SEVENTH FRAMEWORK PROGRAMME

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

XPERIENCE.ORG

Deliverable Number:	D3.2.4
Deliverable Title :	Transfer of Structural Bootstrapping for Planning: Report or scientific publication on implementation of structural bootstrapping for planning within the architecture and final demonstration
Type (Internal, Restricted, Public):	PU
Authors:	Ron Petrick, Kira Mourão, Bart Craenen, Christopher Geib, and Mark Steedman
Contributing Partners:	UEDIN

Contractual Date of Delivery to the EC:	31-01-2015
Actual Date of Delivery to the EC:	04-02-2015

Contents

1 Executive Summary

One of the general objectives of WP3 (Generative Mechanisms) is to investigate and implement structural bootstrapping at the planning level. In particular, WP3.2 (Structural Bootstrapping for Planning) focuses on the task of extending the capabilities of current high-level planning models by applying structural bootstrapping to the knowledge-rich representation of actions and plans. This work has implications across Xperience by providing the apparatus needed to integrate and support plan generation and execution in low-level robotics domains (WP2, Outside In: Development and Representations), higher-level domains requiring language and communication (WP4, Interaction and Communication), and in the project's demonstration scenarios (WP5, System Integration and Demonstration).

This deliverable reports on the implementation of structural bootstrapping within the architecture developed by UEDIN for integrating planning capabilities on robot platforms. To this end, we focus on contributions to Task 3.2.4 (Extended reasoning about object and indexical knowledge), Task 3.2.3 (Plan structure and execution), Task 3.2.2 (Learning knowledge-level control rules), and Task 2.3.2 (Learning high-level action descriptions (rule learning)) from the current reporting period, which have continued to address the problems of learning action models for planning and provide the necessary software-level framework for integrating planning on robot platforms. Two papers are attached to this deliverable (Mourão et al., 2015; Petrick, 2015), which provide details of these contributions, as highlighted below.

A robot operating in a real-world domain requires the ability to reason and plan—and to learn from its past experiences. In Xperience, automated planning techniques are provided by the PKS planner (Petrick and Bacchus, 2002, 2004), which UEDIN has been extending and integrating on the project's robot platforms (in particular, ARMAR), for use in the demonstration scenarios. PKS is a domain independent planner that can construct plans in the presence of incomplete information. Unlike many planners, PKS builds plans at the knowledge level (Newell, 1982), 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 can build contingent plans with sensing actions, and supports numerical reasoning, run-time variables (Etzioni et al., 1992), and features like functions that arise in real-world domains.

Building realistic models of real-world domains that can be used with a general-purpose planner can be a difficult task, often complicated by the representational differences that exist between the robot-level sensorimotor components and the high-level reasoning components. 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 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 an automated planner. A second, but equally important, problem concerns the actual implementation of a high-level planning component on a robot platform, and the architectural framework that this integration requires. Structural bootstrapping plays a role in overcoming the first problem, while appropriate integration tools are needed for the second problem. We describe our work to address both of these problems in this deliverable.

First, we provide a comprehensive report on the problem of learning planning models in uncertain domains, building on our earlier work reported in previous deliverables (D3.2.2 and D3.2.3). These techniques provide a form of structural bootstrapping that enables planning actions to be learnt or improved, from a history of past experiences described by a set of observed successful plans or action traces. This approach is general purpose and can learn planning models using data generated from real-world or simulated domains. Moreover, the resulting models can be used with any planning system supporting STRIPS-like actions, not just PKS. This work is reported in (Mourão et al., 2015), attached below.

Second, we report on an application programming interface (API) to the high-level PKS planner, which provides abstract planning services in a client-server model using the Internet Communications Engine

(ICE). Since the last planning deliverable, D3.2.3, this interface has been substantially rewritten and extended to provide new functionality. The API now abstracts common planning activities supported by the backend planning system, including functions for adding or modifying actions, properties, and objects in a planning domain. Additional functions are provided to control the generation and iteration of plans. This work provides the necessary tools for working with adaptive domain models of the kind provided by (Mourão et al., 2015), and for supporting software-level integration activities. This work is reported in (Petrick, 2015), also attached below.

In addition to these core research and development activities of WP3.2, this reporting period has seen significant progress on the software-level integration of the PKS planner onto the ARMAR robot platform, in conjunction with KIT. PKS is now implemented as an internal component integrated as part of the Plan Execution Monitor (PEM), which in turn is part of the ArmarX framework. The PEM acts as a unified control component, interacting with components of the ArmarX Robot Framework Layer on the one hand, and the (internal) planning component on the other hand (with integration of a plan recognition component (see deliverable D4.2.5) planned for the near future).

In this context, PKS provides dynamic planning functionality and plan monitoring information. From the robot side, the PEM receives memory and perception information from which a world model and state description can be constructed. Throughout a scenario life-cycle the PEM then maintains a cyclical consultation sequence whereby the planner provides both plans and plan execution monitoring information for achieving the system's current goals.

As a result, the initial integration of the PKS component on ARMAR is now complete, and a server-side interface and control object for PKS is realised and available. The client-side implementation and use of this interface by the plan execution monitor (PEM) is finalised. The PKS component itself is integrated within the ArmarX package structure, and the planning-side functionality of the PEM is integrated and implemented. The PEM's interaction with ArmarX components, through the middleware, as well as with the PKS internal component, is realised and integrated. The implementation of the PEM thus far is also integrated in the ArmarX package structure. However, more comprehensive testing and debugging of PKS within the full robot system is still required.

References

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 Knowledge Representation and Reasoning (KR)*, pages 115–125.

Fikes, R. and Nilsson, N. (1971). STRIPS: A New Approach to the Application of Theorem Proving to Problem Solving. *Artificial Intelligence*, 2:189–208.

Mourão, K., Petrick, R. P. A., and Steedman, M. (2015). Learning planning operators under uncertainty. In preparation for submission to the Journal of Artificial Intelligence Research (JAIR).

Newell, A. (1982). The Knowledge Level. *Artificial Intelligence*, 18(1):87–127.

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)*, pages 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)*, pages 2–11.

Petrick, R. P. A. (2015). An application programming interface to high-level planning with pks. Technical report, University of Edinburgh.

2 Attached Papers

(Mourão et al., 2015) Learning Planning Operators under Uncertainty, In preparation for submission to the Journal of Artificial Intelligence Research (JAIR), 2015.

Abstract: Agents learning to act autonomously in real-world domains must acquire a model of the dynamics of the domain in which they operate. For example, most AI planners use pre-specified domain models as input in order to generate plans. However creating domain models is notoriously difficult, even with the involvement of domain experts. Furthermore, to be truly autonomous, agents must be able to learn their own models of world dynamics. An alternative therefore is to learn domain models from observations, either via known successful plans or through exploration of the world. This route is also challenging, as agents typically do not operate in a perfect world: both actions and observations may be unreliable. In this paper we present a method which, unlike other approaches, can learn both from observed successful plans and from action traces generated by exploration. Importantly, the method is robust in a variety of settings, able to learn useful domain models when observations are noisy and incomplete, or when action effects are noisy or non-deterministic.

Our approach is to divide the learning problem into two stages, decoupling the requirement to learn compact planning operators from the requirement to tolerate noise, partial observability and non-determinism. In the first stage we learn action models by constructing a set of classifiers which tolerate noise and partial observability, but whose action models are implicit in the learnt parameters of the classifiers. The initial action model learnt in this first stage acts as a noise-free, fully observable source of observations from which to extract explicit action rules. In the second stage we devise a novel method to derive explicit planning operators from the model implicit in the classifiers.

Through a range of experiments using International Planning Competition domains incorporating noise, partial observability and non-determinism, we show that our approach learns accurate domain models suitable for use by standard planners. We also demonstrate that where settings are comparable, our results equal or surpass the performance of state-of-the-art methods.

(Petrick, 2015) An Application Programming Interface to High-Level Planning with PKS, Technical Report, University of Edinburgh, 2015.

Abstract: The task of integrating planners on robot platforms often centres around the problem of representation, and how to abstract the capabilities of a robot and its working environment into a suitable form for use by the planner. Integration also typically requires the ability to communicate information between system components and, thus, requires a consideration of certain engineering-level concerns, to ensure proper interoperability with components that aren't traditionally considered in theoretical planning settings. This report presents a snapshot of UEDINs contribution to the work of integrating high-level planning on robot platforms in the Xperience project. In particular, this document focuses on the current state of an application programming interface (API) designed to provide an abstract, general-purpose 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 meant to be generic and any planner which supports the API can be used in its place as an alternative backend. This document presents details of the latest version of the interface (Version 0.85, 2014-10-09). An example planning domain is also described, to demonstrate certain features of the representation and API.

