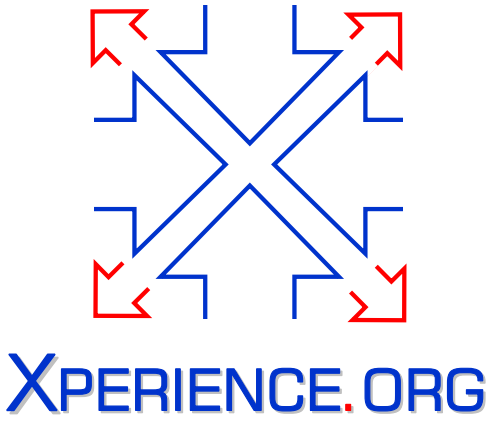
Project Acronym:	Xperience
Project Type:	IP
Project Title:	Robots Bootstrapped through Learning from Experience
Contract Number:	270273
Starting Date:	01-01-2011
Ending Date:	31-12-2015



Deliverable Number:	D3.2.3
Deliverable Title :	Structural Bootstrapping for Planning (III): Extended Reasoning and Indexical Information
Type (Internal, Restricted, Public):	PU
Authors:	Ron Petrick, Kira Mourão, Aris Valtazanos, and Mark Steedman
Contributing Partners:	UEDIN

Contractual Date of Delivery to the EC:	31-01-2014
Actual Date of Delivery to the EC:	28-02-2014

Contents

Executive Summary

A central contribution of WP3 (Generative Mechanisms), and WP3.2 (Structural Bootstrapping for Planning) in particular, is to extend the capabilities of current high-level planning models by applying structural bootstrapping to the knowledge-rich representation of actions and plans, to provide the apparatus needed to support plan generation and execution in low-level robotics domains (WP2; Outside In: Development and Representations) and higher-level domains requiring language and communication (WP4; Interaction and Communication). To this end, we describe the work by UEDIN on high-level planning techniques during the last work period, with a focus on Task 3.2.4 (Extended reasoning about object and indexical knowledge) and Task 3.2.3 (Plan structure and execution). This deliverable also reports on contributions related to Task 3.2.2 (Learning knowledge-level control rules) and Task 2.3.2 (Learning high-level action descriptions (rule learning)), and is related to the project-wide integration and demonstrations of WP5 (System Integration). Four papers are attached to this deliverable ([Pet14, Pet13, MP13, Val14]), which provide details of these contributions, as highlighted below.

The ability to reason and plan is essential for an intelligent agent acting in a dynamic and incompletely known world, such as the robot scenarios that are explored in WP5. High-level planning capabilities in Xperience are (partly) supplied by the PKS planner [9, 10], which UEDIN is extending for use in robotic and linguistic domains. PKS is a state-of-the-art conditional planner that constructs plans in the presence of incomplete information. Unlike traditional planners, PKS builds plans at the knowledge level, by representing and reasoning about how the planner's knowledge state changes during plan generation. Actions are specified in a STRIPS-like [5] manner in terms of action preconditions (state properties that must be true before an action can be executed) and action effects (the changes the action makes to properties of the state). PKS can build contingent plans with sensing actions, and supports numerical reasoning, run-time variables [4], and features like functions that arise in real-world planning scenarios.

Like most AI planners, PKS operates best in discrete, symbolic state spaces described using logical languages. As a result, research that addresses the problem of integrating planning on real-world robot platforms often centres around the problem of representation, and how to abstract the capabilities of a robot and its working environment so that it can be put in a suitable form for use with a goal-directed planner. A key problem when constructing such a representation is the question of how to encode the planner's knowledge about objects that are not completely described in the domain model (but which are known or believed to exist), and how to make assertions about domain properties that reference such objects. Similarly, as new information becomes available to the planner, for instance as a result of sensorimotor processes external to the planner itself, a facility should exist for updating the planner's underlying domain model to make use of this more certain information.

To address these problems in complex planning domains, we report on two main threads of research related to the PKS planner. First, we present an extension of work first reported in deliverable D3.2.1 as part of Task 3.2.4, which uses interval-valued fluents for modelling uncertain numerical information, in order to capture the effects of noisy sensors and noisy effectors at the planning level. Since noisy information is common in real-world robot domains, we believe these extensions will be particularly useful in integrating high-level planning with the low-level sensorimotor systems on Xperience. Moreover, this representation gives rise to a form of indexical (or relative) referencing during plan generation, which addresses the problem of making assertions about domain properties for partially-defined objects. This work is described in [Pet14], attached to this deliverable.

Second, we report on an application programming interface (API) to the planner component, developed as part of Task 3.2.3, which supports the modification of existing planning domains at run time using

the Internet Communications Engine (ICE). In particular, the API abstracts common planning activities already supported by PKS, including functions for adding new actions, properties, and objects to a planning domain, and offers a network-based solution for communicating with the planner component. One of the main contributions of this interface is that it is designed to be generic, which offers the possibility that alternative planners could be used in place of PKS, facilitating future integration tasks, provided they support the same interface. Additional details are described in the attached paper [Pet13].

In addition to work on PKS, we also report on related research which aims to learn knowledge-level action representations of the form used by PKS, from noisy and incomplete observations. This work contributes to Task 3.2.2 and Task 2.3.2, and considers the problem of learning the role of both relational and functional properties in the precondition and effect blocks of knowledge-level action models. In particular, learnt functional properties can act as indexical references to objects which may not have been previously defined in a domain. This work is described in the paper [MP13], attached below.

Finally, we also present preliminary work on a second planning formalism, called reward-adaptive planning, which combines online Monte-Carlo planning with Inverse Reinforcement Learning. This work offers an alternative to PKS with particular emphasis on planning in multiagent environments with probabilistic domain models. This work contributes to both Task 3.2.3 and Task 3.2.4, with a view towards future applications in communication and interaction as part of Task 3.2.5. A short preview of this work is presented in the final attached paper [Val14].

Overall, this deliverable reports a number of significant developments:

- We have implemented a proof-of-concept extension to PKS for *approximate reasoning with numeric fluents*, based on the concept of interval-valued functions. This representation builds on theoretical work first reported in deliverable D3.2.1 and provides a middle ground between planners that cannot work with notions of knowledge and belief, and those that use full probabilistic representations. An initial implementation consists of a standalone software library, providing support for interval-based reasoning, which has been integrated with PKS using the notion of semantic attachments [3].
- Rule learning was extended for *learning knowledge-level domain models*, similar to those supported by PKS, where knowledge is required as a precondition and acquired as an effect. In particular, the representation was modified to work with knowledge fluents that denote domain properties modelled as functions. The approach was tested on pre-existing robot data, demonstrating that the model learns knowledge fluents and functions in line with a 'gold-standard' domain description. This work provides the necessary basis for learning more complex rules related to robot dialogue.
- Reward-adaptive planning provides a novel approach to egocentric *planning with unknown teammates in partially observable, communication-denied domains*. This method extends a state-of-the-art sample-based POMDP planner with interactive reward learning, in order to simultaneously select actions and estimate teammate behavioural models. Experimental results demonstrate that reward-adaptive planning can lead to robust collaboration in varying tasks and team compositions, while also scaling to large problem spaces with complex teamwork constraints.
- We have continued to develop associated planning machinery, such as the *plan execution monitor*, which has been updated to take advantage of improved reasoning capabilities in the planner, and new features provided by the *high-level planning API*. In conjunction with this work, we are continuing to develop high-level domain descriptions for use with our planners (especially our new POMDP-based planner) on the ARMAR platform. Such domains aim to capture the sophisticated capabilities of the ARMAR robot, including object manipulation with multiple grippers and movement between workspaces in its operating environment.

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

- We continue to investigate the role of probabilistic and numeric models in high-level planning and monitoring processes, with our work in [Val14] and [Pet13] adding to our previous contributions here. Since we expect nondeterminacy to arise as the result of perception and action at the sensorimotor level, we continue to study how best to utilise such information at the higher control levels. Currently, we are exploring two approaches: (1) the use of rapid replanning [11], which has been successfully applied by planners in the probabilistic track of the International Planning Competition [6], and (2) the integration of probabilistic information with traditional symbolic logic-based planning approaches.

- The extensions to PKS we reported in this deliverable have primarily focused on the issue of representation and generation in task-based robotics domains. However, this workpackage will also explore the application of modern planning techniques to problems in natural language generation [8, 1, 7, 2]. We continue to explore the use of general-purpose planning for dialogue planning with speech acts, which will be reported in future deliverables.
- In real-world settings, actions are typically non-deterministic and observations are often noisy and incomplete. We plan to make the rule learning model more robust and widely applicable to real-world data by learning probabilistic planning operators, and by expanding the rule extraction process to generate more expressive rules when learning from noisy and incomplete observations.
- We are also investigating dialogue-based enhancements to reward-adaptive planning, where collaborations can be improved through limited communication between teammates. In particular, we are currently working towards extensions in challenging multi-agent domains involving interaction with human-controlled teammates, whose behaviour is not known in advance and is hard to model precisely.

References

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

[2] M. Brenner and I. Kruijff-Korbayová. A continual multiagent planning approach to situated dialogue. In *Proceedings of the 12th Workshop on the Semantics and Pragmatics of Dialogue (LONDIAL 2008)*, pages 67–74, 2008.

[3] C. Dornhege, P. Eyerich, T. Keller, S. Trüg, M. Brenner, and B. Nebel. Semantic attachments for domain-independent planning systems. In *Proceedings of the International Conference on Automated Planning and Scheduling (ICAPS 2009)*, pages 114–121, 2009.

[4] O. Etzioni, S. Hanks, D. Weld, D. Draper, N. Lesh, and M. Williamson. An approach to planning with incomplete information. In W. Swartout, B. Nebel, and C. Rich, editors, *Proceedings of the Third International Conference on Principles of Knowledge Representation and Reasoning (KR-92)*, pages 115–125, Cambridge, MA, Oct. 1992. Morgan Kaufmann Publishers.

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

[6] ICAPS. International Planning Competition. *http://ipc.icaps-conference.org*, 2008.

[7] A. Koller and R. Petrick. Experiences with planning for natural language generation. In *Scheduling and Planning Applications woRKshop (SPARK 2008) at ICAPS 2008*, Sept. 2008.

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

[9] R. P. A. Petrick and F. Bacchus. A knowledge-based approach to planning with incomplete information and sensing. In M. Ghallab, J. Hertzberg, and P. Traverso, editors, *Proceedings of the Sixth International Conference on Artificial Intelligence Planning and Scheduling (AIPS-2002)*, pages 212–221, Menlo Park, CA, Apr. 2002. AAAI Press.

[10] R. P. A. Petrick and F. Bacchus. Extending the knowledge-based approach to planning with incomplete information and sensing. In S. Zilberstein, J. Koehler, and S. Koenig, editors, *Proceedings of the International Conference on Automated Planning and Scheduling (ICAPS-04)*, pages 2–11, Menlo Park, CA, June 2004. AAAI Press.

[11] S. Yoon, A. Fern, and R. Givan. FF-Replan: A baseline for probabilistic planning. In *Proceedings of the International Conference on Automated Planning and Scheduling (ICAPS-07)*, pages 352–359, 2007.

Attached Papers

[MP13] Kira Mourão and Ronald P. A. Petrick. Learning knowledge-level domain dynamics. In *Proceedings of the ICAPS 2013 Workshop on Planning and Learning (PAL)*, pages 23–31, Rome, Italy, June 2013.

[Pet13] Ronald P. A. Petrick. An application programming interface to high-level planning. Technical Report Internal Technical Report, University of Edinburgh, 2013.

[Pet14] Ronald P. A. Petrick. Approximate reasoning with numeric fluents for contingent planning. In *AAAI Conference on Artificial Intelligence.*, 2014. submitted.

[Val14] Aris Valtazanos. Reward-adaptive planning with unknown teammates from limited observations. In *Workshop of the UK Planning and Scheduling Special Interest Group (PlanSIG 2013)*, Edinburgh, Scotland, United Kingdom, January 2014.

Learning knowledge-level domain dynamics

Kira Mourão
School of Informatics
University of Edinburgh
Edinburgh, EH8 9AB, UK
kmourao@inf.ed.ac.uk

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

Abstract

The ability to learn relational action models from noisy, incomplete observations is essential to support planning and decision-making in real-world environments. While some methods exist to learn models of STRIPS domains in this setting, these approaches do not support learning of actions at the knowledge level. In contrast, planning at the knowledge level has been explored and in some domains can be more successful than planning at the world level. In this paper we therefore present a method to learn knowledge-level action models. We decompose the learning problem into multiple classification problems, generalising previous decompositional approaches by using a graphical deictic representation. We also develop a similarity measure based on deictic reference which generalises previous STRIPS-based approaches to similarity comparisons of world states. Experiments in a real robot domain demonstrate our approach is effective.

Introduction

The related problems of planning and learning domain dynamics in domains with incomplete knowledge and uncertainty are both challenging. The planning problem has been tackled using the possible worlds paradigm (Weld *et al.*, 1998; Bonet and Geffner, 2000; Bertoli *et al.*, 2001), where the planner reasons about actions across all possible worlds in which the agent might be operating given its current knowledge. An alternative is to use a knowledge-level representation that describes the agent's knowledge without enumerating possible worlds. One such approach is to restrict the agent's knowledge to simple relational and functional properties using *knowledge fluents*, and then plan with these structures either directly (Petrick and Bacchus, 2002, 2004) or indirectly through compilation techniques (Palacios and Geffner, 2009), in an attempt to build plans more efficiently. However, while a few approaches have tackled learning domain dynamics with incomplete knowledge (Amir and Chang, 2008; Zhuo *et al.*, 2010; Mourão *et al.*, 2012), none have considered learning knowledge-level actions, such as would be required by a planner operating directly at that level.

In this paper we present a method for learning action rules in knowledge domains. We consider the problem of acquiring domain models from the raw experiences of an agent exploring the world, where the agent's observations are incomplete, and observations and actions are subject to noise. The domains we consider are based on relational STRIPS domains (Fikes and Nilsson, 1971) but also include functions, run-time variables and knowledge fluents.

We tackle the problem of learning action models from noisy and incomplete observations by decomposing the problem into multiple classification problems, similar to the work of Halbritter and Geibel (2007) and Mourão *et al.* (2009; 2010; 2012). Our approach generalises these earlier approaches by using a decomposition based on a deictic representation. We represent world states as graphs and develop a similarity measure, also based on deictic reference, to perform similarity comparisons between states. The features used to measure similarity are closely related to the rules underlying the true action models. We reuse the rule extraction method of Mourão *et al.* (2012) to derive planning operators from classifiers trained using our new representation.

We test our approach in a real robot domain. The robot bartender (Petrick and Foster, 2013) serves drinks to customers by generating plans based on input from its vision and dialogue processing systems. State observations derived from these systems can be incomplete or noisy, due to sensing errors. Therefore states are modelled at the knowledge level, with functions and run-time variables used to capture customer requests. Our experiments show that the domain models we learn for the robot bartender perform as well as a "gold-standard" hand-written domain model used to generate the robot's plans.

The Learning Problem

A *domain* $\mathcal{D}$ is defined as a tuple $\mathcal{D} = \langle \mathcal{O}, \mathcal{P}, \mathcal{F}, \mathcal{A} \rangle$, where $\mathcal{O}$ is a finite set of world objects, $\mathcal{P}$ is a finite set of predicate (relation) symbols, $\mathcal{F}$ is a finite set of function symbols, and $\mathcal{A}$ is a finite set of actions. Each predicate, function, and action also has an associated arity. A *fluent expression* of arity n is a statement of the form:
(i) $p(c_1, c_2, ..., c_n)$, where $p \in \mathcal{P}$, and each $c_i \in \mathcal{O}$, or
(ii) $f(c_1, c_2, ..., c_n) = c_{n+1}$, where $f \in \mathcal{F}$, and each $c_i \in \mathcal{O}$.

