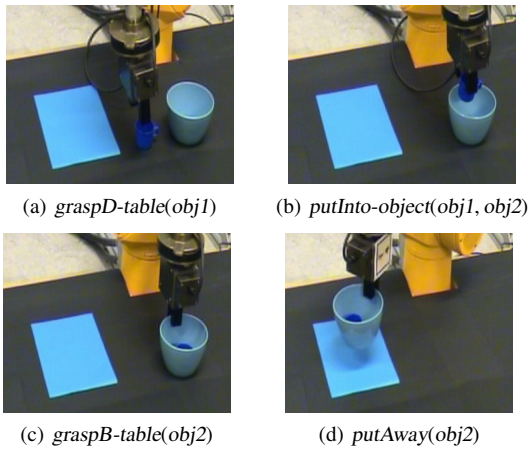
When a conditional plan with sensing actions is executed, the plan execution monitor sends *findout-open* actions to the robot/vision level like any other action. At the robot level, such an action is executed as the specific "poke" test described above to determine an object's openness. The results of this test are returned to the plan execution monitor as part of the updated state report. The monitor then uses this information to determine which branch of the conditional plan it should follow. From the point of view of the robot, it will only receive a sequential stream of actions and will be unaware of the conditional nature of the plan being executed. Figure 6 shows the robot testing the openness of two objects after receiving a sensing action from the planning level. In (a), the test fails since the object is not open; in (b) the test succeeds and the object is assumed to be open.

6.2 Learning STRIPS and ADL action effects in the object manipulation domain

Separate from the above robot/vision-planner integration, we established a preliminary link between the action effect learning mechanism and the planner. In particular, we applied our learning procedure to learn the effects of STRIPS and ADL planning actions, using data simulated from the example object manipulation domain.

The learning mechanism was evaluated using data similar to the ISTFs the robot/vision system is capable of producing. Both STRIPS and ADL versions of the high-level actions were considered. (For example, the two actions *graspA-stack* and *graspA-table* described in Section 4 were merged into a single ADL action, along with other changes.) Sensing actions and references to functional fluents were ignored. Two data sets were constructed to train and test the learning mechanism. Individual input vector instances were generated by randomly selecting an action, and setting the inputs for the preconditions required for the action to 1. The action input was set to 1, and all other action inputs to 0. The remaining input bits were used to create the two separate data sets. For the training data, half of the inputs in each instance were randomly set to 0 or 1, with the other half all set to 0 (vice versa for the testing data). Outputs were set to 1 if a state property changed as a result of the action and 0 if not. Thus, the data used to train the learning mechanism incorporated the (strong) assumptions that (i) all the necessary precondition information for an associated action was included as part of an input vector, and (ii) no spurious state changes was represented as part of an output vector. Noise was introduced in the irrelevant bits of the input vector, however, only relevant changes were included in the corresponding output vector.

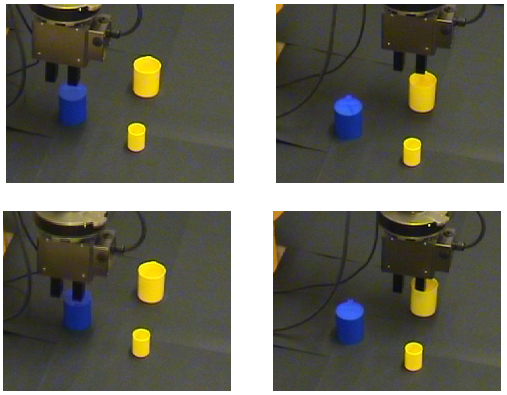
The learning mechanism was evaluated over multiple test runs using 3000 training and 500 testing examples. To determine an error bound on our results, 10 runs with different randomly generated training and testing sets were used. (All testing was done on a 2.4 GHz quad-core system with 6 Gb of RAM. All times were measured for Matlab 7.2.0.294.) The results of our testing are shown in Fig-