Journal of Artificial Intelligence Research - (—-) - Submitted -; published -

Learning Planning Operators under Uncertainty

Kira Mourão KMOURAO@INF.ED.AC.UK
Ronald P. A. Petrick RPETRICK@INF.ED.AC.UK
Mark Steedman STEEDMAN@INF.ED.AC.UK
School of Informatics
University of Edinburgh
Edinburgh, EH8 9AB, UK

Abstract

Agents learning to act autonomously in real-world domains must acquire a model of the dynamics of the domain in which they operate. For example, most AI planners use pre-specified domain models as input in order to generate plans. However creating domain models is notoriously difficult, even with the involvement of domain experts. Furthermore, to be truly autonomous, agents must be able to learn their own models of world dynamics. An alternative therefore is to learn domain models from observations, either via known successful plans or through exploration of the world. This route is also challenging, as agents typically do not operate in a perfect world: both actions and observations may be unreliable. In this paper we present a method which, unlike other approaches, can learn both from observed successful plans and from action traces generated by exploration. Importantly, the method is robust in a variety of settings, able to learn useful domain models when observations are noisy and incomplete, or when action effects are noisy or non-deterministic.

Our approach is to divide the learning problem into two stages, decoupling the requirement to learn compact planning operators from the requirement to tolerate noise, partial observability and non-determinism. In the first stage we learn action models by constructing a set of classifiers which tolerate noise and partial observability, but whose action models are implicit in the learnt parameters of the classifiers. The initial action model learnt in this first stage acts as a noise-free, fully observable source of observations from which to extract explicit action rules. In the second stage we devise a novel method to derive explicit planning operators from the model implicit in the classifiers.

Through a range of experiments using International Planning Competition domains incorporating noise, partial observability and non-determinism, we show that our approach learns accurate domain models suitable for use by standard planners. We also demonstrate that where settings are comparable, our results equal or surpass the performance of state-of-the-art methods.

1. Introduction

Developing agents with the ability to act autonomously in the real world is a major goal of artificial intelligence. One important aspect of this development is the acquisition of domain models to support planning and decision-making: to operate effectively in the world, an agent must be able to determine when to perform particular actions, and what effects its actions are likely to have. For example, we might want an agent to go shopping, or to prepare a meal. To perform these tasks well the agent needs an understanding of how the world usually works, as well as less typical consequences an action may have. For instance, usually items in a shopping bag move with the bag, but sometimes items fall out, or the bag bursts; sometimes we might fail to pick up the bag correctly, and drop it; or sometimes there may be items in the bag that we do not know are there.

In this paper we develop a new approach to acquiring explicit domain models from the raw experiences of an agent exploring the world, where the agent's observations are incomplete, observations and actions are subject to noise, and actions are non-deterministic. The domains we consider are relational, where state descriptions contain objects, object attributes and relations between the objects. Given the autonomous learning setting, we assume only a weak domain model where the agent knows how to identify objects, has acquired predicates to describe object attributes and relations, and knows what elementary actions it may perform, but not the appropriate contexts for the actions, or their effects. Experience in the world is then developed through observing changes to object attributes and relations when motor-babbling with primitive actions.

Even when operating in a noise-free, fully observable and deterministic world, efficiently learning domain dynamics is challenging because of the relational nature of the domains involved. In these domains, learning the preconditions or effects of an action is a form of relational concept learning. Learning concepts in such relational domains is much more complex than in non-relational attribute-based domains, where states can be represented by fixed-length vectors of values, as in robot sensor arrays. For instance, determining whether a precondition holds in an attribute-based domain involves a straightforward comparison of one vector with another of the same length. In a structural domain the corresponding coverage test is NP-hard (Haussler, 1989; Nienhuys-Cheng & de Wolf, 1997). Real-world domains are not only relational, but provide further challenges: agents' observations may be noisy, or incomplete; actions may be non-deterministic; the world may be noisy or contain many irrelevant objects and relations. When an agent makes noisy or incomplete observations of the world, its mechanism for learning action models cannot rely on conditional independence structures common in fully observable domains Similarly when the world is non-deterministic, the learning mechanism must be able to distinguish between an uncertain outcome and noise.

Our approach to learning domain models involves a number of novel contributions. We develop a two-stage approach to the problem which decouples the requirement to tolerate noisy, incomplete observations from the requirement to learn compact planning operators. In the first stage we learn action models by decomposing the problem into multiple classification problems, where the decomposition is based on a deictic representation. Each classifier predicts whether a single fluent (or a set of fluents with the same deictic term) change as a result of a specified action. These kernel classifiers tolerate noise and partial observability, but the underlying action models are implicit in the learnt parameters of the classifiers. We represent world states as graphs and develop a novel graph kernel to perform similarity comparisons between states. The features of our kernel are closely related to the preconditions underlying the true action models.

The initial implicit action model learnt in this first stage acts as a noise-free, fully observable source of observations from which to extract explicit action rules. In the second stage we propose a novel method to derive explicit planning operators, with associated probabilities, from the model implicit in the kernel classifiers. In experiments the resulting rules perform as well as the original classifiers, while providing a compact representation of the action models suitable for use in automated planning systems.

2. Problem Definition

We work with domains described in the classical first-order planning representation (Ghallab, Nau, & Traverso, 2004). A *domain* $\mathcal{D}$ is defined as a tuple $\mathcal{D} = \langle \mathcal{O}, \mathcal{P}, \mathcal{A} \rangle$, where $\mathcal{O}$ is a finite set of world objects, $\mathcal{P}$ is a finite set of predicate (relation) symbols, and $\mathcal{A}$ is a finite set of actions. Each predicate and action also has an associated arity. A *fluent expression* is a statement of the form $p(c_1, c_2, \ldots, c_n)$, where $p \in \mathcal{P}$, n is the arity of p, and each $c_i \in \mathcal{O}$. A *state* is any set of fluent expressions, and $\mathcal{S}$ is the set of all possible states. Since state observations may be incomplete we assume an open world where unobserved fluents are considered to be unknown. For any state $s \in \mathcal{S}$, a fluent expression ϕ is true at s iff $\phi \in s$. The negation of a fluent expression, $\neg\phi$, is true at s (also, ϕ is false at s) iff $\neg\phi \in s$. If $x \in s$ then $\neg x \notin s$. Any (legal) fluent expression not in s is unobserved.

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 fluent expressions and negated fluent expressions. We consider three different kinds of action effects. First, we allow standard STRIPS effects, where each $e \in$ *Eff*$_a$ has the form *add*(ϕ) or *del*(ϕ), and ϕ is any fluent expression. Second, we permit *conditional effects* of the form $C_e \Rightarrow$ *add*(ϕ) or $C_e \Rightarrow$ *del*(ϕ). Here, C_e is any set of fluent expressions and negated fluent expressions, and is referred to as the *secondary preconditions* of effect e. Third, we allow universally quantified effects of the form $\forall x_1, x_2, \ldots, x_n E(x_1, x_2, \ldots, x_n)$ where E is a set of effects, possibly conditional. 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*. For any fluent expression or action ϕ, the function $label(\phi)$ returns its predicate or action symbol, $args(\phi)$ returns the set of arguments of ϕ, and $args_i(\phi)$ returns the i-th argument of ϕ.

In addition we consider domains where action effects are stochastic or noisy. We will learn noisy deictic rules (NDRs[1]) of the form defined by Pasula, Zettlemoyer, and Kaelbling (2007). Rather than explicitly modelling action effects which are highly unlikely or difficult to model, NDRs account for such outcomes with a single *noise outcome*. Each alternative outcome for an action has an associated probability, indicating how likely it is that that particular outcome will arise. With NDRs we assume that the set of rules for any action has mutually exclusive preconditions, ensuring that any state-action pair is not covered by more than one rule, so that the probabilities of the effects for any single rule will sum to 1.

The preconditions and effects of NDRs contain variables corresponding to arguments of the action, as well as *deictic terms*, variables referring to objects defined in terms of their properties and relations to other objects in the world (see Section 4.1). 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*), we make the *deictic scope assumption* that objects mentioned in the preconditions or effects are either action parameters[2] or related to the action parameters, i.e., they have a deictic term. In this work we restrict ourselves to a 1-step deictic scope assumption, where related objects must be directly related to the action parameters.[3] In the rules we learn, the definitions of any deictic terms will be included in the preconditions of the rules.