A *real-world state* is any set of positive or negative fluent expressions, and $\mathcal{S}$ is the set of all possible states. State observations may be incomplete, so we assume an open world where unobserved fluents are deemed to be unknown. At the world level, for any state $s \in \mathcal{S}$, fluent ϕ is true at s iff $\phi \in s$, and false at s iff $\neg\phi \in s$. A fluent and its negation cannot both be in s. If $\phi \notin s$ and $\neg\phi \notin s$ then ϕ is unobserved.

At the knowledge level we transform state observations of the real world into *knowledge states*: statements about the agent's knowledge of the world. A *knowledge fluent* $K\phi$ denotes whether a real-world fluent ϕ is known to be true in the world ($K\phi$), false in the world ($K\neg\phi$) or unknown ($\neg K\phi$ and $\neg K\neg\phi$). Therefore at the knowledge level the closed world assumption can be reinstated and whenever both $K\phi \notin s$ and $K\neg\phi \notin s$, we know that $\neg K\phi \in s$ and $\neg K\neg\phi \in s$. Additionally we introduce the operator K_v which indicates whether the value of a function $f(c_1, c_2, \ldots, c_n)$ is known ($K_v(f(c_1, c_2, \ldots, c_n))$) or unknown ($\neg K_v(f(c_1, c_2, \ldots, c_n))$), regardless of the actual value. Thus $(\exists d \in \mathcal{O})K(f(c_1, \ldots, c_n) = d) \equiv K_v(f(c_1, \ldots, c_n))$. All states at the knowledge level are written entirely in terms of these knowledge fluents.

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 knowledge fluent expressions. We consider two different kinds of action effects. First, we allow STRIPS-like effects, where each $e \in Eff_a$ has the form $add(\phi)$, or $del(\phi)$, and ϕ is any knowledge fluent expression. Second, we permit *conditional effects* of the form $C_e \Rightarrow add(\phi)$ or $C_e \Rightarrow del(\phi)$. Here, C_e is any set of knowledge fluent expressions, and is referred to as the *secondary preconditions* of effect e. Action preconditions and effects can also be parameterised. An action with all of its parameters replaced with objects from $\mathcal{O}$ is an *action instance*.

In contrast to STRIPS domains, which assume that objects mentioned in the preconditions or the effects must be listed in the action parameters (the *STRIPS scope assumption* (SSA)), we make the more general *deictic scope assumption* that objects mentioned in the preconditions or effects are either action parameters or are directly or indirectly related to the action parameters, i.e., they have a deictic term (see Deictic Reference section).

We restrict previous domain knowledge to the assumption of a weak domain model where the agent knows how to identify objects, has acquired predicates to describe object attributes and relations, and knows what types of actions it may perform, but not the appropriate contexts for the actions, or their effects. Experience in the world is then developed by observing changes to object attributes and relations when "motor-babbling" with primitive actions.

The task of the learning mechanism is to learn the preconditions and effects Pre_a and Eff_a for each $a \in \mathcal{A}$, from data generated by an agent performing a sequence of randomly selected actions in the world and observing the resulting states. The sequence of states and action instances is denoted by $s_0, a_1, s_1, ..., a_n, s_n$ where $s_i \in \mathcal{S}$ and a_i is an instance of some $a \in \mathcal{A}$. Our data consists of *observations* of the sequence of states and action instances $s'_0, a_1, s'_1, ..., a_n, s'_n$, where state observations may be noisy (some $\phi \in s_i$ may be observed as $K\neg\phi \in s'_i$) or incomplete (some $\phi \in s_i$ are not in s'_i). Action failures are allowed: the agent may attempt to perform actions whose preconditions are unsatisfied. In these cases the world state does not change, but the observed state may still be noisy or incomplete. To make accurate predictions in domains where action failures are permitted, the learning mechanism must learn both preconditions and effects of actions.

Consider, for example, the dishwasher domain (shown in Figure 1), a domain where an agent can load and unload a dishwasher, switch it on, and check the status of the dishwasher. In our examples we use a PDDL-like syntax to represent knowledge fluents and states. For a state where the agent knows the dishwasher contains some dirty dishes, the real world state could be:

```
(AND (status=dirty) (¬in washer dish1) (¬in washer dish2)
     (in washer dish3) (isdirty dish1) (¬isdirty dish2)
     (isdirty dish3) (in washer dish4) (isdirty dish4)).
```

From this the agent might observe the knowledge state:

```
(AND Kv(status) K(status=dirty) K(¬in washer dish1)
     K(in washer dish3) K(isdirty dish1) K(isdirty dish3)
     K(¬in washer dish2) K(¬isdirty dish2)).
```

A sequence of knowledge states and actions could be:

```
s0:(AND Kv(status) K(status=dirty) K(¬in washer dish1)
     K(in washer dish3) K(isdirty dish1) K(isdirty dish3)
     K(¬in washer dish2) K(¬isdirty dish2))
a1:(load washer dish1)
s1:(AND Kv(status) K(status=dirty) K(in washer dish1)
     K(in washer dish3) K(isdirty dish1) K(isdirty dish3)
     K(¬in washer dish2) K(¬isdirty dish2))
a2:(switchon)
s2:(AND K(in washer dish1) K(in washer dish3)
     K(¬in washer dish2) K(¬isdirty dish2))
a3:(checkstatus)
s3:(AND K(in washer dish1) K(in washer dish3)
     K(¬in washer dish2) K(¬isdirty dish2)
     Kv(status) K(status=clean)).
```

Taking a sequence of such inputs, we learn action descriptions for each action in the domain, such as in Figure 1.

Related Work

Knowledge-level reasoning is not a new idea (Newell, 1982), and the use of knowledge fluents like $K\phi$ and $K\neg\phi$ has been explored as a means of restricting the syntactic form of knowledge assertions in exchange for more tractable reasoning, e.g., by avoiding the drawbacks of possible-worlds models (Demolombe and Pozos Parra, 2000; Soutchanski, 2001; Petrick and Levesque, 2002). Planners like PKS (Petrick and Bacchus, 2002, 2004) attempt to work directly with knowledge-level models, similar to those of knowledge fluents, while approaches like (Palacios and Geffner, 2009) compile traditional open world planning problems into a classical closed-world form, in the process automatically generating knowledge fluents.

Only a few approaches to learning action models are capable of learning under either partial observability (Amir and Chang, 2008; Yang *et al.*, 2007; Zhuo *et al.*, 2010), noise in any form (Pasula *et al.*, 2007; Rodrigues *et al.*, 2010), or both (Halbritter and Geibel, 2007; Mourão *et al.*, 2010). Some rely on prior knowledge of the action model, such as using known successful plans (Yang *et al.*, 2007; Zhuo *et al.*, 2010), or excluding action failures (Amir and Chang, 2008). None explicitly support functions or knowledge fluents.

While the representation used in our previous work (Mourão *et al.*, 2012) does not support functions or the K_v

```
(define (domain dishwasher)
  (:predicates (in ?washer ?dish) (isdirty ?dish))
  (:functions (status ?washer) = ?washstatus)
  (:constants clean dirty)

(:action checkstatus
   :parameters (?washer)
   :precondition ()
   :effect (Kv(status(?washer))))

(:action switchon
   :parameters (?washer)
   :precondition ()
   :effect (¬Kv(status(?washer))))

(:action unload
   :parameters (?washer ?dish)
   :precondition (and Kv(status(?washer)) K(status(?washer)=clean) K(in ?washer ?dish))
   :effect (and K(¬in ?washer ?dish) K(¬isdirty ?dish)))

(:action load
   :parameters (?washer ?dish)
   :precondition (K¬(in ?washer ?dish) )
   :effect (and K(in ?washer ?dish)
            (when (K(isdirty ?dish)) (Kv(status(?washer)) K(status(?washer)=dirty)))
            (when (and ¬K(isdirty ?dish) Kv(status(?washer)) K(status(?washer)=clean)) (¬Kv(status(?washer)))))))
```

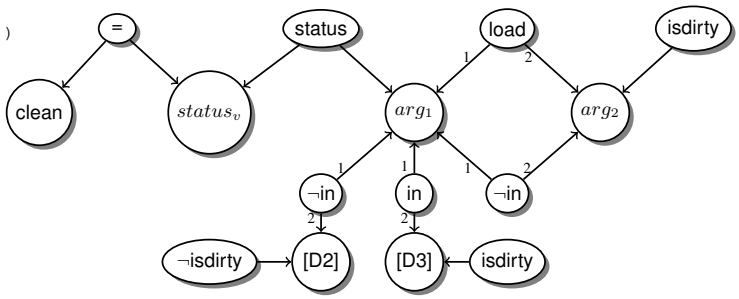


Figure 1: A description of the dishwasher domain (left), and (right) a graphical representation of state s_0 when combined with the `load` action. The node representing the result of the `status(?washer)` function is labelled $status_v$.

operator, it could support knowledge fluents of the form $K\phi$. In this earlier work, each fluent ϕ was assigned one of the values 1, -1 or $*$ which correspond to the $K\phi$, $K\neg\phi$ and $\neg K\phi$/$\neg K\neg\phi$ defined earlier. However, the learning method depended on the SSA to generate vector representations of states. With the introduction of functions the SSA no longer applies and the vector representation can no longer be used.

Our new approach depends upon coding world states (and correspondingly, preconditions and effects) in terms of deictic reference (Agre and Chapman, 1987). A deictic representation maintains pointers to objects of interest in the world, with objects coded relative to the agent or current action. Previous work in learning action models has also used deictic reference (Benson, 1996; Pasula *et al.*, 2007) because there are benefits in doing so: it reduces the size of the state representation, by limiting the observations to a small number of objects, and also permits generalisation across different instances of the same action, as the observations are described in terms of the action and the agent instead of specific objects.

Method outline

Our approach to learning knowledge-level action models is based on the work of Mourão *et al.* (2012), but differs significantly in terms of the representation used and in the details of the learning process. Real-world states are observed by an agent as a knowledge state where each fluent $\phi(\neg\phi)$ is observed as $K\phi(K\neg\phi)$ and when $Kf(c_1,\ldots,c_n) = c_{n+1}$, also $K_v(f(c_1,\ldots,c_n))$. We represent these observations as graphs where objects, *known* fluents and actions are nodes in the graph, and edges link fluents to their arguments. The prediction problem is then to determine which nodes in a graph change as the result of an action. Our strategy is to decompose the prediction problem into many smaller classification problems, where each classifier predicts change to a single fluent of the overall state, given an input situation and an action. After training the classifiers we derive planning operators from the learnt parameters, using the same process described by Mourão *et al.* (2012).

Central to the classification process is a measure of similarity between states. Commonly, similarity comparisons between graphs are performed using graph kernels which implicitly map into another feature space; here we define an explicit mapping of state graphs into a feature space, where the mapping is calculated via a simple relabelling scheme.

The remainder of this paper is structured as follows. We define deictic reference and show how it is used to create the graphical representation of world states. Then we explain how we calculate a similarity measure for two states based on deictic reference. The structure and operation of the classification learning model is described, followed by an explanation of how rules are extracted from the classifiers. Finally, we give some experimental results and discuss conclusions and future work.

Deictic reference

Deictic reference underlies a number of aspects of the learning process. The structure of the state observation graphs is determined by the deictic terms of the objects in the state. In turn, this means that the feature space mapping relies on deictic reference to map objects with the same roles in an action to the same points in the feature space.

In the deictic representation we use, we code objects with respect to the action. Every action parameter is referred to

by its own unique deictic term, corresponding to its position in the parameter list. Constant values are also considered to have their own deictic terms. Deictic terms referring to other objects are their definitions in terms of their relations with the action parameters and other objects.

Thus, similar to Pasula *et al.* (2007), a deictic term is a variable V_i and a constraint ρ_i where ρ_i is a set of literals defining V_i in terms of the arguments of the current action and any previously defined V_j ($j < i$). Then an object has a deictic term if it is an argument of the current action, or it is related directly, or indirectly via other objects, to the arguments of the action. For functions, every argument must already have a deictic term in order for the function result to have a deictic term.

Additionally, we add the constraint that for an object to have a deictic term, it must be linked by a positive fluent to either an action parameter, or another object which has a deictic term (the *positive link assumption*). This additional restriction accounts for the open world representation now in place (at the world level), avoiding deictic terms of the form "the-object-not-under-the-object-I-am-picking-up-and-not-on-the-floor", which will not usually be unique and seem counter-intuitive. Apart from the action parameters, any object in a state may be referred to by several deictic terms, and (unlike Pasula *et al.* (2007)) any deictic term may refer to several objects in a state.

We say that an object has an n-th order deictic term when n is the minimum number of relations relating the object to an action parameter. Thus the parameters of the action have zero-order deictic terms, while objects related to the action parameters have first-order deictic terms.

For example, in the dishwasher domain (Figure 1), if the action were `(load washer dish1)` in state s_0, then action parameters `washer` and `dish1` would have deictic terms arg_1 and arg_2, indicating their positions in the `load` argument list. Relative to the `(load washer dish1)` action, `dish2` is referred to by deictic terms $x : \neg in(washer, dish2)$ and $x : \neg in(washer, dish2) \wedge \neg isdirty(x)$, but not $x : \neg isdirty(x)$ alone. The `dish2` node is labelled `[dish2]` to indicate that it represents all objects with the same deictic terms as `dish2`.

State representation

We represent a knowledge state by a graph, where objects (as deictic terms), known fluents, and the current action are represented by nodes in the graph. Edges link fluents (or the current action) and their arguments, and are labelled with the argument position.

Both predicates and functions are represented by nodes and are only present in the graph if known. However, for functions additionally the result of a function f is represented by a special node f_v, which denotes the deictic term defined by the function. The actual value of the function is linked to f_v by an equality node. Thus, for example, $K(f(c_1, c_2) = c_3)$ would be represented as in Figure 2.

The size of the graph is limited by restricting the deictic terms to zero- or first-order terms only.[1] Using only zero-order terms would be equivalent to working with a STRIPS representation, as we would only consider parameters of the action during learning. Here, we require first-order deictic terms to represent functions, as the result of a function will not usually be an action parameter. Figure 1 shows a graph encoding the state s_0 in the context of the `(load washer dish1)` action, after converting the objects to deictic terms.

[1] Higher order terms are possible but are left to future work.

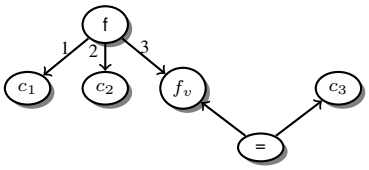


Figure 2: Representation of $K(f(c_1, c_2) = c_3)$. c_1,c_2 and c_3 are represented by nodes labelled with their deictic terms (here we assume they are constants). The function node f has edges to nodes c_1 and c_2, indicating they are parameters, and also an edge connecting to the result node f_v. f_v and c_3 are linked by an equals node, indicating that the value of $f(c_1, c_2)$ is c_3.

Calculating changes

Our classification model operates by taking a knowledge state (as a graph) as input, and predicting which knowledge fluents will change. Each training example must therefore consist of a prior state, an action, and the changes resulting from performing the action on the state.

We denote changes by creating a *change graph*, created by annotating the prior state graph with additional *marker* nodes (similar to Halbritter and Geibel (2007)). Marker nodes have an edge linking to the fluent node which changed. Given a prior and successor state, a marker node M_ϕ is added to the change graph for every fluent ϕ which changes real-world value between the states. A marker node $M_{K\phi}$ is added for every fluent which changes knowledge state between the states. During training, each classifier will learn to predict the presence or absence of a single marker node in the graph (i.e. whether the associated fluent changes).