(a) *graspD-table*(*obj1*) (b) *putInto-object*(*obj1*, *obj2*)

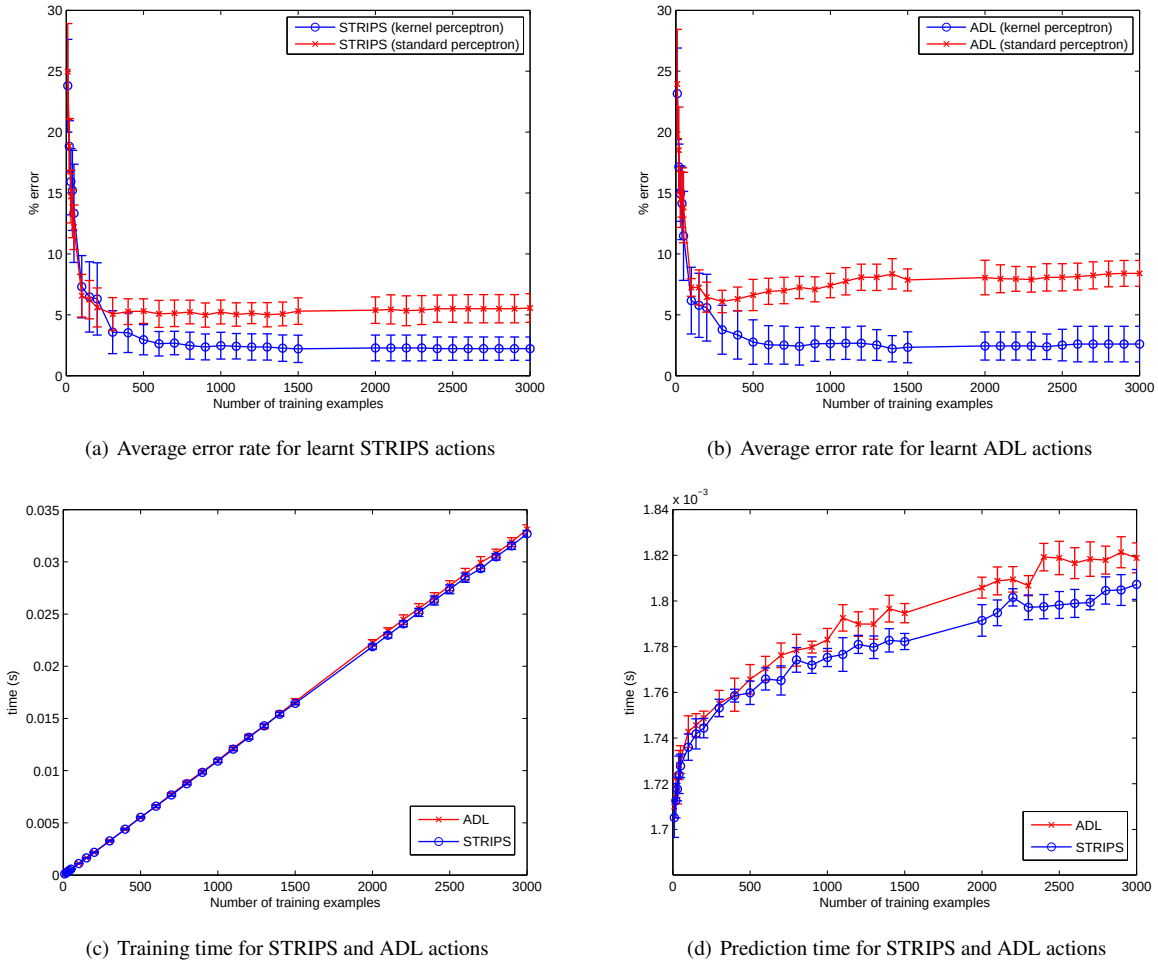
(c) *graspB-table*(*obj2*) (d) *putAway*(*obj2*)