2.1 The learning task

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 either plans, or from data generated by an agent performing a sequence of randomly

1. Note that noisy deictic rules can be translated to alternative planning representations such as PPDDL, as demonstrated by Lang and Toussaint (2010).
2. Thus the STRIPS scope assumption is a special case.
3. Greater depths are possible, where objects are 2,3 or more steps from the action parameters, but this is left to future work.

selected actions in the world and observing the resulting states. In either case, the sequence of states and action instances is denoted by $s_0, a_1, s_1, \ldots, a_n, s_n$ 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 $s'_0, a_1, s'_1, \ldots, a_n, s'_n$, where state observations may be noisy (some $\phi \in s_i$ may be observed as $\neg\phi \in s'_i$) 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 Briefcase domain (shown in Figure 2a), an ADL domain where an agent inserts and removes items from a briefcase, and moves it from location to location. For a state with items A and B in the briefcase at location L1, and item H at location L2, the state description could be:

```
(AND (is-at L1) (in A) (in B) (NOT (in H))
  (at A L1) (at B L1) (at H L2) (NOT (at A L2))
  (NOT (at B L2)) (NOT (at H L1)) (NOT (is-at L2))).
```

A sequence of states and actions could be as follows:

```
s0: (AND (is-at L1) (in A) (in B) (NOT (in H)) (at A L1)
      (at B L1) (at H L2) (NOT (at A L2)) (NOT (at B L2))
      (NOT (at H L1)) (NOT (is-at L2)))
a1: (take-out A)
s1: (AND (is-at L1) (NOT (in A)) (in B) (NOT (in H))
      (at A L1) (at B L1) (at H L2) (NOT (at A L2))
      (NOT (at B L2)) (NOT (at H L1)) (NOT (is-at L2)))
a2: (move L1 L2)
s2: (AND (is-at L2) (NOT (in A)) (in B) (NOT (in H))
      (at A L1) (at B L2) (at H L2) (NOT (at A L2))
      (NOT (at B L1)) (NOT (at H L1)) (NOT (is-at L1)))
a3: (put-in A)
s3: (AND (is-at L2) (NOT (in A)) (in B) (NOT (in H))
      (at A L1) (at B L2) (at H L2) (NOT (at A L2))
      (NOT (at B L1)) (NOT (at H L1)) (NOT (is-at L1))).
```

Taking a sequence of such inputs, we learn action descriptions for each action in the domain. For example, the move action, which moves the briefcase from one location to another, would be represented as:

```
(:action move
  :parameters (?m ?l - location)
  :precondition  (is-at ?m)
  :effect (and (is-at ?l) (not (is-at ?m))        (0.9)
    (forall (?x - portable) (when (in ?x)
      (and (at ?x ?l) (not (at ?x ?m))))))
          no change                              (0.09)
          noise                                  (0.01).
```

3. Outline

The basis of our approach is the division of the learning problem into two parts (summarised in Figure 1). Initially a classification method is used to learn to predict effects of actions, then planning operators are derived from the resulting action representations. We define the *implicit action model* to be the model of the domain implicit in the learnt parameters of the classifiers, and the *explicit action model* to be the domain model described by the derived planning operators.

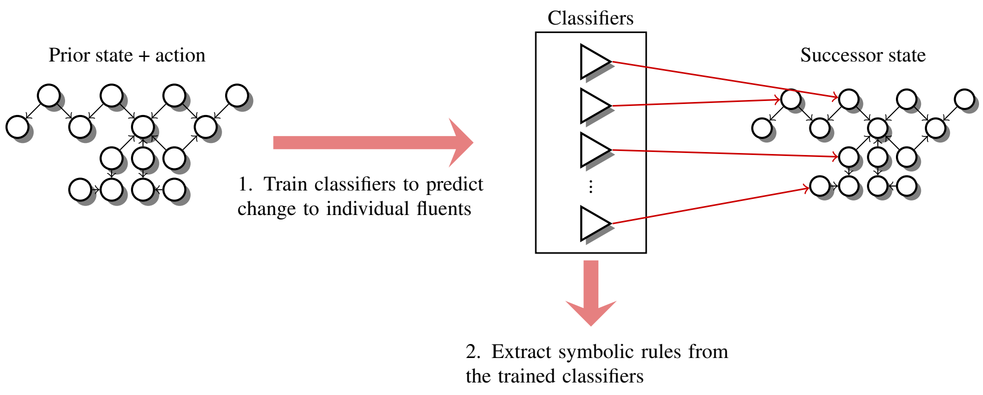

Prior state + action
Classifiers
Successor state
1. Train classifiers to predict change to individual fluents
2. Extract symbolic rules from the trained classifiers

Figure 1: Overview of the learning process: First, classifiers are trained to predict state changes, given an action and a state observation encoded as a graph. Each classifier predicts change to an individual fluent or small set of fluents. Second, symbolic rules (preconditions and effects) are extracted from the trained classifiers.

We represent state observations as graphs where objects, 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 one or a small set of fluents of the overall state, given an input situation and an action. After training the classifiers, we derive planning operators from the learnt parameters.

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.

4. Representation

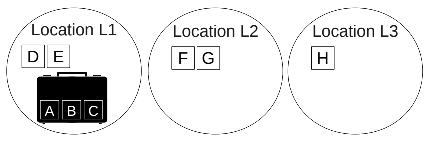
In order to apply classification learning to our problem, we construct a structured graphical representation of state observations. A critical aspect of the representation is the use of deictic reference to determine which elements of the world state to include in the state graphs. In this section we define deictic reference and show how it is applied to the graphical representation of world states.

4.1 Deictic reference

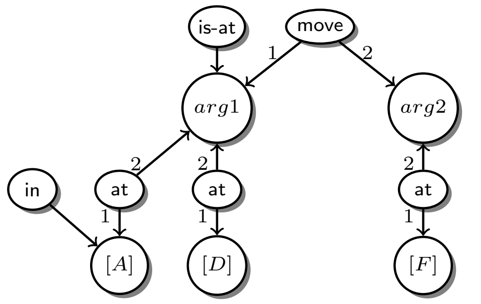
The notion of deictic reference is central to this paper, underpinning both the state representation used and the learning process. The term *deixis* originates from Ancient Greek meaning “pointing”

```
(define (domain briefcase)
  (:requirements :adl)
  (:types portable location)
  (:predicates (at ?y - portable ?x - location)
               (in ?x - portable)
               (is-at ?x - location))

 (:action move
  :parameters (?m ?l - location)
  :precondition (is-at ?m)
  :effect (and (is-at ?l) (not (is-at ?m))
            (forall (?x - portable) (when (in ?x)
              (and (at ?x ?l) (not (at ?x ?m)))))))

 (:action take-out
  :parameters (?x - portable)
  :precondition (in ?x)
  :effect (not (in ?x)))

 (:action put-in
  :parameters (?x - portable ?l - location)
  :precondition (and (not (in ?x)) (at ?x ?l)
                     (is-at ?l))
  :effect (in ?x)))
```

(a)



(b)



(c)

Figure 2: (a) A PDDL description of the Briefcase domain, (b) a state in the Briefcase domain, and (c) its graphical representation (as a situation graph) when combined with the `move` action. Objects are represented by deictic terms indicated by [ ]: here, given the action `(move arg1 arg2)`, `[A]={x:(at x arg1)` $\wedge$ `(in x)` $\wedge$ $\neg$`(at x arg2)}`, `[D]={x:(at x arg1)` $\wedge$ $\neg$`(in x)` $\wedge$ $\neg$`(at x arg2)}`, and `[F]={x:(at x arg2)` $\wedge$ $\neg$`(in x)` $\wedge$ $\neg$`(at x arg1)}`. For clarity, negative fluents are omitted in the graph.

or “showing”, and in linguistics, deixis is the term for expressions which refer to the context of an utterance, for example, “this” or “here” (Finegan, 1998). The context must be known for the expression to be correctly understood. The idea of deixis has been adopted in the cognitive robotics field, where a deictic reference is a pointer to objects which have a particular role in the world, with object roles coded relative to the agent or current action. Agre and Chapman (1987) introduced deictic references in the form of “indexical-functional entities” which refer to objects in the game of Pengo in terms relative to the main agent in the game, e.g., “the-block-I’m-pushing” or “the-bee-on-the-other-side-of-this-block-next-to-me”. Deictic references are only maintained for objects close to the agent, so other objects are effectively invisible to it. As the state changes, objects may move in or out of range, and change role.

Deictic references have also been used to recode a first-order description of the world in terms of the arguments of the current action (Benson, 1996; Pasula et al., 2007). Encoding a state using deictic reference involves assigning deictic terms to objects in the world. Each deictic term uniquely defines an object or set of objects in the current context. In the approach taken by Benson (1996) and Pasula et al. (2007), each domain constant and each argument of the action is considered to be a deictic term in its own right (referring to the actual object which is the argument for the specific action instance). Then any object can be assigned a deictic term if it is related only to objects which have themselves already been assigned deictic terms. The new deictic term for the object is written in terms of the existing deictic terms and the relationships with the object. The process is repeated until no further objects can be coded via deictic reference.

In the deictic representation we use, we similarly 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. The deictic term of any other object is its definition in terms of its relations with the action parameters and other objects in the world. Thus, similar to the definition given by 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.