It is straightforward to determine the marker nodes to add to the change graph, given prior and successor state graphs. For any fluent ϕ in the prior state, if $\neg\phi$ is in the successor state, we add M_ϕ. If neither ϕ nor $\neg\phi$ are present in the successor state we add $M_{K\phi}$. Similarly, any fluent present in the successor state but not the prior state is added to the change graph, along with $M_{K\phi}$. For example, for the `load` action in Figure 1, the changes to the state would be indicated by a node $M_=$ linked to the `(status`$_v$ `= clean)` node and a node M_{in} linked to the `(¬in arg`$_1$ `arg`$_2$`)` node.

Crucially, because the successor state immediately follows the prior state, matching fluents can be determined by matching the actual objects which were arguments of the fluents. In general such matching is not possible between states. We return to this point when describing the structure of the learning model.

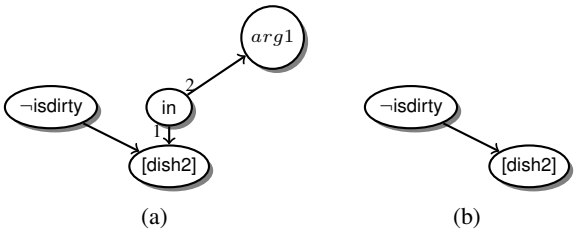


Figure 3: Valid (a) and invalid (b) subgraphs of the state graph in Figure 1.

Comparing states using deictic reference

The classification process requires a measure of similarity between states. In classification problems, graphical inputs are usually mapped either implicitly — via graph kernels — or explicitly into a feature space where the inner product provides a similarity score.

A feature space where the features are all possible conjunctions of fluents would seem to be ideal for learning action preconditions which are arbitrary conjunctions of fluents. However, similarity calculations in this space are unlikely to be tractable as it is closely related to the subgraph kernel (mapping graphs to the space of all possible subgraphs), known to be NP-hard (Gärtner *et al.*, 2003), and contains the feature space of the DNF kernel (Sadohara, 2001; Khardon and Servedio, 2005), which cannot be used by a perceptron to PAC-learn DNF (Khardon *et al.*, 2005).

Following Mourão *et al.* (2012) we therefore work with the space of all possible conjunctions of fluents of length $\leq k$ for some fixed k. The space is further restricted so that in every conjunction, every object must have a valid deictic term depending only on fluents in the conjunction. This restriction avoids learning meaningless preconditions where variables in the preconditions are undefined e.g., action $a(x, y)$ with precondition $p(z)$. Also, it forces the similarity comparison to account for the roles of objects (as defined by their deictic terms) by mapping objects in different states, but with similar deictic terms, to similar sets of features.

We define an explicit mapping into this space, creating a (sparse) feature vector. Each element of the vector corresponds to a conjunction of up to k fluents present in the state graph, subject to the restriction that every object has a valid deictic term depending only on fluents in the conjunction. E.g. considering subgraphs of the dishwasher state shown in Figure 1, Figure 3a would be valid but not Figure 3b. The value of each element in the vector is the number of occurrences of the corresponding subgraph in the state graph.

The feature vector can be constructed via a labelling scheme similar to the process used in some graph kernel calculations (Shervashidze *et al.*, 2011). First we label object nodes with either their position in the action parameter list, or their type if they are not listed in the action parameters. Next we identify the set of *core* fluents, whose arguments are contained within the set of action parameters. By definition, every argument of a core fluent has a deictic term, and so any conjunction of core fluents will be valid.

For each conjunction C of i core fluents ($1 \leq i \leq k$), we identify the set of *supported* fluents, whose arguments are also arguments of either the action or a fluent in C. For example, in Figure 3a, `in` is a core fluent and `isdirty` is a supported fluent. Every argument of a supported fluent will have a deictic term depending only on fluents in C. Now we create all possible conjunctions of supported fluents of size $k - i$ or fewer, and combine each with C in turn to give C'.

We convert each fluent in C' to a string encoding the fluent, the argument positions and their ordering. E.g. `(in arg1 dish)` could convert to "in1(arg1)2(dish)". (Note that here "dish" is a type.) Next we sort the fluent strings and concatenate them to give a unique string representing C'. This string is looked up in a lookup table mapping strings to feature vector locations. If the string is not found in the lookup table, we add a new entry with value 1 to the feature vector and a matching entry in the lookup table. Otherwise we increment the existing entry in the feature vector.

Structure of the learning model

Using the state graphs defined above, the structure of the learning model can be defined. Given a state $s \in \mathcal{S}$ and an action $a \in \mathcal{A}$, the model predicts the successor state s'. Equivalently, the set of fluents which change between s and s' — the deltas — can be predicted. Our strategy is to use multiple classifiers where each classifier predicts change to one or a small set of fluents of the overall state, given an input situation and an action.

Such a structure requires a classifier for each possible fluent node in any state graph. Then given a state graph, we predict the effect of an action by predicting whether each fluent node in the graph changes or not. The conjunction of all the predicted changes is the predicted effect of the action. For example, in Figure 1, consider the following fluents:

1. `(¬in` $\texttt{arg}_1$ $\texttt{arg}_2$`)`
2. `(¬in` $\texttt{arg}_1$ `[dish2])`
   where $[dish2] = \{x : \neg in(arg_1, x) \wedge \neg isdirty(x)\}$
3. `(in` $\texttt{arg}_1$ `[dish4])`
   where $[dish4] = \{x : in(arg_1, x) \wedge \neg isdirty(x)\}$
4. `(¬in` $\texttt{arg}_1$ `[dish5])`
   where $[dish5] = \{x : \neg in(arg_1, x)\}$

Fluents (1) and (2), present in the graph, and (3), not present, but possible, would each have their own classifier. Additionally we must consider fluents with more general deictic terms, such as (4), which includes both (1) and (2). The classifier associated with (4) predicts whether fluent `(in` $\texttt{arg}_1$ `x)` changes for *any* x not in arg_1, whereas the classifiers associated with (1) and (2) predict whether `(in` $\texttt{arg}_1$ `x)` changes for x which is the second argument of the `load` action (1), or for x which is not in arg_1 and not dirty (2). However, although there are many possible fluent nodes, in practice most of the associated classifiers are not instantiated by our algorithm, resulting in a default prediction of no change for the corresponding fluents.

Our training algorithm therefore has two tasks. First, it manages sets of classifiers, in terms of deciding which classifier to train on which data, and when to instantiate new classifiers. Second, it trains the classifiers. Likewise, at prediction our algorithm must select which classifiers to use, and then generate a prediction from them.

As in the work of Mourão *et al.* (2012), we will use voted perceptron classifiers (Freund and Schapire, 1999), since they are known to be robust to noise and efficient to train. We use the standard procedures for training of, and prediction from, individual classifiers. In our algorithm descriptions below, $train(c, x, y)$ denotes updating classifier c with training example (x, y), and $predict(c, x)$ returns classifier c's prediction of the class of example x. We now describe how classifiers are managed during training and prediction.

Initialisation

The algorithm is provided with the set of action labels $\mathcal{A}$, the set of predicates $\mathcal{P}$, the set of functions $\mathcal{F}$, and the number and types of their arguments. In the following description we treat any function $f(c_1, \ldots, c_n) = c_{n+1}$ as two predicates: $f'(c_1, \ldots, c_n, f_v)$ and $equals(f_v, c_{n+1})$, corresponding to the graph structure defined earlier, and contained in an extended set $\mathcal{P}'$. The learning algorithm maintains a set of classifiers $C_{a,p}$ for each action a and predicate p. Initially each $C_{a,p}$ is empty and is populated as training examples are seen by the algorithm. Every member of $C_{a,p}$ will be a classifier $c_{\overline{m}}$ associated with a different tuple of deictic terms $\overline{m}$ which are valid arguments of p. For example, in our dishwasher domain, one of the sets of classifiers would be $C_{(load,in)}$: the set of classifiers which predict changes to the `in` predicate when the `load` action is performed. A member of $C_{(load,in)}$ could be $c_{(arg_1, \{x:in(arg_1,x) \wedge \neg isdirty(x)\})}$.

Training

Each training example consists of a state description x_i, an action a_i, and a successor state x'_i. Both state descriptions are converted into state graphs and a change graph δ_i, based on the action a_i as previously described. The marker nodes from the change graphs will provide target values.

The training process is outlined in Algorithm 1. In the main loop we identify all the fluent nodes $p(\overline{m})$ in a training example x ($\mathit{fluentNodes}(x)$) and determine whether each fluent changed in the example, by checking whether the node has a marker node in the change graph δ ($isFluentInDelta$). If the fluent changed, the target value y is set to 1, otherwise it is set to 0. Then $updateClassifiers$ is called for each fluent node.

In $updateClassifiers$, classifiers which match $p(\overline{m})$ are trained, and new classifiers may be instantiated if necessary. Recall that in principle there is one classifier for every possible fluent, each initially predicting no change to the fluent. 'No-change' classifiers are not actually instantiated since no prediction function is needed. During training, $updateClassifiers$ must decide which classifiers to update, i.e., first, whether to instantiate a classifier, and second, which classifier(s) to train. There is also a secondary goal of minimising the number of instantiated classifiers to keep the calculation tractable.

Thus given any $p(\overline{m})$ we first seek classifiers which predict for $p(\overline{m})$ and then update them with the training example (x, y). A classifier predicts for $p(\overline{m})$ if it is labelled with $p(\overline{m})$ (an exact match) or labelled with $p(\overline{m}')$ where $\overline{m}'$ is equal to or more general than $\overline{m}$ (a subset match). For example, if $q(\{x : a(x) \wedge b(x)\})$ is a unary predicate then

Algorithm 1 Training

Require: training egs $(x_1, a_1, \delta_1), ..., (x_n, a_n, \delta_n) \in X$
Ensure: trained classifiers
1: $C_{a,p} := \varnothing \ \ \forall a \in \mathcal{A}, \forall p \in \mathcal{P}$
2: **for all** $(x, a, \delta) \in X$ **do**
3: **for all** $p(\overline{m}) \in \mathit{fluentNodes}(x)$ **do**
4: $y := isFluentInDelta(p(\overline{m}), \delta)$
5: $C_{a,p} := updateClassifiers(x, y, \overline{m}, C_{a,p})$

function $updateClassifiers$(state graph x, target y, deictic terms $\overline{m}$, set of classifiers C)
1: $exactMatch := false$; $intersectMatches := \varnothing$
2: **for all** $c \in C$ **do**
3: **if** $subsetMatch(c, \overline{m})$ **then**
4: **call** $train(c, x, y)$
5: **call** $updateReliability(c)$
6: **if** $exactMatch(c, \overline{m})$ **then**
7: $exactMatch := true$
8: **else if** $intersectMatch(c, \overline{m})$ **then**
9: $intersectMatches := intersectMatches \cup \{c\}$
10: **if** $(y \neq 0) \wedge (exactMatch = false)$ **then**
11: $C := C \cup createClassifiers(x, intersectMatches, \overline{m})$
12: **return** C

$q(\{x : a(x)\})$ is more general, and so whenever the former changes, so will the latter. Thus whenever we update $c_{q(\{x:a(x) \wedge b(x)\})}$ we must also update $c_{q(\{x:a(x)\})}$. Formally, we define that if classifier c predicts change for $p(\overline{n})$:

- $exactMatch(c, \overline{m})$ when $\overline{n} = \overline{m}$;
- $subsetMatch(c, \overline{m})$ if the i-th term in $\overline{n}$ is a subset of the i-th term in $\overline{m}$ $\forall i$;

Any classifier $c \in C_{a,p}$ for which $subsetMatch(c, \overline{m})$ holds is trained on the training example (x, y), and a measure of its reliability updated (see below).

Next we consider whether any classifiers should be instantiated. There are two cases where instantiation is required. If there was no exactly matching classifier for $p(\overline{m})$ *and* in our training example $p(\overline{m})$ changed, then $c_{p(\overline{m})}$ should be instantiated. If $p(\overline{m})$ did not change then the original 'no-change' classifier is still correct. Additionally, the deictic terms seen in training examples may be more specific than the underlying rules. For example if a and b are deictic terms we may only ever see changes to $p(a, arg1)$ or $p(b, arg1)$ but the true change could be to $p(a \cap b, arg1)$. To predict change to the correct set of fluents we therefore need to consider more general deictic terms, and so whenever a new classifier is instantiated, classifiers for tuples of more general deictic terms are also instantiated. However, it is undesirable to add a classifier for every possible tuple, so only those supported by the data are added. These are cases where the deictic terms of $p(\overline{m})$ intersect with deictic terms of $p(\overline{n})$ already seen in the data. Such $p(\overline{n})$ can be found by considering the terms of previously instantiated classifiers.

Formally, if classifier c predicts change for $p(\overline{n})$: $intersectMatch(c, \overline{m})$ if the i-th term in $\overline{n}$ intersects the i-th term in $\overline{m}$ $\forall i$. A tally is kept of exact matches and intersect matches for $p(\overline{m})$, and if $c_{p(\overline{m})}$ in instantiated, so are classifiers for all the intersecting cases ($createClassifiers$).

Algorithm 2 Prediction

Require: Unlabelled instance (x, a), model parameters $C_{a,p}$
Ensure: Prediction δ
1: $\delta = \varnothing$
2: **for all** $p(\overline{m}) \in \mathit{fluentNodes}(x)$ **do**
3: **if** $\mathit{getPrediction}(C_{a,p}, x, \overline{m}) = 1$ **then**
4: $\delta = \delta \cup \{p(\overline{m})\}$

function $\mathit{getPrediction}$(set of classifiers C, state graph x, deictic terms $\overline{m}$)
1: $r := 0, y := 0$
2: **for all** $c \in C$ **do**
3: **if** $\mathit{subsetMatch}(c, \overline{m})$ and $r < \mathit{getReliability}(c)$ **then**
4: $y := \mathit{predict}(c, x)$
5: $r := \mathit{getReliability}(c)$
6: **return** y

Reliability and Prediction

The algorithm maintains a reliability score for each classifier ($\mathit{updateReliability}$), used during prediction to select the best classifier. The reliability of a classifier is calculated as the fraction of predictions made which were correct during training. We also maintain the *null reliability*, the reliability which would have been achieved if this classifier had always predicted no change. The null reliability score is thus the fraction of training examples where there was no change. In noisy situations, the null reliability may be higher than the classifier reliability, indicating that many training examples were noisy. In this case, predicting no change gives better results than using the classifier's predictions (on the training set). During prediction, $\mathit{getReliability}$ returns either the classifier reliability or the null reliability, whichever is higher. If the null reliability is higher $\mathit{predict}$ will always predict no change, instead of the classifier's prediction. (Additionally, although not used here, low reliability classifiers can be deleted if the number of classifiers grows too large.)

At prediction, given a test example x, each fluent node $p(\overline{m})$ of x is considered in turn and a search for matching classifiers is performed. If no classifiers are found then the model predicts no change for the fluent $p(\overline{m})$. If exactly one classifier is found then its prediction is used, and if there are multiple matching classifiers, the classifier with the highest reliability score is used.

Learning planning operators

Once the classifiers are trained, planning operators can be derived using the approach of Mourão *et al.* (2012). First, rules are extracted from individual classifiers. Since each voted perceptron classifier predicts change to a single fluent, this results in a set of candidate preconditions for each candidate effect. Second, the candidate preconditions and effects are combined via a heuristic merging process to produce planning operators. These steps are outlined below.