Figure 5. Executing a high-level plan to clear a table



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

Figure 6. Testing the openness of an object



(a) Average error rate for learnt STRIPS actions

(b) Average error rate for learnt ADL actions

(c) Training time for STRIPS and ADL actions

(d) Prediction time for STRIPS and ADL actions

Figure 7. Results from experiments in the object manipulation domain (from [19])

ure 7. (A more detailed analysis of these results and description of our implementation can be found in [19].) In (a), the error rate for the learnt STRIPS actions is shown, while (b) shows the error rate for the ADL actions. In both cases, the average error dropped to less than 3% after 700 training examples. The standard perceptron error rate, included for comparison, shows significantly worse performance: over 5% error after 3000 training examples. In (c), the training time for both STRIPS and ADL actions is shown (for 1 bit of the effect vector), while (d) shows the prediction time (for 1 bit of 1 prediction).

In practical terms, the learning mechanism was quite efficient, requiring 0.035 seconds to train the system on 3000 examples and 1.84 $\times 10^{-3}$ seconds to test the system per output. For our particular example domain, there was little difference between the training and prediction times of STRIPS actions, compared with those for ADL actions. (In general we expect performance on ADL domains will always take longer than STRIPS domains, particularly when the conditional effect training examples are very dissimilar to the other training examples available.) Overall, the learning mechanism was able to effectively abstract away the irrelevant information from the ISTFs to produce a high-quality model of the action effects suitable for planning (at least for our current example domain).

7 DISCUSSION

From a representational point of view we have argued that ISTFs and OACs, grounded from actions performed at the robot level, can be viewed as the representational unit that underlies higher-level representations of objects, properties, and actions ("representation through integration"). As the low-level robot/vision system explores the world, successful actions produce ISTFs; on the basis of multiple experiences of particular ISTFs, OACs and high-level action models can be learned. Although some aspects of our approach are currently hard-coded (e.g., the action/motor program mappings), our learning mechanisms are nevertheless able to abstract away from "irrelevant" state information in the ISTFs to learn certain high-level OAC relationships from the robot's interactions with the world.

The resulting representations also enable interesting interactions between the components of the system ("integration through representation"). For instance, the planner can ignore some details about the execution of actions at the robot level (e.g., sensing actions like *findout-open*) and can avoid making certain commitments that are better left to the robot level (e.g., planning-level grasping actions are unaware of low-level properties like object location, gripper orientation relative to an object, etc.). Thus, we do not try to control all

aspects of robot behaviour at the planning level, but apply the planner's strengths to problems it can more readily solve. (For instance, PKS does not perform path planning but is more proficient at planning information-gathering operations.) As future work we are extending these ideas, for instance to allow the robot/vision system to choose between a set of possible tests to perform when executing a *findout-open* sensing action, while leaving the planning-level action specification unchanged.

We have focused on two particular learning problems in this work: object learning and action effect learning. As a result, we have avoided addressing other learning problems (e.g., learning the low-level sensor combinations that lead to particular high-level properties, or the mapping of high-level actions to low-level motor programs), which we leave to future work. Our focus on an implementation "from the world level to the knowledge level," however, provides us with a suitable testing framework for investigating such learning challenges as well as new planning contexts. Moreover, we are also interested in using this platform to explore other high-level learning tasks such as language acquisition.

We must also improve the scalability of our approach and overcome certain assumptions that are not realistic in real-world robotic systems. For instance, the learning mechanism has mainly been tested using state descriptions that are more "complete" than the ISTFs the robot/vision system is likely to produce. One way we can adapt our approach is by using a noise-tolerant variant of the perceptron algorithm, such as adding a margin term [11]. We also believe these techniques can be applied to irrelevant output data (i.e., irrelevant state changes in the action effects), since such changes behave like noise. Additional work is needed to extend our approach to more complex action representations, notably sensing actions and functions. We also believe our approach can be extended to learn action preconditions, provided it is possible to only represent a small number of objects in the state at a time. An attentional mechanism of some sort may be of help in this task [14]. Finally, although we have tested our learning mechanism on simulated data from the same domain used for the robot/vision-planner experiments, we are also aiming to test our learning mechanism with online data generated directly from the robot/vision system. Additional work is also needed to complete the remaining components of our system, most notably the plan execution monitor.