We extend the definition of deixis by adding 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, 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 in the work of 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 positive 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 Briefcase domain (Figure 2) if the action were `(move L1 L2)` in the state shown in Figure 2b, $L1$ and $L2$, as action parameters, would have deictic terms $arg1$ and $arg2$ indicating their positions in the *move* action argument list. Relative to the `(move L1 L2)` action, object A is referred to by the deictic terms $x : at(x, arg1) \wedge in(x) \wedge \neg at(x, arg2)$, $x : at(x, arg1) \wedge \neg at(x, arg2)$, and $x : at(x, arg1)$ but not, for example, $x : in(x)$, $x : \neg at(x, arg2)$ or $x : in(x) \wedge \neg at(x, arg2)$. The A node is labelled $[A]$ to indicate that it represents the equivalence class of all objects with the same deictic terms as A. Object H has no positive relations with the action parameters or other objects with deictic terms, and so has no deictic term in this case.

Using a representation based on deictic reference means that observations are described in terms of the action and the agent instead of specific objects. Objects are considered in terms of their roles in an action, as well as their relationships with other objects in the world. Therefore this representation supports generalisation by abstracting from specific objects and allowing states to be compared in terms of the roles of objects and their relations. This abstraction is already a property of simpler STRIPS representations, because STRIPS actions are defined in terms of their arguments only: when comparing instances of the same STRIPS action, it is natural to consider objects in the same position in the action's argument list as objects with the same role in the action. The deictic representation we describe is simply a generalisation of the STRIPS representation to a broader set of possible object roles.

A further benefit of using a deictic representation is that it reduces the size of the state representation, by limiting the observations to a small number of objects. Providing the roles corresponding to the deictic references are relevant to the task, the reduced representation makes learning of domain dynamics easier, as there are fewer possible states and actions to consider. The relevant deictic terms may be themselves learnt (Whitehead & Ballard, 1991) or be obtained by creating deictic terms for all possible relevant roles (Benson, 1996; Pasula et al., 2007), although this leads to a corresponding increase in the size of the state space.

How might a deictic representation arise? Any agent needs to be able to connect its actions with the world. This is particularly apparent in situated cognition, where it has been proposed that the external world could be used as an external visual memory (Brooks, 1990; O'Regan, 1992), reducing (or in the extreme, removing) the need for internal representation of the world. If the world is acting as a memory, this implies a need for a mechanism to address this memory, so that parts of the visual scene can be returned to when required.

A possible mechanism is given by the visual indexing hypothesis (Pylyshyn, 2000), which proposes that humans can maintain pointers to objects in the world (a visual index or "FINger of INSTantiation" known as a FINST (Pylyshyn, 1989)). The main purpose of a visual index is to bind an argument of a mental relational predicate or motor command to a real-world object. Thus, a visual index differs from putative object representations such as object files (Kahneman, Treisman, & Gibbs, 1992), event files (Hommel, 2004) or object-action complexes (OACs) (Krüger, Geib, Piater, Petrick, Steedman, Wörgötter, Ude, Asfour, Kraft, Omrčen, Agostini, & Dillmann, 2011), in that these are proposed as temporary instantiations of the collection of features and affordances of an object, whereas the visual index is the link between them and the object to which they refer.

There are clear parallels between visual indexing, deictic reference and attention. A visual index is clearly a deictic reference: it points from the representation of an object with a particular role (the argument of a mental relational predicate or motor command) to an object in the world. Ballard, Hayhoe, Pook, and Rao (1997) describe attention as a "neural deictic device", while Hurford (2003) explicitly identifies deictic reference with attention, although Pylyshyn (2001, 2009) argues slightly differently that visual indexing is one stage in a series of processes which constitute attention.

Moreover, Hurford (2003) proposes that there are neural correlates of the full predicate-argument structure of a logical formula, where deictic references, in some form, take the role of arguments in relational predicates. He relates the differential processing of sensory inputs in the ventral and dorsal streams (Goodale & Milner, 1992) to the respective construction of predicates and arguments. The dorsal stream, concerned with visuomotor control and the capture of egocentric locational properties of objects corresponds to the maintenance of deictic references and thence arguments, while the ventral stream, concerned with perception of object properties, corresponds to the creation and maintenance of predicates.

Deictic representations therefore support the goal of grounding representations in the real world while also enabling the creation and maintenance of predicate-argument structures necessary for reasoning in relational domains. Since deictic coding may come about through attentional mechanisms already well-explored in the literature (e.g., saliency maps (Koch & Ullman, 1985; Itti, Koch, & Niebur, 1998)), we will assume the existence of some form of deictic coding mechanism to generate state observations.

4.2 State representation

We will represent states by labelled directed graphs. We first set out our terminology, defining a graph G to be a pair (V, E) where V is a finite set of vertices and E a finite set of directed edges. An edge is an ordered pair of elements of V. Additionally there is a labelling function $label$ that assigns labels to nodes and edges in the graph.

We represent a state s by a labelled directed graph G, where objects (as deictic terms), fluents, and the current action are represented by nodes in the graph. Edges link fluents (or the current action) to their arguments, and are labelled with the argument position. Thus whenever the fluent $p(o_1, \ldots, o_n) \in s$ and for each i, o_i has a deictic term, the following holds of the corresponding graph $G = (V, E)$:

$\exists v \in V$ such that $label(v) = p$ and
$\exists v_1, \ldots, v_n \in V$ such that $label(v_i) = o_i$ and
$\exists e_i, \ldots, e_n \in E$ such that $e_i = \langle v, v_i \rangle$ and $label(e_i) = i$.

The current action $a(o_1, \ldots, o_n)$ is treated analogously. The same state in different action contexts will therefore be represented by different graphs. We term nodes corresponding to fluents, actions and objects as *fluent nodes*, *action nodes* and *object nodes* respectively.

Note that both positive- and negative-valued fluents are included in the graph. Therefore the size of the graph corresponding to some state is limited by restricting the deictic terms to zero- or first-order terms only.[4] 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. Learning STRIPS action representations is itself a non-trivial learning problem (Walsh & Littman, 2008), but here we use first-order deictic terms in order to learn the dynamics of domains more complex than STRIPS. Figure 2 shows a graph encoding a briefcase domain state in the context of the `(move L1 L2)` action, after converting the objects to deictic terms.

4.3 Calculating changes

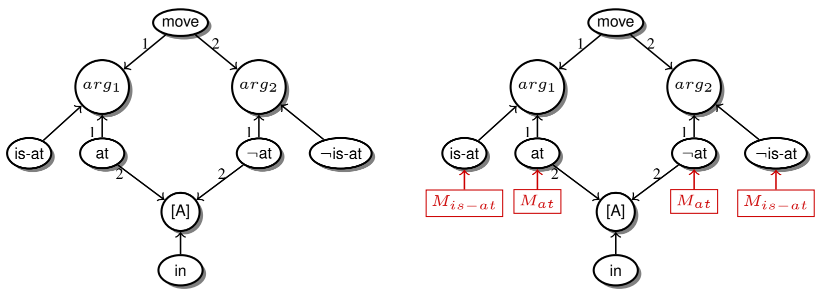
Our classification model operates by taking a world state (as a graph) as input, and predicting which fluents will change. Each training example must therefore consist of a prior state, an action, and the changes resulting from performing the action on the state.

We denote changes by creating a *change graph*, created by annotating the prior state graph with additional *marker* nodes (similar to the approach of Halbritter & 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. 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. For example, for the `move` action in Figure 2, the changes to the state would be indicated as shown in Figure 3.

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.

4. Higher order terms are possible but are left to future work.



(a) Prior state (b) Change graph

Figure 3: The change graph is created by adding marker nodes to the prior state, indicating which fluents change as a result of the action. In the case of the Briefcase domain `move` action, (`at [A] arg`$_1$) and (`is-at arg`$_1$) become false and (`at [a] arg`$_2$) and (`is-at arg`$_2$) become true.

4.4 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 the space is closely related to the subgraph kernel (mapping graphs to the space of all possible subgraphs), known to be NP-hard (Gärtner, Flach, & Wrobel, 2003), and contains the feature space of the DNF kernel (Sadohara, 2001; Khardon & Servedio, 2005), which cannot be used by a perceptron to PAC-learn DNF (Khardon, Roth, & Servedio, 2005).