Algorithm 3 Rule extraction

Require: Positive support vectors SV^+
Ensure: Rules $R = \{rule_v : v \in SV^+\}$
1: **for** $v \in SV^+$ **do**
2: $child := v$
3: **while** $child$ only covers +ve training examples **do**
4: $parent := child$
5: **for** each fluent node in $parent$ **do**
6: flip node to its negation and calculate weight
7: $child :=$ child whose parents have least weight difference
8: $rule_v := parent$

Extracting rules from individual classifiers

Extracting rules from individual classifiers in the graphical case is a straightforward reapplication of the approach used for STRIPS vectors (Mourão *et al.*, 2012). A key point is that the decision function of the voted perceptron is a function of the set of support vectors identified during learning, where the set of support vectors is some subset of the set of training examples.[2]

Rules are extracted from a voted perceptron with kernel K and support vectors $SV = SV^+ \cup SV^-$, where SV^+ (SV^-) is the set of support vectors whose *predicted* values are 1 (-1). Value 1 means the corresponding fluent changes, and -1 means there is no change. The positive support vectors are each instances of some rule learnt by the perceptron, and so are used to "seed" the search for rules. The extraction process aims to identify and remove all irrelevant nodes in each support vector, using the voted perceptron's prediction calculation to determine which nodes to remove.

We define the *weight* of any possible state graph $\mathbf{x}$ to be the value calculated by the voted perceptron's prediction calculation before thresholding. The basic intuition behind the rule extraction process is that more discriminative features will contribute more to the weight of an example. Thus the rule extraction process operates by taking each positive support vector and repeatedly deleting the fluent node which contributes least to the weight until some stopping criterion is satisfied. This leaves the most discriminative features underlying the example, which can be used to form a precondition. This algorithm is detailed in Algorithm 3.

Combining rules into planning operators

Finally we combine the rule fragments ((precondition,effect) pairs) resulting from the rule extraction process into planning operators. For each action the process derives a rule (g_{rule}, e_{rule}) from the set of rules $R = \{(g_1, e_1), \ldots, (g_r, e_r)\}$ produced by rule extraction, ordered by decreasing weight. The process first initialises g_{rule} to the highest weighted precondition in R and sets $e_{rule} = \varnothing$. The rule is then refined by combining it with each of the remaining per-fluent rules in turn, in order of highest weight.

Combining rules involves merging the graphs encoding the preconditions, as well as the markers encoding the effects, into a new candidate rule. After merging, a simplifica-

[2]Note that support vectors are therefore state graphs.

tion step removes unnecessary fluents in the preconditions and effects by testing the coverage and weight of the candidate rule without each new fluent. Then the new rule is accepted if its F-score on the training set is within some tolerance of the F-score of the previous rule on the training set. Lastly the rule is translated into PDDL or some variant.

Experiments

We evaluate our approach by learning planning operators in a real robot domain, whose underlying model is defined at the knowledge level. We compare the F-scores for predictions made by both the learnt planning operators and underlying classification model with predictions made by the "gold-standard" domain description: the original specification of the behaviour of the robot.

The data used for training and testing was generated from logs of the JAMES robot bartender system, recorded during a drink ordering scenario in which human subjects were asked to order drinks from the robot. State descriptions were generated by the system's state manager, based on real-world sensor data (vision and automatic speech recognition), interleaved with the names of planned actions generated for the goal of serving all agents. In total, 93 interactions were recorded for 31 human users. Each interaction involves approximately 5-10 robot actions.

The robot bartender domain description is at the knowledge-level, and several actions require functions in their definitions. One action is of particular interest: `ask-drink`, where the robot asks a human customer for their order. If successful, `ask-drink` has the effect that the robot now knows the value of the customer's requests (K_v`(request ?x)`). Although `ask-drink` will also result in the robot knowing the actual drink requested (e.g. `K(request(?x) = water)`) this is only useful at runtime, whereas K_v`(request ?x)` is needed at plan-time. Furthermore, because `ask-drink` involves accurately interpreting the user's chosen drink, it is particularly prone to failure. Therefore it is of additional interest to investigate how well this action is learnt.

Results

A ten-fold cross-validation procedure was used to test the performance of the learning model, and was repeated across different numbers of training examples to assess how many examples would be needed to learn an adequate model. The performance was measured by considering the fluents which the model predicted would change versus the fluents which did change, and calculating the F-score, the harmonic mean of precision and recall (true positives/predicted changes and true positives/actual changes, respectively).

The results were compared to the predictions made by the gold-standard model. In Figure 4 we show F-scores for action predictions made by the classifiers; by rules derived from the classifiers; and by the gold-standard model on data from the robot experiment. As can be seen in the graph, the rules extracted from the classifiers perform similarly to making predictions directly with the classifiers, but with the added benefit of providing action descriptions which can

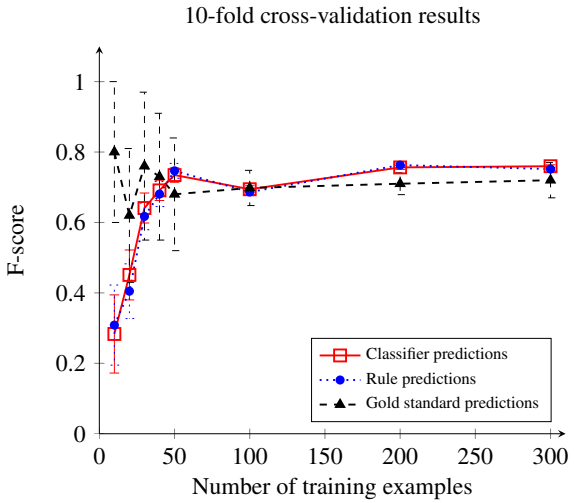


Figure 4: Results from the robot experiment: Mean F-scores from ten-fold cross-validation for predictions from the classifiers, extracted rules and gold-standard action descriptions.

be used for planning. The F-scores for the classifiers and extracted rules are not significantly different from the F-score of the gold standard rules (noise in the domain means that even the gold-standard rules cannot always predict the changes which will or will not occur).

An example of an action description learnt for `ask-drink` with 200 training examples is given below. Fluents marked in italic do not exist in the gold standard domain description. Some fluents are also missing, all relating to preconditions involving other agents which we currently do not represent. However, the crucial K_v`(request ?x)` effect is learnt.

```
(:action ASK-DRINK
 :parameters (?x)
 :precondition (AND K(transHistory RobotAckAttention ?x)
          K(¬transHistory AgentOrdered ?x)
         ¬Kv(request ?x) K(closeToBar ?x) K(faceseen ?x))
 :effect (AND (Kv(request ?x)
               K(transHistory AgentOrdered ?x))))
```

Conclusions and Future Work

Our results show that we can learn knowledge-level planning operators in a noisy robot domain. The approach we use depends on decomposing the learning problem into many small classification problems, using the deictic scope assumption to constrain the problem. Deictic reference also plays an important role in defining the representation for functions and in the similarity calculations made by the classifiers. In future work we plan to test our approach in other real or simulated knowledge-level domains. Another step will be to use the learnt planning operators in an automated knowledge-level planning system such as PKS (Petrick and Bacchus, 2002, 2004).

Acknowledgements This work was partially funded by the European Commission through the EU Cognitive Systems and Robotics projects Xperience (FP7-ICT-270273) and JAMES (FP7-ICT-270435).

References

Agre, P. E. and Chapman, D. (1987). Pengi: An implementation of a theory of activity. In *AAAI*, pages 268–272.

Amir, E. and Chang, A. (2008). Learning partially observable deterministic action models. *JAIR*, **33**, 349–402.

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

Bertoli, P., Cimatti, A., Roveri, M., and Traverso, P. (2001). Planning in nondeterministic domains under partial observability via symbolic model checking. In *Proc. of IJCAI 2001*, pages 473–478.

Bonet, B. and Geffner, H. (2000). Planning with incomplete information as heuristic search in belief space. In *Proc. of AIPS 2000*, pages 52–61.

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

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

Freund, Y. and Schapire, R. (1999). Large margin classification using the perceptron algorithm. *Machine Learning*, **37**, 277–96.

Gärtner, T., Flach, P., and Wrobel, S. (2003). On graph kernels: Hardness results and efficient alternatives. In *Proc. of COLT 2003*, pages 129–143.

Halbritter, F. and Geibel, P. (2007). Learning models of relational MDPs using graph kernels. In *Proc. of MICAI 2007*, pages 409–419.

Khardon, R. and Servedio, R. A. (2005). Maximum margin algorithms with Boolean kernels. *JMLR*, **6**, 1405–1429.

Khardon, R., Roth, D., and Servedio, R. A. (2005). Efficiency versus convergence of Boolean kernels for on-line learning algorithms. *JAIR*, **24**, 341–356.

Mourão, K., Petrick, R. P. A., and Steedman, M. (2009). Learning action effects in partially observable domains (1). In *Proc. of ICAPS 2009 Workshop on Planning and Learning*, pages 15–22.

Mourão, K., Petrick, R. P. A., and Steedman, M. (2010). Learning action effects in partially observable domains (2). In *Proc. of ECAI 2010*, pages 973–974.

Mourão, K., Zettlemoyer, L., Petrick, R. P. A., and Steedman, M. (2012). Learning STRIPS operators from noisy and incomplete observations. In *Proc. of UAI 2012*, pages 614–623.

Newell, A. (1982). The knowledge level. *Artif. Intell.*, **18**(1), 87–127.

Palacios, H. and Geffner, H. (2009). Compiling uncertainty away in conformant planning problems with bounded width. *JAIR*, **35**(1), 623–675.

Pasula, H., Zettlemoyer, L. S., and Kaelbling, L. P. (2007). Learning symbolic models of stochastic domains. *JAIR*, **29**, 309–352.

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

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

Petrick, R. P. A. and Foster, M. E. (2013). Planning for social interaction in a robot bartender domain. In *Proc. of ICAPS 2013, Special Track on Novel Applications*. To appear.

Petrick, R. P. A. and Levesque, H. (2002). Knowledge equivalence in combined action theories. In *Proc. of KR 2002*, pages 303–314.

Rodrigues, C., Gérard, P., and Rouveirol, C. (2010). Incremental learning of relational action models in noisy environments. In *Proc. of ILP 2010*, pages 206–213.

Sadohara, K. (2001). Learning of Boolean functions using support vector machines. In *Proc. of ALT*, pages 106–118.

Shervashidze, N., Schweitzer, P., van Leeuwen, E. J., Mehlhorn, K., and Borgwardt, K. M. (2011). Weisfeiler-Lehman graph kernels. *JMLR*, **12**, 2539–2561.

Soutchanski, M. (2001). A correspondence between two different solutions to the projection task with sensing. In *Commonsense 2001*.

Weld, D. S., Anderson, C. R., and Smith, D. E. (1998). Extending graphplan to handle uncertainty and sensing actions. In *Proc. of AAAI 1998*, pages 897–904.

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

Zhuo, H. H., Yang, Q., Hu, D. H., and Li, L. (2010). Learning complex action models with quantifiers and logical implications. *Artif. Intell.*, **174**(18), 1540–1569.

AN APPLICATION PROGRAMMING INTERFACE TO HIGH-LEVEL PLANNING

Ronald P. A. Petrick
University of Edinburgh
rpetrick@inf.ed.ac.uk

2013-12-18

Internal technical report – not to be distributed publicly

Abstract

This report describes part of UEDIN's contribution to the ongoing work of high-level planning in the Xperience project. In particular, this document reports on the current state of an application programming interface (API) which provides an abstract specification of common planning activities: planner configuration, domain definition, plan generation, and plan iteration. Although this interface is currently implemented using PKS as its backend planning system, it is designed to be generic and any planner which supports the API can be used in its place as an alternative backend. This document presents a first snapshot of the API and its functions, which may be updated and extended in the future.

Revision history

2013-12-18 : An initial report presenting a snapshot of the current application programming interface (API) for high-level planning, defining a set of abstract planning services that are implemented by an underlying planning system (currently PKS).

Contents

1 Introduction

In this document we focus on the state of high-level planning work by UEDIN in the Xperience project. This work forms part of WP3 (Generative Mechanisms) and, in particular, WP3.2 (Structural Bootstrapping for Planning). The present report primarily addresses Task 3.2.3 (Plan structure and execution) and Task 3.2.4 (Extended reasoning about object and indexical knowledge) and presents a snapshot of the current *application programming interface (API)* that is used to integrate high-level planning (currently the PKS planner) with robot platforms in the Xperience project (and beyond). This work is also closely related to WP4 (Interaction and Communication) and the project-wide integration work and demonstrations of WP5 (System Integration).

High-level planning capabilities in the Xperience project are currently supplied by the PKS planner [Petrick and Bacchus, 2002, 2004], which UEDIN is extending for use in robotic and linguistic domains as part of WP3 (with some connections to WP4). PKS is a state-of-the-art knowledge-level planner that constructs plans in the presence of incomplete information. Unlike traditional planners, PKS builds plans at the "knowledge level", by representing and reasoning about how the planner's knowledge state changes during plan generation. Actions are specified in a STRIPS-like [Fikes and Nilsson, 1971] manner in terms of action preconditions (state properties that must be true before an action can be executed) and action effects (the changes the action makes to properties of the state). PKS is able to construct conditional plans with sensing actions, and supports numerical reasoning, run-time variables [Etzioni et al., 1992], and features like functions that arise in real-world planning scenarios.

Like most AI planners, PKS operates best in discrete, symbolic state spaces described using logical languages. As a result, work that addresses the problem of integrating planning on real-world robot platforms often centres around the problem of representation, and how to abstract the capabilities of a robot and its working environment so that it can be put in a suitable form for use by a goal-directed planner. Integration also requires the ability to communicate information between system components. Thus, the design of a planning system often has to take into consideration external concerns, to ensure proper interoperability with modules that aren't traditionally considered in pure theoretical planning settings.

At a purely programming level, the task of integrating the planner on a robot platform (or other complex system) relies on providing a suitable interface to the underlying planning capabilities that are required. This involves providing appropriate methods for manipulating domain representations, to improve our ability to model real-world problems at the planning level, as well as functions for controlling certain aspects of the the plan generation process itself (e.g., selecting goals, generation strategies, or planner-specific settings). Moreover, functions that allow plans to be manipulated as first-class entities are useful when considering multiple behaviour strategies, or when replanning is used as a means of recovery from unexpected changes in the world.

Overall, the set of functions defined by this API can be thought of as an interface to a series of abstract *planning services* which are ultimately implemented by some underlying "black box" planning system. As with other types of complex software components, such an interface removes the need for the application programmer to know about how such services are actually implemented within the black box, but instead allows the designed to build more complex modules that simply make use of these services. As a result, this interface is designed to be generic and is not tied to any one platform (or project).

In the remainder of this document we will present a brief overview of the API we have developed for providing a specification of common planning activities, which we believe should aid in the task of integrating high-level planning with robot platforms.

2 An Application Programming Interface (API) for planning services

The ability to reason and plan is essential for an intelligent agent acting in a dynamic and incompletely known world—such as the robot scenarios we consider on Xperience. Achieving goals under such conditions often requires complex deliberation that cannot easily be achieved by simply reacting to a situation without considering the long term consequences of a course of action.

In order to facilitate the task of providing planning services to more complex systems (e.g., robot platforms), we have created an *application programming interface (API)*, which abstracts many common planning tasks into a series of functions which can be called by clients that require such services. The current set of functions is shown in Figure 1, which groups the services into a variety of categories, and also provides some additional support structures.[1] We discuss the main components of the interface below.