Our approach for integrating a robot/vision system with a high-level planner and action learning mechanism combines ideas from robot vision, symbolic knowledge representation and planning, and connectionist machine learning. The current state of our work highlights some significant interactions between the specific components of our system, however, we believe our approach is much more general and can be applied to other robot platforms and planners. (For instance, we have recently begun work to test some of our components and specifications on a humanoid robot platform.) The components we describe in this paper form part of a larger project called PACO-PLUS[4] investigating perception, action, and cognition—combining robot platforms with high-level representation and reasoning based on formal models of knowledge and action [12].

ACKNOWLEDGEMENTS

The work in this paper was partially funded by the European Commission through the PACO-PLUS project (FP6-2004-IST-4-27657).

[4] See `www.paco-plus.org` for more information about this project.

REFERENCES

[1] D. Aarno, J. Sommerfeld, D. Kragic, N. Pugeault, S. Kalkan, F. Wörgötter, D. Kraft, and N. Krüger, 'Early reactive grasping with second order 3D feature relations', in *Recent Progress in Robotics; Viable Robotic Service to Human, Selected papers from ICAR'07*, eds., S. Lee, I. Hong Suh, and M. Sang Kim, LNCIS Series, Springer-Verlag, (2007).

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

[3] Renaud Detry, Nicolas Pugeault, and Justus H. Piater, 'Probabilistic pose recovery using learned hierarchical object models', in *International Cognitive Vision Workshop (Workshop at the 6th International Conference on Vision Systems)*, (2008).

[4] 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).

[5] Richard E. Fikes and Nils J. Nilsson, 'STRIPS: A new approach to the application of theorem proving to problem solving', *Artificial Intelligence*, **2**, 189–208, (1971).

[6] Yoav Freund and Robert Schapire, 'Large margin classification using the perceptron algorithm', *Machine Learning*, **37**, 277–296, (1999).

[7] Christopher Geib, Kira Mourão, Ron Petrick, Nico Pugeault, Mark Steedman, Norbert Krueger, and Florentin Wörgötter, 'Object action complexes as an interface for planning and robot control', in *IEEE-RAS Humanoids-06 Workshop: Towards Cognitive Humanoid Robots*, (2006).

[8] Yolanda Gil, 'Learning by experimentation: Incremental refinement of incomplete planning domains', in *Proc. of ICML-94*. MIT Press, (1994).

[9] Michael Holmes and Charles Isbell, 'Schema learning: Experience-based construction of predictive action models', in *Advances in Neural Information Processing Systems (NIPS) 17*, pp. 585–562, (2005).

[10] Roni Khardon, Dan Roth, and Rocco A. Servedio, 'Efficiency versus convergence of boolean kernels for on-line learning algorithms', *Journal of Artificial Intelligence Research*, **24**, 341–356, (2005).

[11] Roni Khardon and Gabriel M. Wachman, 'Noise tolerant variants of the perceptron algorithm', *Journal of Machine Learning Research*, **8**, 227–248, (2007).

[12] D. Kraft, E. Başeski, M. Popović, A. M. Batog, A. Kjær-Nielsen, N. Krüger, R. Petrick, C. Geib, N. Pugeault, M. Steedman, T. Asfour, R. Dillmann, S. Kalkan, F. Wörgötter, B. Hommel, R. Detry, and J. Piater, 'Exploration and planning in a three-level cognitive architecture', in *Int. Conference on Cognitive Systems (CogSys 2008)*, (2008).