Following Mourão, Zettlemoyer, Petrick, and Steedman (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 we exclude *invalid* conjunctions where any object is not related (directly or indirectly) to the action parameters by fluents in the conjunction itself. For example, considering subgraphs of the briefcase state shown in Figure 2, the conjunction corresponding to the subgraph in Figure 4a would be *valid*, but the conjunction corresponding to Figure 4b would be *invalid*, as the `portable` object is not related to `arg1` or `arg2` in this subgraph. This restriction avoids learning meaningless preconditions where variables in the preconditions are undefined e.g., action $a(x, y)$ with the single 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 valid conjunction of up to k fluents present in the state graph. 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, Schweitzer, van Leeuwen, Mehlhorn, & Borgwardt,

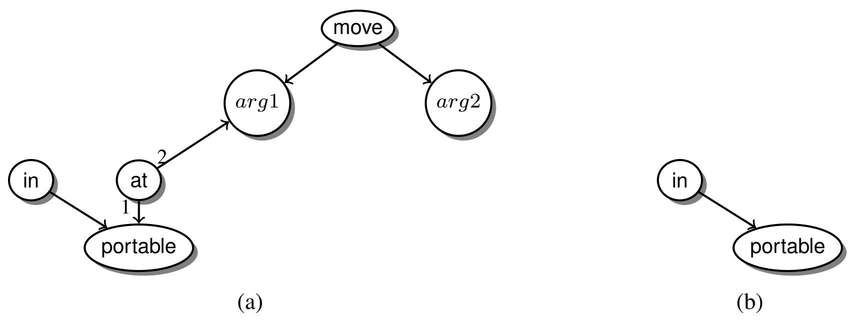


Figure 4: Valid (a) and invalid (b) subgraphs of Figure 2c.

2011). First we label object nodes. Objects which are arguments of the action are labelled with their position in the action parameter list, so that the object which is the nth argument of the action is labelled "arg-n". Other objects are labelled with their type. 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 4a, `at` is a core fluent and `in` 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. `(at portable arg1)` could convert to "at1(portable)2(arg1)". (Note that here "portable" 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.

5. Learning

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 2, consider the following fluents:

1. `(at [A] arg`$_1$`)`
   where $[A] = \{x : at(x, arg_1) \wedge in(x) \wedge \neg at(x, arg_2)\}$

2. `(at [D] arg`$_1$`)`
   where $[D] = \{x : at(x, arg_1) \wedge \neg in(x) \wedge \neg at(x, arg_2)\}$

3. `(at [Z] arg`$_1$`)`
   where $[Z] = \{x : at(x, arg_1) \wedge \neg at(x, arg_2)\}$

Fluents (1) and (2), present in the graph, would each have their own classifier. Additionally we must consider fluents with more general deictic terms, such as (3), which includes both (1) and (2). The classifier associated with (3) predicts whether fluent `(at x arg`$_1$`)` changes for *any* x for which `(at x arg`$_1$`)` holds, whereas the classifiers associated with (1) and (2) predict whether `(at x arg`$_1$`)` changes for x which is in the briefcase (1), or for x which is not in the briefcase (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 & 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, setting the voted perceptron parameter for the number of epochs T to 1. 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.

5.1 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. 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 the briefcase domain, one of the sets of classifiers would be $C_{(move,at)}$: the set of classifiers which predict changes to the `at` predicate when the `move` action is performed. A member of $C_{(move,at)}$ could be $c_{(\{x:at(x,arg_1)\wedge\neg in(x)\},arg_1)}$, which predicts whether `(at x arg`$_1$`)` changes under the `move` action when x satisfies $at(x, arg_1) \wedge \neg in(x)$.

5.2 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 for the classifiers to learn.

Algorithm 1 Training

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

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

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

In $\mathit{updateClassifiers}$, classifiers which predict for $p(\overline{m})$ 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, $\mathit{updateClassifiers}$ 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 $q(\{x : a(x)\})$ 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 $p(\overline{n})$:

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

Algorithm 2 Prediction

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

function $getPrediction$(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 $subsetMatch(c, \overline{m})$ and $r < getReliability(c)$ then
4:         $y := predict(c, x)$
5:         $r := getReliability(c)$
6: return $y$
```

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 $p(\overline{n})$: $intersectMatch(c, \overline{m})$ holds 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 ($createClassifiers$).

5.3 Reliability and Prediction

The training algorithm maintains a reliability score for each classifier ($updateReliability$), 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, $getReliability$ returns either the classifier reliability or the null reliability, whichever is higher. If the null reliability is higher $predict$

(is_at arg$_1$) changes when:
[332] (is_at arg$_1$)

(is_at arg$_2$) changes when:
[346] (AND (is_at arg$_1$) (NOT(is_at arg$_2$)) ***(at x arg$_1$) (in x))***
where $x \in \{x : (at\ x\ arg_1) \wedge (in\ x)\}$

[48] ***(AND (NOT(is_at arg$_1$)) (is_at arg$_2$) (at x arg$_1$) (NOT(in x)) (NOT(at x arg$_2$)))***
where $x \in \{x : (at\ x\ arg_1) \wedge \neg(at\ x\ arg_2) \wedge \neg(in\ x)\}$

(at x arg$_1$) changes when:
[85] (AND (is_at arg$_1$) ***(is_at arg$_2$)*** (at x arg$_1$) (NOT(at x arg$_2$)) (in x))
where $x \in \{x : (at\ x\ arg_1) \wedge \neg(at\ x\ arg_2) \wedge (in\ x)\}$

(at x arg$_2$) changes when:
[169] (AND (is_at arg$_1$) (at x arg$_1$) (NOT(at x arg$_2$)) (in x))
where $x \in \{x : (at\ x\ arg_1) \wedge \neg(at\ x\ arg_2) \wedge (in\ x)\}$

Figure 5: Per-effect rules generated for the Briefcase move action from 1000 examples in a world with 1% noise and 50% observability. Weights are shown in square brackets. Fluents in bold are neither in, nor implied by, the true action specification. Such fluents arise because these rules are derived from support vectors which may contain noisy fluents. Many of these fluents will later be excluded by the rule combination process (Section 6.2).

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.

6. Deriving Planning Operators

Once the classifiers have been trained, the first step in deriving explicit action rules is to extract individual *per-effect* rules to predict each fluent in isolation. We will look at how to combine the rules to extract postconditions in Section 6.2.

6.1 Rule Extraction

The rule extraction process takes as input a classifier $c_{\overline{m}} \in C_{a,p}$ and returns a set of preconditions (as graphs) which predict that the fluent $p(\overline{m})$ will change if action a is performed. For example, in the Briefcase domain a set of preconditions extracted from the classifiers for move is shown in Figure 5.

Rules are extracted from a voted perceptron with kernel K and a set of support vectors SV. The 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 as a scoring function to determine which nodes to remove. The *score* of any possible state graph $\mathbf{x}$ is defined to be the value calculated by the voted perceptron's prediction calculation before thresholding (Freund & Schapire, 1999):

$$score_{p(\overline{m})}(\mathbf{x}) = \sum_{i=1}^{n} c_i \; sign \sum_{j=1}^{i} y_j \alpha_j K(\mathbf{x}_j, \mathbf{x})$$

where each x_i is one of the n support vectors of the classifier $c_{a,p}$, y_i is the corresponding target value, c_i and α_i are the parameters learnt by the classifier, and $p(\overline{m})$ is the effect predicted by the classifier. The predicted value for $\mathbf{x}$ is 1 if $score_{p(\overline{m})}(\mathbf{x}) > 0$ and -1 otherwise. So that the scoring function is consistent with the classification of examples, for rule extraction, all the training examples, including the support vectors, are reclassified according to the classes assigned by the voted perceptrons.

The basic intuition behind the rule extraction process is that more discriminative fluents will contribute more to the score of an example. Thus the rule extraction process operates by taking each support vector and repeatedly deleting the fluent which contributes least to the score until some stopping criterion is satisfied. This leaves the most discriminative fluents underlying the example, which can be used to form a precondition. In general there will also be some deictic terms (and *all* their associated fluents) which will be irrelevant to the underlying rule, and so we can speed up the rule extraction process by first deleting deictic terms in a similar fashion: we repeatedly delete the deictic term which contributes least to the score until the stopping criterion is satisfied.

We introduce some terminology to support the description of the rule extraction process. A *child* of graph $\mathbf{x}$ is any distinct graph obtained by deleting either a single fluent node or a single object node in $\mathbf{x}$. Similarly, a *parent* of $\mathbf{x}$ is any graph obtained by inserting a fluent or object node. Deleting an object node also entails deleting all connected fluent nodes, as these are meaningless without the object node, while deleting a fluent node may lead to the deletion of an object node, if the object node no longer has a corresponding deictic term.

The rule extraction process is given in Algorithm 3, as follows. Take each support vector v in turn, and aim to find a subgraph $rule_v$ which covers v and does not cover any training examples in a different class, but where every child of $rule_v$ does cover at least one example in a different class. Construct $rule_v$ by a greedy algorithm which first takes v as a candidate rule and then repeatedly creates a new candidate rule by choosing one node to delete. Initially nodes to delete are selected from the set of object nodes in v. Once no further object nodes can be deleted, nodes are selected from the set of fluent nodes in v. In either case, the node is chosen as the node n whose removal from the current candidate rule gives the child $child_n$ with the highest score:

$$\underset{n \in nodes}{\operatorname{argmax}} \, score_{p(\overline{m})}(child_n).$$

Removing the resulting node n removes the least discriminative feature, or set of features, in the current candidate rule. After each removal, the new candidate rule is tested against the training examples. If it misclassifies a training example, then the rule is too general, $rule_v$ is set to the previous candidate rule, and the process terminates; otherwise the process repeats. The result is a set of rules for each action, predicting whether a particular fluent changes. There may be many rules, up to one per support vector, each consisting of a set of preconditions which, if satisfied, predict whether a fluent will change.

Algorithm 3 Rule extraction