2.1 Properties and states

A structure called `StateProperty` defines the abstract notion of a domain property (or feature, fluent, relation, function, etc.) as an entity with a `name`, a list of arguments `args`, a `sign`, and a `value`. This definition is meant to accommodate both relational and functional entitles which commonly arise in planning states. For instance, a relation like $\neg F(a)$ could be encoded as:

```
name     :  F
args[0]  :  a
sign     :  false
value    :  (unused),
```

while a function mapping like $f(a) = c$ could be encoded as:

```
name     :  f
args[0]  :  a
sign     :  true
value    :  c.
```

A state can be thought of as simply a list of `StateProperty` definitions, denoted in the API as `StatePropertyList`. Note that this definition supports the standard STRIPS-style view of states as collections of instantiated properties, and is consistent with many types of planning approaches. It can also be used to encode the notion of an *observed state*, as a collection of properties as returned from a set of sensors.

2.2 Plan steps and plan sequences

A `PlanStep` can be thought of as a particular instantiated action in a plan, which is defined by a structure specifying its `name`, its `type`, and a list of parameters, `args`. For instance, an instantiated action `pickup(blockA, table, lefthand)` (pick up `blockA` from the `table` using the `lefthand`), which denotes a type of manipulation action, could be encoded in this structure as:

```
name     :  pickup
type     :  manipulation
args[0]  :  blockA
args[1]  :  table
args[2]  :  lefthand.
```

The `type` field is not often used by many traditional "linear" (i.e., non-hierarchical) planners, but may provide useful heuristic information to an execution system at run time.

[1] The API is presented in a code-like syntax which is similar to C++. We will not focus on the precise form of the implementation in this report.

A `PlanStepList` is simply a sequence of `PlanStep`s (i.e., instantiated actions) which can also be thought of as a simple linear plan, of the form that most classical planners are able to generate. This allows the possibility of providing a generic container for returning plans to external modules that does not rely on the particular plan encoding used by the underlying planning system. More complex plans (e.g., contingent plans involving branches, or programs involving loops) could be encoded using standard containers (e.g., trees, maps, etc.) found in most modern programming languages (e.g., STL containers in C++).

2.3 Planner configuration and debugging

The first set of functions in the planning API provide a planner-independent way of configuring the underlying planning system, and providing access to certain features needed for debugging:

- `reset()`: This function resets the planner to its initial state.
- `getPlannerProperty(string s)`: This function returns the state of the planner property variable `s`. The precise set of accessible properties is defined by the underlying planner.
- `setPlannerProperty(string s, string t)`: This function sets the state of planner property `s` to value `t`. The precise set of accessible properties and associated values is defined by the underlying planner.
- `getInternalStructure(string s)`: This function is a hook to allow internal planning structures to be queried by external modules. This is primarily included to provide access to internal debugging information.

In general, the implementation of these functions relies on such features being supported by the underlying planner. Since many planners offer such functionality already, these functions simply standardise the interface.

2.4 Domain configuration

The next set of functions provide the main methods for defining planning domain models to the planning system. These functions provide support for loading predefined models, or incrementally augmenting existing models at runtime:

- `clearDomain()`, `clearActions()`, `clearProblems()`, `clearStates()`: These functions direct the planner to delete any domain (similarly, actions, problems, or states) that are currently defined.
- `loadDomain(string s)`: This function directs the planner to load a domain from the specified file/URL `s`. The actual format of the domain is specified by the backend planner.
- `loadSymbols(string s)`: This function directs the planner to load a set of symbol definitions from the specified file/URL `s`. Symbol definitions typically involve a specification of the allowable objects, types, and properties in a planning domain.
- `loadActions(string s)`: This function directs the planner to load a set of action definitions from the specified file/URL `s`. Actions are defined in a language supported by the backend planner.
- `loadProblems(string s)`: This function directs the planner to load a set of problem definitions from the specified file/URL `s`. A problem definition typically consists of initial state and goal specifications, but may also contain additional problem constraints or control information. Again, the precise form of a planning problem is specified by the backend planner.

- `loadPlanState(string s)`: This function allows the planner to load a cache a state definition from the specified file/URL `s`. Such a state can be used by the planner as a starting state, or a possible recovery state for replanning purposes. The only hard requirement this function imposes on the backend planner is that this state be cached for future use.
- `loadObservedState(string s)`: This function is similar to `loadPlanState` except the loaded state is additionally tagged as being an observed state. The only hard requirement this function imposes on the backend planner is that this state be cached for future use.
- `defineDomain(string s)`, ..., `defineObservedState(string s)`: These functions are analogous to the functions `loadDomain(string s)`, ..., `loadObservedState(string s)`, as defined above, except rather than loading definitions from a specified file or URL, the definitions are directly included in the parameter string `s`. These functions allow all domain definitions to be performed directly through function calls, without requiring access to external files.
- `definePlanStateFromList(StatePropertyList s)`, `defineObservedStateFromList(StatePropertyList s)`: These functions are similar to their counterparts `definePlanState` and `defineObservedState`, except rather than specifying a state definition in a string `s`, it is defined using the state structure `StatepropertyList`, as described above.

One of the important ideas behind these functions is that they offer the possibility of specifying domains to the planner incrementally, using function calls alone, rather than specifying a single monolithic domain file to the planner as a single entity, as is usual for many off-the-shelf planners from the planning community. This means that an initial domain could be specified and then later revised, for instance due to additional information discovered by an external learning process (e.g., new domain objects, revised action descriptions, additional properties corresponding to new capabilities of the robot, etc.). This is a potentially powerful mechanism, however, it pushes the problem of how a planner should react to a change in the planning domain onto the planner itself. Conceptually, this may present problems for the underlying planner, especially in the presence of partially built plans, and this API offers no solution to this problem.

2.5 Plan generation and plan iteration

The final set of functions defined in the API specify methods for controlling various aspects of the plan generation process, and for iterating through generated plans:

- `buildPlan()`: This function directs the planner to generate a plan using the current settings, domain, and default planning problem.
- `clearPlan()`: This function directs the planner to clear the current plan in its memory, if one exists.
- `getCurrentPlan()`: This function directs the planner to return the current plan as a string. This function is normally used to direct the planner to return a plan in its native format.
- `getCurrentPlanAsList()`: This function directs the planner to return the current plan as a `PlanStepList` structure. As a result, this function currently only supports plans that can be returned as a linear sequence of actions.
- `getNextAction()`: This function directs the planner to return the next action in a plan, as a `PlanStep` structure.
- `getNextActionUsingControlInfo()`: This function is similar to `getNextAction` except it allows for the specification of additional control information in the parame-

ter string `s`. This information is intended to help resolve plan ambiguities concerning execution decisions (e.g., which branch of a plan should be followed, whether a loop termination decision has been achieved). The precise form of the control information is planner dependent.

- `isNextActionEndOfPlan()`: This function determines whether we have reached the end of the plan during plan iteration.
- `isPlanDefined()`: This function returns a status update on whether or not a valid plan currently exists.
- `setProblem(string s)`: This function informs the planner that it should work with the planning problem specified by the string `s`. The string may specify a label to a previously defined problem, or contain the definition of a new problem.
- `setProblemGoal(string s)`: This function informs the planner that the current goal condition should be replaced by the goal specified by the string `s`. The problem is otherwise unchanged.

The idea behind many of these functions is to extend a degree of control over the plan generation and execution processes, as necessary, to components outside the planner itself, to the extent that simple plan execution monitoring activities can be supported without reliance on the planner. As a result, a client using these services can determine whether to generate a plan, and can iteratively ask for individual plan steps, advancing the plan one step at a time. Entire plans can also be processed by external processes in their entirety. The functions also support run-time updates to certain aspects of the planning problem, such as goal change.

3 Implementation

This document is not meant to provide precise details concerning the implementation of the API, however, we note the following design decisions which affect our current implementation:

- **Internet Communications Engine (ICE)**: The API in Figure 1 is implemented using the Internet Communications Engine (`http://www.zeroc.com/ice.html`), which provides an object-oriented middleware for building distributed applications. The default implementation using the PKS planner provides a planning server which allows clients to to access the services provided by the API.
- **Support for multiple backends**: The current implementation of the planning API was adapted from the interface to the PKS planner, but has been abstracted to avoid PKS-specific representations and syntax. API functions connect the ICE layer to a version of PKS implemented as a C++ library, which is linked to form the plan server. However, there is no strict requirement that PKS must be used as the planning backend, and any planner which is able to implement the API can be used in its place as an alternative backend.
- **Backend-dependent syntax**: Many of the functions in our API require specifying a file or a definition for a particular aspect of the planning domain (e.g., actions, problems, symbols, etc.). The precise syntactic form of these definitions is left to the backend planner. As a result, this means that the content (parameters) of certain function calls may change from backend to backend. However, this also means that all planning domain entities do not need to be standardised in order to support the API. This functionality may change in the future as we reconsider certain aspects of the interface.

More details of the implementation may be included in future versions of this report.

```
1     // Abstract property definition
2     struct StateProperty {
3         string name;
4         StringList args;
5         bool   sign;
6         string value;
7     };
8
9     // Abstract state definition
10    sequence<StateProperty> StatePropertyList;
11
12    // Abstract plan step definition
13    struct PlanStep {
14        string name;
15        string type;
16        StringList args;
17    };
18
19    // Abstract plan sequence definition
20    sequence<PlanStep> PlanStepList;
21
22    interface PlannerController {
23        // Configuration and debugging
24        void         reset();
25        string       getPlannerProperty(string s);
26        bool         setPlannerProperty(string s, string t);
27        string       getInternalStructure(string s);
28
29        // Domain configuration
30        void         clearDomain();
31        void         clearActions();
32        void         clearProblems();
33        void         clearStates();
34
35        bool         loadDomain(string s);
36        bool         loadSymbols(string s);
37        bool         loadActions(string s);
38        bool         loadProblems(string s);
39        bool         loadPlanState(string s);
40        bool         loadObservedState(string s);
41
42        bool         defineDomain(string s);
43        bool         defineSymbols(string s);
44        bool         defineActions(string s);
45        bool         defineProblems(string s);
46        bool         definePlanState(string s);
47        bool         defineObservedState(string s);
48        bool         definePlanStateFromlist(StatePropertyList s);
49        bool         defineObservedStateFromList(StatePropertyList s);
50
51        // Plan generation and plan iteration
52        bool         buildPlan();
53        void         clearPlan();
54        string       getCurrentPlan();
55        PlanStepList getCurrentPlanAsList();
56        PlanStep     getNextAction();
57        PlanStep     getNextActionUsingControlInfo(string s);
58        bool         isNextActionEndOfPlan();
59        bool         isPlanDefined();
60        bool         setProblem(string s);
61        bool         setProblemGoal(string s);
62    };
```

Figure 1: An API for high-level planning services

4 Future directions and final notes

In this section we briefly note some directions for this work that are currently under investigation. Some of these tasks are at a very early stage and may not be fully active until a later stage of the project.

- This report is not meant to provide comprehensive documentation for the planning API or ICE but, instead, is meant to provide a common basis for future discussions concerning this interface, especially with regard to integration activities involving alternative (i.e., non-PKS) planners. (Code-level documentation may be provided at a later time.) It is important to note that this interface evolved naturally from the pre-existing PKS interface and, as a result, will continue to be used with PKS in some form, regardless of any future project-specific decisions that are taken.
- The API we presented is a snapshot of our current implementation and is subject to change. In particular, as we attempt to integrate new planners using this interface, subtle changes may be required to accommodate new features, alternative state representations, or different action models. However, we note that the existing interface is more than suitable for the needs of PKS and has been success fully used to integrate the planner on multiple robot platforms, including some beyond the Xperience project. In fact, one of the strengths of the current API is that it is not tied to any one project or robot platform; instead it simply provides the abstract planning interface from which one could build the necessary interfaces to a range of (robot) platforms. Thus, it is important to note that extra layers of abstraction may be needed to properly integrate a planner on a specific robot platform: this API is meant to address the problem of abstracting *planning features*, which is a necessary first step towards wider integration as part of a complex system.
- The current API does not include any direct references to probabilistic representations, temporal constraints, or cost-based encodings. We are exploring extensions to include these ideas as native concepts in our API.
- Unlike many off-the-shelf planners, our API doesn't rely on text files as the main interface to the planner. As a proof-of-concept example of the genericity of our interface, we plan to adapt the interface to a PDDL-based planner, to show it can work with our interface. As a first step, we may address this problem by writing a new interface layer that works with ordinary PDDL files, before adapting the interface to the planner directly.
- The current API provides limited features for iterating through plans, in a manner that is external to a plan execution monitor. We are considering extensions to this interface that provide the necessary functionality for processing different types of plans (linear, branching, looping), so that an external controller could be built, using these functions.
- One part of the API that isn't (currently) constrained, is the content of many of the domain configuration functions, which simply require a string as a parameter. Since the precise syntactic definition of certain planning concepts often differs greatly between planners, this interface offers the possibility of each planner interpreting how a domain element should be defined, given the backend planner. This situation isn't completely ideal, however, and we are in the process of designing a more well-structured interface for these operations.

Acknowledgements

Special thanks go to Nils Adermann whose ideas helped shape this API. Discussions with Nils helped motivate the need for methods that support incremental domain definition, which lead to interesting questions (which haven't completely been answered yet) on how best a planner should work with partially-defined domain models.

References

O. Etzioni, S. Hanks, D. Weld, D. Draper, N. Lesh, and M. Williamson. An approach to planning with incomplete information. In W. Swartout, B. Nebel, and C. Rich, editors, *Proceedings of the Third International Conference on Principles of Knowledge Representation and Reasoning (KR-92)*, pages 115–125, Cambridge, MA, Oct. 1992. Morgan Kaufmann Publishers.

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

R. P. A. Petrick and F. Bacchus. A knowledge-based approach to planning with incomplete information and sensing. In M. Ghallab, J. Hertzberg, and P. Traverso, editors, *Proceedings of the Sixth International Conference on Artificial Intelligence Planning and Scheduling (AIPS-2002)*, pages 212–221, Menlo Park, CA, Apr. 2002. AAAI Press.

R. P. A. Petrick and F. Bacchus. Extending the knowledge-based approach to planning with incomplete information and sensing. In S. Zilberstein, J. Koehler, and S. Koenig, editors, *Proceedings of the International Conference on Automated Planning and Scheduling (ICAPS-04)*, pages 2–11, Menlo Park, CA, June 2004. AAAI Press.

Approximate Reasoning with Numeric Fluents for Contingent Planning

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

Abstract

We investigate the problem of reasoning about numeric fluents in the presence of incomplete information, sensing, and contingent plans. An interval-based representation is used to compactly represent sets of possible values for a numeric fluent, as an approximation of the true value of the fluent. We demonstrate how such information can be used to model uncertain sensors and effectors, and show how such fluents form a type of noisy run-time variable in plans. A proof-of-concept implementation using the PKS (Planning with Knowledge and Sensing) planner is given, with examples taken from a simple robot domain.

Introduction and Motivation

An agent operating in a real-world domain often needs to do so with *incomplete information* about the state of the world. An agent with the ability to *sense* the world can also gather information to generate plans with *contingencies*, allowing it to reason about the outcome of sensed data at plan time.

One useful type of sensed information is *numerical* information, which is often required to build plans that work with numeric state properties (e.g., the robot is 10 metres from the wall), limited resources (e.g., ensure the robot has enough fuel), numeric constraints (e.g., only grasp an object if its radius is less than 10 cm), or arithmetic operations (e.g., advancing the robot one step reduces its distance from the wall by 1 metre). The importance of numerical reasoning in planning has been recognised with the inclusion of numeric state variables in PDDL (Fox and Long 2003), and in planners like MetricFF (Hoffmann 2003).