[13] D. Kraft, N. Pugeault, E. Başeski, M. Popović, D. Kragic, S. Kalkan, F. Wörgötter, and N. Krüger, 'Birth of the Object: Detection of objectness and extraction of object shape through object action complexes', *Special Issue on "Cognitive Humanoid Robots" of the International Journal of Humanoid Robotics*, (2008). (accepted).

[14] Danica Kragic, Måarten Björkman, Henrik I. Christensen, and Jan-Olof Eklundh, 'Vision for robotic object manipulation in domestic settings', *Robotics and Autonomous Systems*, **52**(1), 85–100, (July 2005).

[15] N. Krüger, M. Van Hulle, and F. Wörgötter, 'ECOVISION: Challenges in early-cognitive vision', *Int. Journal of Computer Vision*, (2006).

[16] John McCarthy and Patrick J. Hayes, 'Some philosophical problems from the standpoint of artificial intelligence', *Machine Intelligence*, **4**, 463–502, (1969).

[17] Marvin L. Minsky and Seymour A. Papert, *Perceptrons*, The MIT Press, December 1969.

[18] Joseph Modayil and Benjamin Kuipers, 'Where do actions come from? Autonomous robot learning of objects and actions', in *AAAI Spring Symposium Series*, (2007).

[19] Kira Mourão, Ronald P. A. Petrick, and Mark Steedman, 'Using kernel perceptrons to learn action effects for planning', in *International Conference on Cognitive Systems (CogSys 2008)*, (2008).

[20] R.M. Murray, Z. Li, and S.S. Sastry, *A mathematical introduction to Robotic Manipulation*, CRC Press, 1994.

[21] Albert B. Novikoff, 'On convergence proofs for perceptrons', in *Proceedings of the Symposium on the Mathematical Theory of Automata*, volume 12, pp. 615–622, (1963).

[22] Hanna M. Pasula, Luke S. Zettlemoyer, and Leslie P. Kaelbling, 'Learning symbolic models of stochastic world domains', *JAIR*, **29**, 309–352, (2007).

[23] Edwin P. D. Pednault, 'ADL: Exploring the middle ground between STRIPS and the situation calculus', in *Proc. of KR-89*, pp. 324–332. Morgan Kaufmann, (1989).
[24] Ronald P. A. Petrick and Fahiem Bacchus, 'A knowledge-based approach to planning with incomplete information and sensing', in *Proc. of AIPS-2002*, pp. 212–221. AAAI Press, (2002).
[25] Ronald P. A. Petrick and Fahiem Bacchus, 'Extending the knowledge-based approach to planning with incomplete information and sensing', in *Proc. of ICAPS-04*, pp. 2–11. AAAI Press, (2004).
[26] N. Pugeault, F. Wörgötter, and N. Krüger, 'Multi-modal scene reconstruction using perceptual grouping constraints', in *Proc. of the IEEE Computer Society Workshop on Perceptual Organization in Computer Vision*, (2006).
[27] Raymond Reiter, *Knowledge In Action: Logical Foundations for Specifying and Implementing Dynamical Systems*, MIT Press, 2001.
[28] Frank Rosenblatt, 'The perceptron: a probabilistic model for information storage and organization in the brain', *Psychological Review*, **65**(6), 386–408, (November 1958).
[29] Ken Sadohara, 'Learning of boolean functions using support vector machines', in *Proc. of Algorithmic Learning Theory,* Lecture Notes in Artificial Intelligence, volume 2225, pp. 106–118. Springer, (2001).
[30] Dafna Shahaf and Eyal Amir, 'Learning partially observable action schemas', in *Proc. of AAAI-06*. AAAI Press, (2006).
[31] Mark Steedman, 'Plans, affordances, and combinatory grammar', *Linguistics and Philosophy*, **25**, 723–753, (2002).
[32] Xuemei Wang, 'Learning by observation and practice: An incremental approach for planning operator acquisition', in *Proc. of ICML-95*, pp. 549–557, (1995).