```
Require: Support vectors $SV$ for classifier $c_{\overline{m}} \in C_{a,p}$
Ensure: Rules $R = \{rule_v : v \in SV\}$
1: for $v \in SV$ do
2:     $objectNodes$ := set of object nodes in $v$
3:     $fluentNodes$ := set of fluent nodes in $v$
4:     $rule_v := eliminateNodes(rule_v, objectNodes)$
5:     $rule_v := eliminateNodes(rule_v, fluentNodes)$

function $eliminateNodes$(rule candidate $rule$, nodes $nodes$)
1: $child := rule$
2: while $child$ only covers training examples of same class do
3:     $parent := child$
4:     $maxnode := \underset{n \in nodes}{\operatorname{argmax}}\, score_{p(\overline{m})}(parent \setminus n)$
5:     $child := parent \setminus maxnode$
6:     $nodes := nodes \setminus maxnode$
7: return $parent$
```

6.2 Rule Combination

The rule extraction process described above produces a set of rule fragments for an action, where each rule fragment consists of some preconditions and a single effect, such as for the Briefcase `move` action shown in Figure 5. However, we seek compact PDDL-style rules for each action, consisting of a set of preconditions and a set of effects, as in the definition of the Briefcase `move` action given in Section 2.

Intuitively, we want to merge together the rule fragments for each action to create preconditions and effects. However the merging process is confounded by inconsistencies between rule fragments, caused by noise in the data, disjoint preconditions, conditional effects and probabilistic outcomes. We therefore propose a greedy heuristic to merge the rule fragments while also accounting for inconsistencies. Our algorithm uses the classifier decision functions and coverage on the training data to determine whether one rule fragment is a better predictor than another, and whether one fluent value is a better predictor than another. As in the rule extraction algorithm, a higher decision function is taken to indicate better predictive value (since it indicates a precondition which is further from the decision boundary). Coverage is used to ensure that the generated rules cover at least some of the training data.

The algorithm operates by maintaining a set of candidate planning operators for an action, initially empty. At each step it selects the best unmerged rule fragment to merge with the current candidate. The consistency of the rule fragment with the current candidate is tested, on a fluent by fluent basis. Fluents in the rule fragment can be rejected, or added to the current candidate, possibly changing the previous value in the candidate, depending on decision function and coverage tests. Similarly the entire rule fragment may be rejected, to be considered for later addition as a conditional effect of the same candidate, a disjunctive precondition or a probabilistic outcome. The remainder of this section describes the rule combination algorithm in more detail.

Algorithm 4 Rule Combination