Reasoning about numerical information under conditions of incomplete information is often problematic, however, especially when planners use possible-world or belief state representations. In such representations, the set of possible values for an incompletely known state property is often explicitly represented, e.g., by a set of states, each of which denotes a possible configuration of the actual world state. If the value of a *numeric fluent* is unknown, then the belief state must contain a state representing every possible mapping of the fluent, which could be a potentially large (or even infinite) set. Even when the range of possible values is small, the number of required states can quickly grow. E.g., if a fluent f could map to any natural number between 1 and 100, then we require 100 states to capture f's possible mappings. The state explosion resulting from large sets of mappings can be computationally difficult for planners that must reason with individual states to construct plans.

The problem of reasoning about knowledge and action without possible worlds has been studied in languages like the situation calculus (Demolombe and Pozos Parra 2000; Soutchanski 2001; Liu and Levesque 2005; Petrick 2006; Vassos and Levesque 2007). Many of these accounts model restricted types of knowledge directly, rather than indirectly inferring knowledge from sets of worlds, trading representational expressiveness for tractable reasoning. One representation for modelling uncertain numerical information without possible worlds uses the notion of *interval-valued fluents* (Funge 1998). The idea is simple: instead of representing each possible mapping by a separate state, a single interval mapping is used, where the endpoints of the interval indicate the fluent's range of possible values. Thus, a fluent f that could map to values between 1 and 100 can be denoted in interval-valued form as $f = \langle 1, 100 \rangle$.

Interval-valued numeric models have been previously investigated in planning contexts, e.g., for modelling time as a resource (Edelkamp 2002; Frank and Jónsson 2003; Laborie 2003). A similar representation to ours for bounding uncertain numeric properties has also been proposed by Poggioni, Milani, and Baioletti (2003). This idea also has parallels to work on *register models* (van Eijck 2013).

In this paper, we explore numerical reasoning in the context of incomplete information, sensing, and contingent planning, which to the best of our knowledge has not been previously explored. Our proof-of-concept demonstration of this approach is implemented as a set of extensions to the PKS planner (Petrick and Bacchus 2002; 2004). In particular, building on PKS's existing ability to work with limited numerical information (e.g., function (in)equalities and arithmetic operations), we use interval-valued fluents to provide a compact means of modelling noisy actions and incomplete knowledge, without possible worlds or belief states. We also introduce a mechanism for progressing certain types of noisy sensed information through physical actions, by treating it as a type of interval-valued "run-time variable" (Etzioni et al. 1992). For instance, if the (noisy)

position of a robot is sensed and the robot then moves 2 steps forward, we should still be able to bound the robot's current location, even if the location isn't explicitly known. We illustrate these extensions with a set of example problems (described below), taken from a simple robot domain.

Planning with Knowledge and Sensing (PKS)

PKS (Planning with Knowledge and Sensing) is a contingent planner that builds plans in the presence of incomplete information and sensing actions (Petrick and Bacchus 2002; 2004). PKS works at the *knowledge-level* by reasoning about how the planner's knowledge state, rather than the world state, changes due to action. PKS works with a restricted subset of a first-order language, and a limited amount of inference, allowing it to support a rich representation with features such as functions and variables. This approach differs from planners that work with possible worlds models or belief states. By working at the knowledge level, PKS does not have to reason in terms of individual worlds. However, as a trade-off, its restricted representation means that certain types of knowledge cannot be directly modelled.

PKS is based on a generalisation of STRIPS (Fikes and Nilsson 1971). In STRIPS, the state of the world is modelled by a single database. Actions update this database and, by doing so, update the planner's world model. In PKS, the planner's knowledge state, rather than the world state, is represented by a set of five databases, each of which models a particular type of knowledge. The contents of these databases have a fixed, formal interpretation in a modal logic of knowledge. Actions can modify any of the databases, which has the effect of updating the planner's knowledge state. To ensure efficient inference, PKS restricts the type of knowledge (especially disjunctions) that it can represent:

$\mathbf{K_f}$: This database is like a STRIPS database except that both positive and negative facts are permitted and the closed world assumption is not applied. K_f is used for modelling action effects that change the world. K_f can include any ground literal ℓ, where $\ell \in K_f$ means "the planner knows ℓ." K_f can also contain known function (in)equality mappings.

$\mathbf{K_w}$: This database models the plan-time effects of "binary" sensing actions. $\phi \in K_w$ means that at plan time the planner either "knows ϕ or knows $\neg\phi$," and that at execution time this disjunction will be resolved. In such cases we will also say that the planner "knows whether ϕ." Know-whether knowledge is important since PKS can use such information to construct conditional plans with branches (see below).

$\mathbf{K_v}$: This database stores information about function values that will become known at execution time. In particular, K_v can model the plan-time effects of sensing actions that return constants, such as numeric values. K_v can contain any unnested function term f, where $f \in K_v$ means that at plan time the planner "knows the value of f." At execution time the planner will have definite information about f's value. As a result, PKS is able to use K_v terms as "run-time variables" (Etzioni et al. 1992) or placeholders in its plans.

$\mathbf{K_x}$: This database models the planner's "exclusive-or" knowledge of literals, namely that the planner knows "exactly one of a set of literals is true." Entries in K_x have the form $(\ell_1|\ell_2|\ldots|\ell_n)$, where each ℓ_i is a ground literal. Such formulae represent a particular type of disjunctive knowledge that is common in many planning scenarios, namely that "exactly one of the ℓ_i is true."

$\mathbf{LCW}$: This database stores the planner's "local closed world" information (Etzioni, Golden, and Weld 1994), i.e., instances where the planner has complete information about the state of the world. We will not use LCW in this paper.

PKS's databases can be inspected through a set of *primitive queries* that ask simple questions about the planner's knowledge state, namely whether facts are (not) known to be true, whether function values are (not) known, or if the planner "knows whether" certain properties are true or not. An inference algorithm evaluates primitive queries by checking the contents of the databases, taking into consideration the interaction between different knowledge in the databases.

An action in PKS is modelled by a set of *preconditions* that query the agent's knowledge state, and a set of *effects* that update the state. Action preconditions are simply a list of primitive queries. Action effects are described by a collection of STRIPS-style "add" and "delete" operations that modify the contents of individual databases. E.g., *add*(K_f, ϕ) adds ϕ to K_f, and *del*(K_w, ϕ) removes ϕ from K_w. Actions can also have ADL-style context-dependent effects (Pednault 1989), and can use a form of quantification.

PKS constructs plans by reasoning about actions in a simple forward-chaining manner: if the preconditions of an action are satisfied by the planner's knowledge state, then the action's effects are applied to the state to produce a new knowledge state. Planning then continues from the resulting state. PKS can also build plans with branches, by considering the possible outcomes of its K_w knowledge. Planning continues along each branch until it satisfies the *goal* conditions, also specified as a list of primitive queries.

Functional Fluents, Intervals, and Knowledge

In this paper, we will only focus on *functional fluents* (hereafter, a *function*) that map to numerical values, rather than general constants or terms. An *interval-valued fluent (IVF)* is a function whose denotation is an *interval* of the form $\langle u, v\rangle$. The values u and v are called the *endpoints* of the interval, and indicate the bounds on a range of possible mappings for the fluent. Since we are primarily interested in reasoning about an agent's (incomplete) knowledge during planning, a mapping of the form $f = \langle u, v\rangle$ will mean that the value of f is known to be in the interval $\langle u, v\rangle$. For instance, the IVF mapping *robotLoc* $= \langle 5, 10\rangle$ might indicate that the distance to a wall is known to be between 5 and 10 metres. If a fluent maps to a *point interval* of the form $\langle u, u\rangle$, for some u, then the mapping is certain and known to be equal to u.

Each IVF is associated with a particular *number system* $\mathbb{X}$ that restricts the range of permissible intervals for a fluent. Typically, the number system will be one of the standard mathematical number systems (e.g., the reals $\mathbb{R}$, the natural numbers $\mathbb{N}$, the integers $\mathbb{Z}$, etc.), extended to include the points at infinity, ∞ and $-\infty$. Given a number system $\mathbb{X}$, a mapping $f = \langle u, v\rangle$ is permitted, provided $u, v \in \mathbb{X}$ and $u \leq v$. For every number system $\mathbb{X}$, the special interval

$\langle \bot, \top \rangle$ represents the *maximal interval* for that number system. For instance, $\langle \bot, \top \rangle \stackrel{\text{def}}{=} \langle -\infty, \infty \rangle$ in $\mathbb{R}$, however in $\mathbb{B}$, the binary number system consisting of the two elements 0 and 1, $\langle \bot, \top \rangle \stackrel{\text{def}}{=} \langle 0, 1 \rangle$. In terms of knowledge, a mapping of the form $f = \langle \bot, \top \rangle$ means that the agent considers every element of $\mathbb{X}$ as a possible denotation for f. In other words, the value of f is completely unknown to the agent. (For simplicity, we will assume that all interval-valued fluents in this paper range over $\mathbb{N}$ unless otherwise indicated.)

It is often useful to interpret an IVF in terms of a possible worlds representation. For instance, if W^* is a set of worlds modelling an agent's knowledge state, and $W^* \models f = \langle u, v \rangle$ (i.e., all values between u and v are considered as possible mappings for f), then for each x, $u \leq x \leq v$, it must be the case that there exists a world $w \in W^*$ such that $w \models (f = x)$. In other words, a world must exist for each possible mapping of f. In this view (abuse of notation aside), IVFs can be seen as a compact means of representing a set of possible worlds, at least relative to a particular fluent mapping. It is this observation that we will use as the basis for our planning representation, to replace the possible worlds model and instead work directly at the IVF level.

Interval-Valued Knowledge in PKS

The addition of IVFs potentially affects a planner's representation, reasoning, and planning components. In PKS, we consider changes to the planner's databases, query language, and action representation arising from IVFs.

Representing definite and indefinite interval knowledge The first question that we must address is where to represent IVF information in PKS. The most commonly used PKS database is K_f which stores the planner's knowledge of facts, including functional equalities (e.g., $f = 10$) and inequalities (e.g., $g \neq 12$). In this view, K_f is responsible for modelling *definite* knowledge about the world. However, IVFs are designed to model *indefinite* information, namely a disjunctive set of epistemic alternatives for an underlying function mapping. Thus, K_f is not a perfect fit for IVFs since it requires a change of semantics to properly interpret IVFs.

Instead, we use the K_x database to store the planner's knowledge of (general) IVFs. K_x is normally responsible for modelling the planner's "exclusive-or" information, where a discrete number of alternatives are presented and only one can be true. (E.g., if $(\ell_1|\ell_2|\ell_3) \in K_x$ then the planner knows that one, and only one, of ℓ_1, ℓ_2, or ℓ_3 can hold.) IVFs provide a generalisation of this concept for numerical functions and so we extend K_x to permit IVFs of the form $f = \langle u, v \rangle$. We note that only IVFs that contain numeric constants are allowed, and point intervals are not permitted. That is, a fluent like $f = \langle 5, 10 \rangle$ is allowed in K_x, but $g = \langle 5, 5 \rangle$ and $h = \langle 5, x \rangle$ are not, where x is a variable. Intuitively, a fluent of the form $f = \langle u, v \rangle \in K_x$ means that f is known to map to a single value between u and v.

An IVF can also be part of an ordinary K_x formula $(\ell_1|\ell_2|\ldots|\ell_n)$, where each ℓ_i can be a ground literal or an interval mapping. In general, such formulae are treated as standard K_x formulae and are subject to PKS's conservative update rules. (We refer the reader to (Petrick and Bacchus 2004) for more details.) In particular, physical actions that change any function or relation mentioned in a K_x formula cause that formula to be completely removed from K_x since it's "exclusive-or" property may no longer hold.

One interesting consequence of such formulae is the case where all the ℓ_i are IVFs with the same underlying function. In this case, such a formula can be used to model *disjunctive intervals*, i.e., sets of disjoint interval mappings. For instance, if a fluent f could possibly map to any value between 5 and 10 or, additionally, map to values between 15 and 18, we can represent such information by the K_x formula $(f = \langle 5, 10 \rangle | f = \langle 15, 18 \rangle)$. We could not represent such knowledge using a single IVF alone since the "holes" in the interval would also be included. (I.e., $f = \langle 5, 18 \rangle$ would cover the necessary interval but also admit values in the range 10–18.) Such formulae are updated in a different way than ordinary K_x formulae, where we attempt to track the possible values of the IVF through action (see below).

We also permit certain IVFs references in K_f, in keeping with K_f's standard use of functional equality and inequality mappings. In particular, function mappings involving point intervals are allowed. Thus, $g = \langle 5, 5 \rangle$ (equivalent to $g = 5$ in usual function notation) is permitted, meaning the planner knows that g is equal to 5. An assertion like $h \neq \langle 10, 10 \rangle$ is also allowed, meaning h is known to be unequal to 10.

Finally, the exclusion of an IVF from all PKS databases means that the planner has no information about that IVF. In other words, it is treated as if it has a maximal interval mapping in which all denotations are considered possible.

Interval-based sensors The K_v database is primarily used to represent the results of sensing actions that return functions. For an ordinary function f, a formula $f \in K_v$ means that the value of f will become known to the planner at execution time. At plan time, f can be used as a "run-time variable" (Etzioni et al. 1992) that acts as a placeholder to the actual sensed valued of the function. We extend this representation to include IVFs in K_v. Thus, an IVF $f \in K_v$ means that the point value of f will become known at execution time, and f can be used as a run-time variable denoting the sensed point mapping of f. In other words, IVFs are treated the same as ordinary functions.

However, we also extend our notion of K_v knowledge to allow *noisy* sensed information to be modelled. In this case, we specify an *interval schema* for the associated IVF, using a variable (x in our examples) to denote the actual value of the fluent. For instance, a fluent of the form:

$$f : \langle x - 1, x + 1 \rangle \in K_v$$

means that the value of the fluent f is known, and f is in the range $x \pm 1$, for some x. The value of f in this case is "noisy" as it admits a range of possible values. In practice, we allow formulae of the form $f : \langle x - u, x + v \rangle$ in K_v, where u and v are numeric constants. This type of information is particularly useful for tracking changes to sensed numeric values through the effects of certain physical actions, as we will see in one of the examples below.

Comparing numeric intervals The K_w database is used to model sensing actions with binary outcomes, i.e., those

that return one of two possible values. With numeric fluents (interval or not), certain types of numeric relations become useful in a planning context. In particular, the relational operators $=, \neq, >, <, \geq$, and $\leq$ often arise in many planning scenarios. In our extended version of PKS, we allow simple formulae using such operators to be explicitly represented in K_w, provided such formulae have the form f **op** c, where **op** $\in \{=, \neq, >, <, \geq, \leq\}$ and c is a numeric constant. Thus, $f > 5 \in K_w$ can be used to model a sensing action that determines whether f is greater than 5 or not.

K_w is also important since information in this database can be used to build contingent branches into a plan: one branch is added for each possible outcome of the sensed information. In our extended version of PKS, we allow branches that reason about the range of an IVF, using the numeric relations described above. The process by which branches are added to a plan is given below.

Querying interval knowledge The primitive query language used by PKS allows the planner to answer certain questions about the planner's databases. In particular, primitive queries have the following form: (i) Kp, is p known to be true?, (ii) $K_v t$, is the value of t known?, (iii) $K_w p$, is p known to be true or known to be false? (i.e., does the planner know-whether p?), or (iv) the negation of queries (i)–(iii). With the addition of IVFs, we extend the query language to interval-valued numerical relations and functions.