```
1: $R := \{(g_1, e_1), \ldots, (g_r, e_r)\}$
2: $rule := (g_1, \emptyset)$
3: $locks = \emptyset$
4: **while** $R \neq \emptyset$ **do**
5:     $next :=$ highest scoring rule in $R$
6:     $R := R \setminus \{next\}$
7:     **for** each condition $(g_{rule_i}, e_{rule_i})$ in $rule$ **do**
8:         $g_{candidate} := CombinePrecons(rule, next, locks)$
9:         $g_{candidate} := SimplifyPrecons(rule, next, g_{candidate})$
10:        **if** $!ProcessConditionals(i, rule, next, g_{candidate})$ **then**
11:            **if** $AcceptPrecons(rule, g_{candidate})$ **then**
12:                $g_{rule_i} := g_{candidate}$
13:            **if** $AcceptEffect(rule, e_{next})$ **then**
14:                $e_{rule_i} := e_{rule_i} \cup e_{next}$
15:            $e_{rule_i} := SimplifyEffects(rule)$
```

6.2.1 PRELIMINARIES

The rule combination process builds PDDL-like rules for each action, taking as input the set of rules produced by the rule extraction process: $\{(g_1, e_1), (g_2, e_2), \ldots, (g_r, e_r)\}$ where g_i is the graph representing the i-th set of preconditions, and e_i is the marker node indicating the change when g_i holds. Rule combination generates a rule (g_{rule}, e_{rule}) by progressively merging the precondition graphs and marker nodes. Where conditional effects arise, the secondary preconditions and effects are given by additional *(graph,marker nodes)* pairs. Once rule combination is complete, then given the definitions in Section 4, we can directly convert a state graph g_{rule} and its effects e_{rule} into a precondition and effect in PDDL format.[5]

The combination process relies on being able to repeatedly merge the current candidate rule with a per-effect rule, which requires determining a mapping between the nodes in the two graphs: there may be multiple possible mappings. Furthermore, when learning from noisy examples, unwanted additional fluents can be introduced to the per-effect rules via noisy support vectors. Similarly, incomplete training examples can mean some necessary fluents are missing from individual per-effect rules. In this section we describe an approach to merge state graphs while also identifying and eliminating fluents introduced by noise, and adding in fluents omitted due to partial observability.

To support the process of choosing between different potential rules which may contain noisy fluents, or omit necessary fluents, we introduce two filtering functions. *AcceptPrecons* takes an existing precondition and effects (g_{rule}, e_{rule}) for an action a, and assesses whether a new precondition g_{new} predicts the effects of a at least as well as the current precondition. *AcceptEffects* takes an existing precondition and effects for an action a, and assesses whether the precondition predicts a new effect e_{new} of a at least as well as it predicts the current effects. We describe *AcceptPrecons* and *AcceptEffects* in detail later in this section.

5. Since the effects in e_{rule} are changes it is sometimes necessary to identify from what value the change is made, by referring to the support vector from which the effect marker originated.

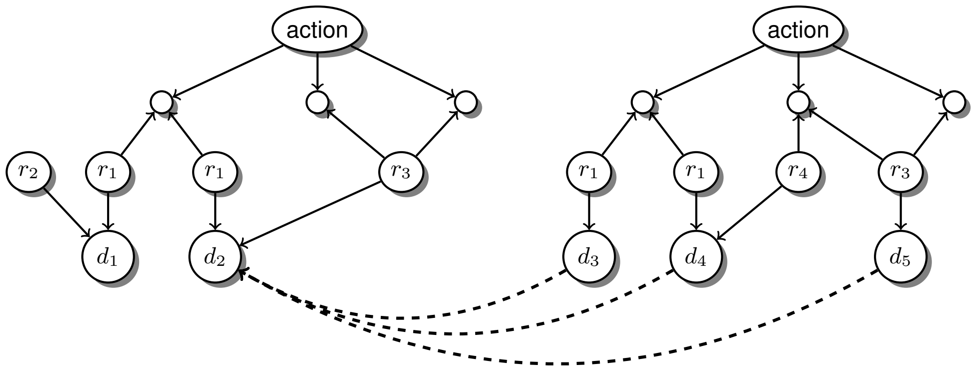


Figure 6: An example of matching deictic terms in the current rule g_{rule} (left) with those in the candidate rule g_{new} (right). Three deictic terms in g_{new} could match deictic term d_2 in g_{rule}.

6.2.2 COMBINING PRECONDITIONS

CombinePrecons attempts to combine g_{rule} with g_{next}. The first step is to identify which deictic terms in g_{rule} correspond to which deictic terms in g_{next}, if any. This is straightforward where deictic terms match exactly, but when there are subset or intersect matches there can be multiple deictic terms in g_{next} which are possible matches for a single deictic term in g_{rule}. Similarly a single deictic term in g_{next} could match several terms in g_{rule} (e.g. Figure 6).

We therefore construct a set of possible mappings between terms in g_{rule} and g_{next}, and use the score calculation to choose one mapping to use. First we match any deictic terms which correspond to the arguments of the action, constants or function terms, as these are unambiguous matches. Next we consider every possible mapping m of the remaining deictic terms. where m is an injective function from the set of object nodes in g_{next} to the set of object nodes in g_{rule}.

The number of mappings depends on the number of nodes in the two graphs, however at this stage we expect the graphs to be relatively small because they have already been reduced by rule extraction, and the underlying rules are compact (by assumption). Furthermore two deictic terms may only be mapped to one another if they have compatible types, and they intersect on at least one fluent. Each mapping is used to construct a new candidate rule $g_{candidate}$, which is scored using the classifier for e_{next}. We select the candidate rule with the highest score.

We construct $g_{candidate}$ from a mapping m as follows. We first initialise $g_{candidate}$ to g_{rule}. There may be some object nodes in g_{next} which are not mapped by m into $g_{candidate}$. For each such $o_{next,i}$ in g_{next} we insert a new node o_i into $g_{candidate}$ and define $m(o_{next,i}) = o_i$. Next we merge the fluents in g_{next} with the fluents in $g_{candidate}$. A fluent f in g_{next} matches a fluent f' in $g_{candidate}$ if the arguments of f are mapped to the arguments of f', that is $\forall i \ args_i(f) = m(args_i(f'))$. Whenever there is a fluent f in g_{next} without a matching f' in $g_{candidate}$ we insert a new fluent node f' with links to object nodes so that $\forall i \ args_i(f) = m(args_i(f'))$. Finally, we create a list of conflicting fluents. A fluent f in g_{next} is conflicting if its value is different from the value of its matching fluent in $g_{candidate}$.

Now we check whether e_{next} contradicts any effect in e_{rule}. Effects conflict if both rules predict change to a fluent, but the rules have different values for the fluent in their preconditions. If there is a conflict, the effect is rejected but we continue with the process of combining the preconditions, as the contradictory effect may simply be noise.

Next *CombinePrecons* decides which value each fluent in the conflicts list should take in $g_{candidate}$, as follows. For each conflicting fluent, there are three possible values the fluent could take in the preconditions of the true rule: $*$ (unobserved), *true* or *false*. The score $score_e$ (for each effect e in e_{rule}) of each variant is calculated (with the values of other conflicting fluents set to $*$). The preferred variant is where the value is $*$, indicating a non-discriminative feature, and giving the simplest precondition. However, a variant is only acceptable if the score of the resulting precondition is positive for all effects in e_{rule}, since then the new precondition still predicts the same effects as the current precondition g_{rule}. If accepted, the fluent is locked at the $*$ value, to prevent later, possibly noisy rules, from resetting it. Locked fluents are recorded in the $locks$ variable (see Algorithm 4). If the $*$-variant is unacceptable, then the *true*-valued or *false*-valued cases are considered, provided they have positive scores on all the effects. If both variants are acceptable, whichever has the highest average score over all the effects is selected. If neither variant is acceptable then the conflict is unresolved for this fluent. As long as the conflicts on every fluent are resolved, the rule combination process can continue with the new candidate precondition. If not, the current rule is rejected (and *CombinePrecons* returns g_{rule}).

6.2.3 SIMPLIFYING PRECONDITIONS

Once *CombinePrecons* has generated a candidate precondition $g_{candidate}$, *SimplifyPrecons* considers alternative, less specific preconditions. It creates a set of alternatives $g_{candidate\setminus f}$ for each fluent f in $g_{candidate}$ which differs from g_{rule}. $g_{candidate\setminus f}$ is constructed by deleting f from $g_{candidate}$. Whenever the filtering function $AcceptPrecons$ rates $g_{candidate}$ as worse than any $g_{candidate\setminus f}$, f is removed from $g_{candidate}$.

6.2.4 CONDITIONAL EFFECTS

When considering the next rule fragment (g_{next}, e_{next}) to be merged with the current rule candidate, *ProcessConditionals* considers whether the rule fragment is part of the main preconditions of the rule, or part of the secondary preconditions for some conditional effect. There are four possibilities when merging g_{next} with the current candidate preconditions $g_{candidate}$:

1. $g_{candidate}$ is part of the secondary preconditions and g_{next} part of the main preconditions;
2. g_{next} is part of the secondary preconditions and $g_{candidate}$ part of the main preconditions;
3. both are part of the main preconditions for the underlying rule; or
4. g_{next} is not part of the underlying rule and is rejected.

We note that if the true underlying rule has a conditional effect then merging one or more secondary preconditions with main preconditions will cause coverage to drop. A drop in coverage of different preconditions can be identified by considering the relative F-scores of the preconditions when predicting different effects. In this case we compare F-scores on the new effects e_{next} versus the currently accepted effects e_{rule}.

Consider case (1) where $g_{candidate}$ is part of the secondary preconditions and g_{next} part of the main preconditions. When we merge g_{next} with $g_{candidate}$, to form g_{new}, the F-score of g_{new} on the existing effects $e_{candidate}$ should remain the same as the F-score of $g_{candidate}$, because the set of examples which g_{new} covers should be the same as the set of effects covered by $g_{candidate}$. However, the F-score of g_{new} on the new effect e_{next} should be higher than $g_{candidate}$ because if g_{next} is part of the main preconditions it will cover examples where e_{next} changes but $e_{candidate}$ (conditional on $g_{candidate}$) does not.

Now consider case (2) where g_{next} is part of the secondary preconditions and $g_{candidate}$ part of the main preconditions. Here, conversely, the F-score of g_{new} on the existing effects $e_{candidate}$ will be lower than $g_{candidate}$, because the inclusion of g_{next} means that g_{new} covers fewer examples than $g_{candidate}$. However, the F-score of g_{new} on the new effect e_{next} will be higher than $g_{candidate}$ because only some of the examples covered by $g_{candidate}$ will have the conditional change to e_{next}.

If neither (1) nor (2) holds, we test whether g_{new} and $g_{candidate}$ have similar F-scores for each effect in e_{new}: if so then the new precondition is accepted, otherwise it is rejected.

6.2.5 SIMPLIFYING EFFECTS

In light of the new preconditions, *SimplifyEffects* tests if any of the effects should be removed from the new g_{rule} . For instance, more specific preconditions may lower the incidence of some effects (as seen in the training data) to the extent that *AcceptEffects* rejects them. Each effect is tested against all the other effects by *AcceptEffects* and, if rejected, removed from e_{rule}.

6.2.6 PRECONDITION FILTERING

In *AcceptPrecons*, the precondition filtering function, before even making a comparison between preconditions, the new precondition g_{new} must be checked to ensure that it is consistent with the classifiers and supported by the training data. For this we require a notion of coverage of the training set. Coverage is defined to account for partial observability, so that precondition g_{new} covers example x at effect e (denoted $covers_e(g_{new}, x)$) if none of the fluents in the example state contradict the fluents in the rule preconditions, and e is in both the example state changes and the rule effects. Now g_{new} can form a rule precondition if:

1. g_{new} is consistent with the classifiers: for each $e \in e_{rule}$, g_{new} should be classified by the classifier $C_{a,e}$ as predicting change, that is,
   $\forall e \in e_{rule}\ score_e(g_{new}) > 0$; and

2. g_{new} is supported by the training data: for each $e \in e_{rule}$, g_{new} should cover at least one training example where e changed, that is,
   $\forall e \in e_{rule}\ |\{x : covers_e(g_{new}, x)\}| > 0$.

Both should be considered, as score alone may permit rules which do not cover any training examples, while coverage alone may allow negatively weighted rules.

Additionally, *AcceptPrecons* uses differences in precision and recall to identify and reject any new precondition which performs significantly worse than the existing precondition. It rejects preconditions where either the precision or recall on the training set drops substantially for any $e \in e_{rule}$. Since precision and recall is a trade-off, the comparison is made using the F-score[6] for precondition pre at effect e: $F_{pre,e}$. Ideally we want the new precondition to improve on (or at least not worsen) the F-score, but we must introduce some tolerance to account for the effects of

6. F-score is the harmonic mean of precision and recall (true positives/predicted changes and true positives/actual changes, respectively) (Van Rijsbergen, 1979).

noise. Thus we allow the new F-score $F_{g_{new},e}$ to drop to some fraction ϵ_p of the F-score for the existing precondition $F_{g_{rule},e}$, for any effect $e \in e_{rule}$. For new F-scores below this value, the new precondition is rejected.

Finally, *AcceptPrecons* performs a complexity check on the preconditions. If the new preconditions involve more objects than before, then the resulting rule will be more complex. This additional complexity is only accepted if there is a worthwhile improvement in F-score: if, for each effect $e \in e_{rule}$, the old F-score is more than a fraction ϵ_p of the new F-score then the new preconditions are rejected.

6.2.7 EFFECTS FILTERING

The effects filtering function, *AcceptEffect*, similarly compares F-scores. Given (g_{rule}, e_{rule}) it compares how well g_{rule} predicts a new effect e_{new} relative to how well it predicts each $e \in e_{rule}$: specifically it compares $F_{g_{rule},e_{new}}$ to $F_{g_{rule},e}$ for each $e \in e_{rule}$. This identifies effects which are inconsistent with the other effects in terms of precision and recall. In particular, effects which occur in far fewer examples than other effects are identified in this way: these are likely to be caused by noise, or could be conditional effects. An effect is rejected by the function if its F-score is less than some fraction ϵ_e times the F-score on any other effect of the same rule.

In our evaluation, ϵ_p and ϵ_e were set to 0.95 and 0.5 respectively, and not varied across the domains. The values were selected empirically via experiments on a holdout dataset from one experimental domain (ZenoTravel).

6.2.8 PROBABILISTIC OUTCOMES

We can extend the rule combination process to generate alternative, less probable outcomes. Taking a newly derived planning operator, we consider all the training examples which match the planning operator's preconditions. Some examples will also match the planning operator's effects and so are fully covered by it. Other examples will have different outcomes which we now aim to cover by generating alternative effects for the planning operator.

Alternative effects are sourced from the rule fragments used in constructing the original planning operator. Some rule fragments contributed only to the operator's preconditions, while others contributed to both. The rule fragments which only contributed to the preconditions thus provide a source of alternative effects for the planning operator, as the effects for those fragments were not used in the final operator.

We take a sequential covering approach. An alternative rule is created by reusing the precondition from the planning operator, and progressively combining effects from the rule fragments. Each new effect is accepted if the resulting rule gives better F-scores than the previous rule on the remaining uncovered training examples. Once all the rule fragments have been considered, those fragments which contributed to the new rule are deleted, and training examples which are covered by the new rule are removed. The process is then repeated until no further rules can be created. If unused rule fragments remain, a noise outcome is added (if there are rule fragments with effects which change fluent values) and a "no change" outcome is added (if there are rule fragments without such effects).

We now have one or more planning operators consisting of an action name, its arguments, a precondition and a set of possible outcomes. It remains to assign probabilities to these outcomes. Following Pasula et al. (2007), we use conditional gradient descent (in conjunction with the Armijo rule and parameters $s = 1.0$, $\beta = 0.1$ and $\sigma = 0.01$) to assign probabilities to the outcomes based on the training data.

6.2.9 LEARNING FROM PLANS

An alternative setting to learning from exploration traces is to learn from plans (Yang, Wu, & Jiang, 2007; Zhuo, Yang, Hu, & Li, 2010). Zhuo et al.(2010) discuss how such data may arise, for example, in the logs of operating systems or web services, where sequences of planned actions and partial state observations may be found, but where the underlying domain model may be unavailable.

Unlike exploration traces, plans typically do not contain examples of action failures, presenting a difficulty for our classification-based learning method, which requires negative examples. We tackle this problem by preprocessing the plan data before learning, as follows. Given an action instance, the probability that a randomly selected state will satisfy the preconditions for the action is very small. Therefore for each action instance a in the plan, we sample uniformly at random a state s from the set of input plans. We assume that s does not satisfy the action preconditions and so s does not change as a result of a. This gives the negative example (s, a, s) which is added to the training data. Although on rare occasions s may in fact satisfy the preconditions for a, our approach is designed to tolerate the presence of such noisy training examples.

However, this process will not elicit a particular type of negative example, namely where static predicates hold for some of the action arguments, and these predicates also occur in the action preconditions. For example, in the ZenoTravel domain, fuel levels of the planes range from *FL1,FL2,. . .,FL6* and the (static) *next* predicate indicates the ordering, so that *next(FLi,FL(i+1))* always holds for each i but *next* is false for any other pairing. Every successful *REFUEL* action takes a pair of consecutive fuel levels as arguments, and thus every negative *REFUEL* example generated by the process above will also have consecutive fuel levels in its argument list, as changing the state will never change the value of *next*. Effectively this excludes all cases of *REFUEL* with non-consecutive fuel-levels in the argument list, which means that the learning algorithm cannot learn that the precondition for *REFUEL* includes the requirement that the fuel-levels are consecutive. We address this problem by first preprocessing to identify static predicates in the training data. Then whenever all the arguments of any static fluent are contained in the set of arguments of an action, we add a new action with the same prior state and arguments, but we replace one argument uniformly at random from the set of possible arguments of the same type. For example, this would arise for the *REFUEL(PLANE1, CITY1, FL5, FL6)* action since *FL5* and *FL6* are the arguments of the static fluent *next(FL5,FL6)*. As before, we assume this new action is unsuccessful and set the successor state to be equal to the prior state.

Since adding many negative examples for each positive example will unbalance the dataset, for each action we randomly select one of the negative examples generated by the processes above. Only this action is added to the data as a new negative example.

7. Evaluation

In this section we present our results on learning action models in domains selected from the International Planning Competition. We also compare our results to a state-of-the-art method for learning action models from exploration traces (Pasula et al., 2007) using their test domains. We describe the domains and how data was generated for them below, and then present our experiments and results.