Standard PKS permits queries in (i) and (iii) (and their negations) with numeric relations of the kind we considered in the extended K_w database (e.g., f **op** c, where c is a constant). However, evaluating such queries in the presence of IVFs involves reasoning about the endpoints of known intervals, and possibly the underlying number systems. For instance, a query $K(f > 3)$ only evaluates as true given an interval $f = \langle u, v\rangle \in K_x$ provided $u > 3$. Similarly, a query $K(g \neq 5)$ is true if both $5 < u$ and $5 > v$. Evaluating queries for the other allowable numeric relations is also straightforward, given the restricted form of those relations.

Queries of the kind in (ii) involving functions are also permitted for IVFs. In this case, they are evaluated in the same way as ordinary functions. In other words, a query $K_v(f)$ (is the value of the IVF f known?) evaluates as true provided $f \in K_v$ or a point interval is known, i.e., $f = \langle c, c\rangle \in K_f$.

Actions and database progression Actions provide the primary means of progressing a set of databases and are defined in a similar way to ordinary PKS actions, with preconditions and effects. Preconditions are simply sets of primitive queries, as defined above. Effects permit updates to individual databases, with references to IVFs limited to a set of simple arithmetic operations. (We only consider the arithmetic addition and subtraction operators here.) In particular, we allow updates of the form *add*$(K_f, f := f \pm d)$, where f is an IVF and d is either a numeric constant or constant interval (i.e., no variables). This gives rise to a simple procedure for updating IVFs across the set of databases:

1. If f is in K_f with a point interval and d is a numeric constant, update f by adding or subtracting d as appropriate.
2. If f has an interval mapping $\langle u, v\rangle$ in K_x and d is a numeric constant, update f so that $f = \langle u \pm d, v \pm d\rangle$.

PlanPKS$^+$(**DB**, P, G):
  if *goalsSatisfied*(**DB**, G) **then return** P
  else
    pick$(a \in A)$: *precondsSatisfied*$(a,$ **DB**$)$
    applyEffects$(a,$ **DB**, **DB**$')$ *and resolve intervals*
    return *PlanPKS*$^+$(**DB**$', (P, a), G)$.
  or
    pick(α) : α is a ground instance of an entry in K_w
    form new branch roots and resolve intervals
    branch(**DB**, α, **DB**$_1$, **DB**$_2$)
    $C :=$ {*PlanPKS*$^+$(**DB**$_1, \emptyset, G$), *PlanPKS*$^+$(**DB**$_2, \emptyset, G$)}
    return P, C.

Table 1: Extended PKS planning algorithm

3. If f is in K_f or K_x and d is an interval constant $\langle a, b\rangle$, update f so that $f = \langle u \pm a, v \pm b\rangle$. If f was initially in K_f, remove it and put the updated mapping in K_x.
4. If f is a disjunctive interval $(\ell_1|\ell_2|\ldots|\ell_n) \in K_x$, update each ℓ_i as in step 3, above.
5. If f is mentioned in an ordinary K_x formula ϕ, then remove ϕ from K_x.
6. If f is an interval schema $\langle x - u, x + v\rangle$ in K_v, update f to map to $\langle x - u \pm d, x + v \pm d\rangle$ if d is a numeric constant, or $\langle x - u \pm a, x + v \pm b\rangle$ if d is an interval $\langle a, b\rangle$.

We note that even though the above updates are triggered by an *add* rule that references the K_f database, its actual effects may update K_v or K_w, in addition to K_f. We also note that the updated interval calculation in steps 3 and 6 assumes an initially well-formed interval and a well-behaved number system like $\mathbb{N}$ or $\mathbb{R}$. In other number systems, the process of updating an interval might be more complicated. In such cases, a new range must be calculated for the updated IVF, ensuring a well-formed interval as a result. In step 5 we note the conservative behaviour of ordinary K_x updates, which is similar to standard PKS: a change to an IVF in an ordinary K_x formula potentially invalidates that formula so that its exclusive-or property may no longer hold.

Finally, in addition to the above numeric updates, actions can include simple database assertions (i.e., *add* and *del*) that include IVFs, provided the form of those assertions satisfies the restrictions on the knowledge that can stored in a given database. Thus, we can specify "noisy" knowledge through an update such as *add*$(K_x, f = \langle 3, 5\rangle)$ that adds $f = \langle 3, 5\rangle$ to K_x, or *del*$(K_f, f = 3)$ that removes a point interval from K_f (possibly making its value unknown).

Contingent planning and plan correctness Given the above changes to PKS's databases, primitive queries, and update mechanism, the extended planning algorithm (see Table 1) operates with very little change from the standard PKS algorithm, taking as input a set of initial databases **DB**, a set of actions A, and a goal G. The plan generation process is treated as a search through the set of database states, starting from the initial **DB**. Plans are built in a forward-chaining manner by choosing an action to add to a plan whose preconditions are satisfied in the current state (*precondsSatisfied*), or by introducing branches into the plan.

Conditional plans are formed as usual in PKS, by reasoning about the possible outcomes of K_w formulae. For a given K_w formula ϕ, which may now include IVFs, two branches

Action	Effects
moveForward	$add(K_f, robotLoc := robotLoc - 1)$
moveBackward	$add(K_f, robotLoc := robotLoc + 1)$
atTarget	$add(K_w, robotLoc = targetLoc)$
noisyForward	$add(K_f, robotLoc := robotLoc - \langle 1, 2\rangle)$
withinTarget	$add(K_w, robotLoc \leq targetLoc)$
noisyLocation	$add(K_v, robotLoc : \langle x, x + 1\rangle)$

Table 2: Action specifications for the example domains.

are added to a plan: along the positive K^+ branch ϕ is assumed to be true, while along the negative K^- branch $\neg\phi$ is assumed to be true. For instance, if a formula $f > 5 \in K_w$ is used as the basis for a new branch point then $f > 5$ is assumed to be true along the K^+ branch, and $f \leq 5$ is assumed to be true along the K^- branch. An important part of the branching process is resolving IVF knowledge resulting from K_w assumptions, by combining it with knowledge in other databases, and possibly refining it further. Space prohibits us from describing this process completely; for instance, the following are a subset of the rules for (in)equality updates, which differ from those of ordinary action updates:

1. If $f = c$ is assumed to be true along a branch, where c is a numeric constant, then add $f = c$ to K_f and remove any IVFs or disjunctive intervals from K_x.
2. If $f \neq c$ is assumed to be true along a branch, where c is a numeric constant, and $f = \langle u, v\rangle$ is an IVF in K_x where $u < c < v$, then add $f \neq c$ to K_f and update f to be a disjunctive interval $(f = \langle u, c^-\rangle \,|f = \langle c^+, v\rangle)$ in K_x where c^- (c^+) is the smallest value permitted by the number system less than (greater than) c.[1]
3. Repeat the calculation in step 2 for any disjunctive intervals in K_x where $f = \langle u, v\rangle$ is a subinterval, extending the number of disjunctive subintervals.

Other interval assertions lead to useful updates. E.g., if $f > 5$ is assumed to be true and $f = \langle 3, 10\rangle \in K_x$, then K_x is updated so that $f = \langle 6, 10\rangle$. Similarly, if $f \leq 5$ is assumed to be true then K_x is updated so that $f = \langle 3, 5\rangle$. This process gives rise to a technique for splitting intervals into smaller components that acts as a form of reasoning by cases.

Planning continues until the goal conditions are achieved along every branch of a plan (*goalsSatisfied*), or no plan is found. As a result, this process enforces a correctness criteria on the plans it generates: actions and branches are only introduced if the planner has sufficient knowledge at each step of the plan. (In particular, it is this condition that allows branches to be based on K_w information, but not K_x.)

Examples

We now illustrate our proof-of-concept PKS implementation on three examples taken from a simple robot domain.

Example 1 Consider a robot whose location, *robotLoc*, is measured by its distance to a wall. The robot has two physical actions available to it: *moveForward* moves the robot one unit towards the wall, and *moveBackward* moves the robot one unit away from the wall. The robot also has a sensing action, *atTarget*, which senses whether the robot is at a target location specified by the fluent *targetLoc*. The definition of these actions is shown in Table 1. The robot's initial location is specified by the IVF mapping $robotLoc = \langle 3, 5\rangle \in K_x$. The goal is to move the robot to a target location, i.e., $K(robotLoc = targetLoc)$, where $targetLoc = 2 \in K_f$.

One solution generated by PKS is the conditional plan:

Plan step		*robotLoc*
0.		$\langle 3, 5\rangle$
1.	*moveForward* ;	$\langle 2, 4\rangle$
2.	*atTarget* ;	
3.	*branch(robotLoc = targetLoc)*	
4.	K^+ : **nop**.	2
5.	K^- :	$\langle 3, 4\rangle$
6.	*moveForward* ;	$\langle 2, 3\rangle$
7.	*atTarget* ;	
8.	*branch(robotLoc = targetLoc)*	
9.	K^+ : **nop**.	2
10.	K^- :	3
11.	*moveForward*.	2

In step 1, *moveForward* decreases the value of *robotLoc* in K_f by one unit so that $robotLoc = \langle 2, 4\rangle$. In step 2, *atTarget* senses whether $robotLoc = targetLoc$, which has the effect of adding $robotLoc = 2$ to K_w (i.e., the planner knows whether *robotLoc* is 2). In step 3, a branch point is added to the plan based on this K_w formula, allowing the plan to consider its two possible outcomes. Along one branch (step 4), $robotLoc = 2$ is assumed to be true (i.e., $robotLoc = 2$ is added to K_f) and the goal is achieved. Along the other branch (step 5), $robotLoc \neq 2$ is assumed to be true (i.e., $robotLoc \neq 2$ is added to K_f). As a result, the interval mapping for *robotLoc* in K_x is refined to remove 2 as a possible mapping, so that $robotLoc = \langle 3, 4\rangle$. The *moveForward* action then updates *robotLoc* so that $robotLoc = \langle 2, 3\rangle$. The sensing action in step 7 again adds $robotLoc = 2$ to K_w. In step 8, another branch point is added to the plan. Along one branch (step 9), $robotLoc = 2$ is assumed to true, satisfying the goal. Along the other branch (step 10), $robotLoc \neq 2$ is assumed to be true. In this case, refining *robotLoc* results in $robotLoc = \langle 3, 3\rangle = 3$, which is added to K_f. A final *moveForward* results in $robotLoc = 2$, satisfying the goal.

We note that if the initial location of the robot included a "wider" interval, e.g., $robotLoc = \langle 0, 10\rangle$, we could still possibly achieve the goal by reasoning with disjunctive intervals. For instance, if the *atTarget* action was added given the wider interval for *robotLoc*, then after branching we would have a K^+ branch where $robotLoc = 2$ and a K^- branch where $(robotLoc = \langle 0, 1\rangle \,|robotLoc = \langle 3, 10\rangle) \in K_x$. In this case, further movement actions would be needed before subsequent *atTarget* actions could resolve the K_x knowledge appropriately to satisfy the goal.

Example 2 We next consider a robot with the *moveBackward* and *atTarget* actions, but with *moveForward* replaced by a "noisy" action, *noisyForward*, which moves the robot forward either 1 or 2 units. Additionally, the robot also has a second sensing action, *withinTarget*, that determines whether or not the robot is within the target distance ($targetLoc = 2 \in K_f$). The specification of these new actions is given in Table 1. The robot's initial location is specified by the IVF $robotLoc = \langle 3, 4\rangle \in K_x$. The goal is to

[1] In practice, we may have to use open or partially open intervals here. The process is also similar for updating upper/lower bounds.

move the robot to the target, i.e., $K(\mathit{robotLoc} = \mathit{targetLoc})$.

One solution generated by PKS is the conditional plan:

Plan step		*robotLoc*
0.		$\langle 3, 4\rangle$
1.	*noisyForward* ;	$\langle 1, 3\rangle$
2.	*withinTarget* ;	
3.	*branch(robotLoc* $\leq$ *targetLoc)*	
4.	K^+ :	$\langle 1, 2\rangle$
5.	*atTarget* ;	
6.	*branch(robotLoc* $=$ *targetLoc)*	
7.	K^+ : **nop**.	2
8.	K^- :	1
9.	*moveBackward*.	2
10.	K^- :	3
11.	*noisyForward* ;	$\langle 1, 2\rangle$
12.	*atTarget* ;	
13.	*branch(robotLoc* $=$ *targetLoc)*	
14.	K^+ : **nop**.	2
15.	K^- :	1
16.	*moveBackward*.	2

Since forward movements may change the robot's position by either 1 unit or 2 units, *noisyForward* in step 1 results in an even less certain position for the robot, namely $\mathit{robotLoc} = \langle 1, 3\rangle \in K_x$. However, the sensing action in step 2, together with the branch point in step 3, lets us split this interval into two parts. In step 4, we assume that $\mathit{robotLoc} \leq 2$ and consider the case where $\mathit{robotLoc} = \langle 1, 2\rangle$. *atTarget*, together with the branch in step 6, lets us divide this interval even further: in step 7, $\mathit{robotLoc} = 2$ and the goal is satisfied, while in step 8, $\mathit{robotLoc} = 1$ and a *moveBackward* action achieves the goal. In step 10 we consider the other sub-interval of the first branch, i.e., $\mathit{robotLoc} = 3 \in K_f$. In this case we have definite knowledge, however, a subsequent *noisyForward* results in $\mathit{robotLoc} = \langle 1, 2\rangle$. The remainder of the plan in steps 12–16 is the same as in steps 5–9: the robot conditionally moves backwards in the case that *robotLoc* is determined to be 1, while the plan trivially achieves the goal if $\mathit{robotLoc} = 2$.

If we instead consider a wider initial *robotLoc* interval, as at the end of Example 1 where $\mathit{robotLoc} = \langle 0, 10\rangle$, then an action like *withinTarget* can possibly be used to further refine a disjunctive interval. For instance, after *atTarget* produces a disjunctive interval $(\mathit{robotLoc} = \langle 0, 1\rangle \,|\mathit{robotLoc} = \langle 3, 10\rangle) \in K_x$ in Example 1, then including *withinTarget* in the plan introduces a pair of branches that splits the K_x formula: along one branch $\mathit{robotLoc} = \langle 0, 1\rangle \in K_x$ and along the other branch $\mathit{robotLoc} = \langle 3, 10\rangle \in K_x$. At this point the planner can focus on the individual reduced intervals in an attempt to achieve the goal along each branch.

Example 3 Finally, we consider a robot with the *moveBackward* and *noisyLocation* actions in Table 1. Here, *noisyLocation* is a noisy sensing action that either senses the actual value of the robot's location, or 1 unit more than the actual location. This is denoted by the interval $\langle x, x + 1\rangle$, where x acts as a placeholder for the actual location. Initially, the location of the robot is unknown, i.e., *robotLoc* is not listed in the planner's databases. The goal is to ensure the robot has moved to or past the target location, i.e., $K(\mathit{robotLoc} \geq \mathit{targetLoc})$, where $\mathit{targetLoc} = 2 \in K_f$.

Here, PKS generate a simple 3-step plan: *noisyLocation*, *moveBackward*, *moveBackward*. The first action adds $\mathit{robotLoc} = \langle x, x + 1\rangle$ to K_v, indicating that the planner has (noisy) knowledge of the robot's location. The second action, *moveBackward*, updates the planner's parametrized K_v knowledge to $\mathit{robotLoc} = \langle x + 1, x + 2\rangle$, which has the effect of tracking the (ungrounded) location across the movement action. Finally, the second *moveBackward* action results in $\mathit{robotLoc} = \langle x + 2, x + 3\rangle$. The planner can then reason $\mathit{robotLoc} \geq 2$ holds since *robotLoc* ranges over $\mathbb{N}$: since $x \geq 0$, it must be the case that $x + 2 \geq 2$.

Although the above examples are simple, they demonstrate interesting plan-time reasoning. In Example 1, the planner tracks the robot's uncertain location with IVFs as physical actions change this information and sensing produces definite knowledge. We note that PKS could also represent the disjunctive nature of *robotLoc* (for $\mathbb{N}$ at least) using the K_x database (e.g., $(\mathit{robotLoc} = 3 \mid \mathit{robotLoc} = 4 \mid \mathit{robotLoc} = 5) \in K_x$), however, an action like *moveForward* would invalidate this information, causing it to be removed, since it changes *robotLoc*. In Example 2, we again show how sensing together with conditional branching allows us to perform a type of reasoning by cases that subdivides intervals into more manageable components. Finally, Example 3 illustrates the ability to track sensed information through physical actions using a placeholder variable.

Example 3 also demonstrates one of the inherent drawbacks of plan-time sensing: for such values to be useful we often need to "ground" them. Here we use the underlying number system and the interval offset to make an assertion about a lower bound. However, PKS also has the ability to work with functions in an "unground" form, allowing them to be composed with other functions, or included in parametrized plans. One of the goals of this work is to extend our model of IVFs so they can also be used in this way.

Conclusions and Future Work

IVFs provide an interesting middle ground between those representations that do not model uncertainty of numeric fluents and those that reason with full possible-world models, or models based on probabilistic distributions. IVFs make a good fit for planners like PKS, but we also believe offer a useful tool that goes beyond knowledge-level planning. For instance, the underlying implementation of IVFs in PKS uses the notion of *semantic attachments* (Dornhege et al. 2009), allowing most of the interval-specific reasoning to be executed through an external library. As a result, similar ideas could be extended to existing PDDL-based planners. We are also currently exploring how IVFs can be encoded directly in PDDL (with minimal extensions).

Finally, although we have not focused on PKS's efficiency, we note that the above examples are all generated in 0.1 seconds using PKS on a single CPU running at 1.86 GHz with 2Gb of RAM. To address the problem of scalability with IVFs, we are exploring compilation techniques in the spirit of (Palacios and Geffner 2007), to treat IVFs as ordinary functions. However, we believe that translation methods will only solve part of the problem and enhancements at the planner level are needed to fully utilise IVFs.

References

Demolombe, R., and Pozos Parra, M. P. 2000. A simple and tractable extension of situation calculus to epistemic logic. In *Proceedings of the International Symposium on Methodologies for Intelligent Systems (ISMIS-2000)*, 515–524.

Dornhege, C.; Eyerich, P.; Keller, T.; Trüg, S.; Brenner, M.; and Nebel, B. 2009. Semantic attachments for domain-independent planning systems. In *Proceedings of ICAPS 2009*.

Edelkamp, S. 2002. Taming numbers and durations in the model checking integrated planning system. *Journal of Artificial Intelligence Research* 20:195–238.

Etzioni, O.; Hanks, S.; Weld, D.; Draper, D.; Lesh, N.; and Williamson, M. 1992. An approach to planning with incomplete information. In *Proceedings of the International Conference on Principles of Knowledge Representation and Reasoning (KR-92)*, 115–125.

Etzioni, O.; Golden, K.; and Weld, D. 1994. Tractable closed world reasoning with updates. In *Proceedings of the International Conference on Principles of Knowledge Representation and Reasoning (KR-94)*, 178–189.

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

Fox, M., and Long, D. 2003. PDDL2.1: An extension to PDDL for expressing temporal planning domains. *Journal of Artificial Intelligence Research* 20:61–124.

Frank, J., and Jónsson, A. 2003. Constraint-based attribute and interval planning. *Journal of Constraints, Special Issue on Constraints and Planning* 8:339–364.

Funge, J. 1998. Interval-valued epistemic fluents. In *AAAI Fall Symposium on Cognitive Robotics*, 23–25.

Hoffmann, J., and Brafman, R. 2005. Contingent planning via heuristic forward search with implicit belief states. In *Proceedings of the International Conference on Automated Planning and Scheduling (ICAPS-2005)*, 71–80.

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

Laborie, P. 2003. Algorithms for propagating resource constraints in AI planning and scheduling: Existing approaches and new results. *Artificial Intelligence* 143:151–188.

Levesque, H. J. 2005. Planning with loops. In *Proceedings of the International Joint Conference on Artificial Intelligence (IJCAI-05)*, 509–515.

Liu, Y., and Levesque, H. J. 2005. Tractable reasoning with incomplete first-order knowledge in dynamic systems with context-dependent actions. In *Proceedings of the International Joint Conference on Artificial Intelligence (IJCAI-05)*, 522–527.

Palacios, H., and Geffner, H. 2007. From Conforman into Classical Planning: Efficient Translations that may be Complete Too. In *Proceedings of the International Conference on Automated Planning and Scheduling (ICAPS-07)*, 264–271.

Pednault, E. P. D. 1989. ADL: Exploring the middle ground between STRIPS and the situation calculus. In *Proceedings of the International Conference on Principles of Knowledge Representation and Reasoning (KR-89)*, 324–332.

Petrick, R. P. A., and Bacchus, F. 2002. A knowledge-based approach to planning with incomplete information and sensing. In *Proceedings of the International Conference on Artificial Intelligence Planning and Scheduling (AIPS-2002)*, 212–221.

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

Petrick, R. P. A. 2006. *A Knowledge-level approach for effective acting, sensing, and planning*. Ph.D. Thesis, Department of Computer Science, University of Toronto, Toronto, Ontario, Canada.

Poggioni, V.; Milani, A.; and Baioletti, M. 2003. Managing interval resources in automated planning. *Journal of Information Theories and Applications* 10:211–218.

Soutchanski, M. 2001. A correspondence between two different solutions to the projection task with sensing. In *Symposium on Logical Formalizations of Commonsense Reasoning (Commonsense 2001). http://www.cs.nyu.edu/faculty/davise/commonsense01/.*

van Eijck, J. 2013. Elements of Epistemic Crypto Logic. Slides from a talk at the LogiCIC Workshop, Amsterdam.

Vassos, S., and Levesque, H. 2007. Progression of situation calculus action theories with incomplete information. In *Proceedings of the International Joint Conference on Artificial Intelligence (IJCAI-07)*, 2029–2034.

Reward-Adaptive Planning with Unknown Teammates from Limited Observations

Aris Valtazanos
School of Informatics
University of Edinburgh
a.valtazanos@ed.ac.uk

Overview

Autonomous agents are increasingly being deployed in tasks requiring interaction with other agents, where collaborative actions and plans are needed to achieve a goal. In these settings, agents must not only deal with uncertainty in how they perceive and act on their environment, but also with not having explicit control over the actions of others. Thus, if different agents have incompatible views of the optimal joint plan of actions, it is highly possible that they will fail to reach their overall goal.

When a task involves collaboration with a priori unknown teammates such as humans (or human-controlled agents), there is also a lack of a clear model that reliably predicts the behaviour of interacting agents, making joint plan selection even harder. Furthermore, when communication resources are scarce or unavailable, agents must rely solely on their own observations of the environment (and of others), in order to select appropriate actions. Additionally, tasks and/or goals within a specific task may vary over time, or teammates may be replaced by other agents, so plans need to be generated anew. When all the above challenges combine, fast adaptation to changing specifications becomes essential. This highlights the need for fast planning methods that can efficiently generate robust action sequences, without recourse to exhaustive, time-consuming exact optimisation.

Motivated by the above constraints, in this paper we introduce *Reward-Adaptive Planning*, an online algorithm for *single-agent* plan generation in partially observable *multi-agent* systems with unknown teammates. Our method builds on the Partially Observable Markov Decision Process formulation, which selects actions with respect to a given reward function and goal. However, unlike many approaches to decentralised coordination, reward-adaptive planning does *not* assume that other agents in the environment use the *same* reward function (even if they do work towards the same goal). Instead, we interleave traditional Monte-Carlo Tree Search with Bayesian Inverse Reinforcement Learning, in order to *simultaneously* plan and learn an approximation of the interacting agents' reward processes. The learning step of the algorithm is *model-free*, so we do not characterise others with respect to pre-defined behavioural classes, but we instead use the search process as an implicit generative mechanism for teammate actions. Moreover, reward-adaptive agents do not model the observations and beliefs of others, and only update reward values over their own action space (as opposed to the entire joint action space, which is typically much larger). This keeps our algorithm tractable and scalable to larger problems.

In order to demonstrate the efficacy of reward-adaptive planning, we consider (and claim that our approach is suitable for) multi-agent domains with the following features:

- Agents do not know the behavioural model of others, who could be executing the same or a different autonomous algorithm, or some human-specified decision tree.
- All agents have the same capabilities (actions and observations), but they cannot observe each other's actions and observations at each time step.
- There is *no* communication between agents either before or during the interaction, so there is no prior agreement on plans; each agent operates in a self-centric manner (but the desired goal is common and known to everyone).
- Each domain has a mixture of individual and joint actions, the latter of which succeed only if they are simultaneously executed by multiple agents. The domain goals also require agents to collaborate (through joint action execution) in order to maximise their reward.

Related Work and Motivation

Planning in partially observable single-agent domains is often posed in terms of a Partially Observable Markov Decision Process (POMDP) (Kaelbling, Littman, and Cassandra 1998). A Decentralised POMDP (Dec-POMDP) (Bernstein et al. 2002) is a generalisation of a POMDP to multi-agent systems, which incorporates joint actions and observations. Despite their representational power, both POMDPs and Dec-POMDPs are hard to solve exactly. This has prompted researchers to study approximate POMDP planning methods. Partially Observable Monte Carlo Planning (POMCP) (Silver and Veness 2010) uses Monte-Carlo Tree Search (MCTS) to *sample* the belief space efficiently. This method has been applied successfully to problems with large branching factors, such as the computer game Go, while also used as part of the winning entry of the 2011 International Probabilistic Planning Competition (Coles et al. 2012).

With respect to the systems considered in this paper, POMCP planning offers a fast and robust solution to single agent design, but does not directly account for multiple

agents. One important problem is that Dec-POMDPs assume that all agents are using the same reward process, which would be limiting in systems with heterogeneous agents. To address this issue, the problem of *ad-hoc* teamwork (Stone et al. 2010) considers collaboration without pre-coordination in multi-agent domains. In this context, Monte-Carlo planning has been used in conjunction with transfer learning to generate team-level strategies (Barrett et al. 2013). This approach uses teammate models that have been learned offline, and updates their likelihood online based on acquired experience. In our work, we choose not to incorporate prior teammate models for two reasons. First, we want our planner to be easily adaptible to varying tasks, for which prior data may not always be available. Second, even when such data exists, there may still be agents not consistent with observations made in past interactions.

In response to the above issues, the main contribution of reward-adaptive planning is the incorporation of *inverse reinforcement learning* (IRL) within the Monte-Carlo planning procedure, in order to approximate the reward process of interacting agents. IRL (Ng and Russell 2000) is the problem of inferring the reward process of an agent from supplied trajectories, and is usually applied in the context of learning from expert demonstrations. In its general form, the problem is ill-posed, as there could be several reward functions with respect to which a set of trajectories is optimal.

Nevertheless, in the systems we consider, we do not assume that teammates are acting optimally, so it is sufficient to compute a reward function approximation that can serve as a generative model for teammate actions in Monte-Carlo simulations. To this end, we use the Bayesian IRL algorithm (Ramachandran and Amir 2007), which is based on a Markov Chain Monte-Carlo (MCMC) sampling procedure and is thus compatible with the POMCP planner. Our algorithm alternates between regular POMCP simulations and reward-adaptive iterations that generate rewards for the interacting teammates based on a Beta distribution prior. At each reward-adaptive iteration, the MCMC process is used to determine whether the planner should continue sampling in this mode or revert to regular POMCP updates. Moreover, the action selection procedure accounts not only for the planning agent's values (as in the original single-agent POMCP algorithm), but also for the learned teammate rewards. Thus, we progressively build a teammate action model directly from the samples obtained over the course of the interaction.

Preliminary Results

We have conducted preliminary tests on a cooperative box pushing domain (Seuken and Zilberstein 2007), where agents get a high reward for jointly pushing a large box to a target position, and a smaller reward for individually pushing smaller boxes. Our results show that reward-adaptive planning outperforms a decentralised POMCP algorithm (with no reward-adaptive iterations) when both agents run the same algorithm, while also performing more robustly in the presence of heterogeneous teammates. We are currently testing our method in a more complex robot kitchen domain involving multiple joint actions and tighter collaboration constraints.

	Mean Return	**Time(s)**
Rand v **Rand**	-15.25 $\pm$ 0.64	0.001
MAOP-0 v **MAOP-0**	7.50	0.14
D-POMCP v **D-POMCP**	10.08 $\pm$ 0.51	0.099
RewAd v **RewAd**	15.61 $\pm$ 0.53	0.131
HumDes v **HumDes**	17.71 $\pm$ 0.90	0.239
RewAd v **D-POMCP**	13.41 $\pm$ 0.58	0.111
D-POMCP v **HumDes**	12.35 $\pm$ 0.54	0.172
RewAd v **HumDes**	13.99 $\pm$ 0.58	0.189

Table 1: Results from the Cooperative Box Pushing Domain. Results are averaged over 1000 runs with 512 action sequence samples at each decision step and a horizon of 20 steps. *Mean return*: average discounted reward obtained in each experiment. *Time*: mean total computation time per agent per run (on a standard desktop 2.93GHz dual-core machine). The top part of the table corresponds to experiments with identical agents, whereas the bottom part gives results for coordination with heterogeneous teammates. **Rand**: random action selection. **RewAd**: reward-adaptive planning. **D-POMCP**: decentralised POMCP implementation. **HumDes**: human-designed agent running a fixed behaviour. **MAOP-0**: Multi-agent Online Planning with no communication from (Wu, Zilberstein, and Chen 2011).

Acknowledgments This work has been funded by the European Commission through the EU Cognitive Systems and Robotics project Xperience (FP7-ICT-270273).

References

Barrett, S.; Stone, P.; Kraus, S.; and Rosenfeld, A. 2013. Teamwork with limited knowledge of teammates. In *AAAI*.

Bernstein, D. S.; Givan, R.; Immerman, N.; and Zilberstein, S. 2002. The complexity of decentralized control of Markov Decision Processes. *Math. Oper. Res.* 27(4):819–840.

Coles, A. J.; Coles, A.; Olaya, A. G.; Celorrio, S. J.; López, C. L.; Sanner, S.; and Yoon, S. 2012. A survey of the seventh International Planning Competition. *AI Magazine* 33(1).

Kaelbling, L. P.; Littman, M. L.; and Cassandra, A. R. 1998. Planning and acting in partially observable stochastic domains. *Artificial Intelligence* 101(1-2):99–134.

Ng, A. Y., and Russell, S. 2000. Algorithms for Inverse Reinforcement Learning. In *ICML*, 663–670.

Ramachandran, D., and Amir, E. 2007. Bayesian inverse reinforcement learning. In *IJCAI*.

Seuken, S., and Zilberstein, S. 2007. Memory-Bounded Dynamic Programming for DEC-POMDPs. In *IJCAI*.

Silver, D., and Veness, J. 2010. Monte-Carlo Planning in Large POMDPs. In *NIPS*, 2164–2172.

Stone, P.; Kaminka, G. A.; Kraus, S.; and Rosenschein, J. S. 2010. Ad Hoc Autonomous Agent Teams: Collaboration without Pre-Coordination. In *AAAI*.

Wu, F.; Zilberstein, S.; and Chen, X. 2011. Online planning for multi-agent systems with bounded communication. *Artificial Intelligence* 175(2):487–511.