```
(define (domain trucksanddrivers)
  (:requirements :probabilistic-effects)
  (:predicates (at ?t ?l)
               (in ?o ?t)
               (driving ?d ?t)
               (path ?l ?l)
               (link ?l ?l)
               (empty ?t))

(:action load
 :parameters (?o ?t ?l)
 :precondition (and (at ?t ?l) (at ?o ?l))
 :effect (probabilistic 0.9 (and (not(at ?o ?l)) (in ?o ?t))))

(:action unload
 :parameters (?o ?t ?l)
 :precondition (and (at ?t ?l) (in ?o ?t))
 :effect (probabilistic 0.9 (and (at ?o ?l) (not(in ?o ?t)))))

(:action board
 :parameters (?d ?t ?l)
 :precondition (and (at ?t ?l) (at ?d ?l) (empty ?t))
 :effect (probabilistic 0.9 (and (not(at ?d ?l)) (driving ?d ?t) (not(empty ?t)))))

(:action disembark
 :parameters (?d ?t ?l)
 :precondition (and (at ?t ?l) (driving ?d ?t))
 :effect (probabilistic 0.9 (and (not(driving ?d ?t)) (at ?d ?l) (empty ?t))))

(:action drive
 :parameters (?t ?fr ?to ?d)
 :precondition (and (at ?t ?fr) (driving ?d ?t) (link ?fr ?to))
 :effect (probabilistic 0.9 (and (not(at ?t ?fr)) (at ?t ?to))))

(:action walk
 :parameters (?d ?fr ?to)
 :precondition (and (at ?d ?fr) (path ?fr ?to))
 :effect (and (not(at ?d ?fr))
         (probabilistic 0.9 (at ?d ?to))
         (probabilistic 0.1 (exists ?x (path ?fr ?x)) (at ?d ?x) ))))
```

Figure 7: PPDDL description of the trucks and drivers domain.

7.1 Data

Pasula et al. (2007) experimented with the Trucks and Drivers domain (Figure 7), a probabilistic version of the IPC Driverlog domain, and the Slippery Gripper domain, a form of the BlocksWorld domain (Figure 8). From the PPDDL domain descriptions we generated ten sets of training and testing data, in the form of sequences of random actions and resulting states, using the Random Action Generator 0.5[7] modified to also generate action failures. As in Pasula et al.'s work, in our data, each action's preconditions are satisfied approximately 50% of the time. For the Trucks and Drivers domain, we used a world with two trucks, two drivers, two objects and four locations. For the Slippery Gripper domain, we used a world with four blocks.

Finally we generated data for the IPC Briefcase domain using the Random Action Generator as before. We selected the Briefcase domain as it contains actions with conditional effects. Ten se-

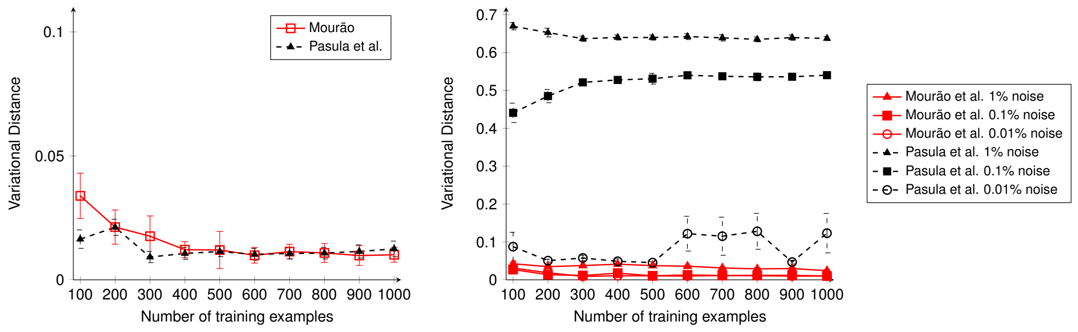
7. http://magma.cs.uiuc.edu/filter/

```
(define (domain slipperygripper)
  (:requirements :probabilistic-effects)
  (:predicates (inhand ?x)
               (clear ?x)
               (on ?x ?y)
               (block ?x)
               (table ?x))

(:action pickup
 :parameters (?x ?y)
 :precondition (and (on ?x ?y) (clear ?x) (inhand NIL) (block ?x))
 :effect (when (and (block ?y) (not (wet)))
            (probabilistic 0.7 (and (inhand ?x) (not (clear ?x)) (not (inhand NIL))
                                    (not (on ?x ?y)) (clear ?y)))
            (probabilistic 0.2 (and (on ?x TABLE) (not (on ?x ?y)) (not (inhand NIL)))))

         (when (and (block ?y) (wet))
            (probabilistic 0.33 (and (inhand ?x) (not (clear ?x)) (not (inhand NIL))
                                    (not (on ?x ?y)) (clear ?y)))
            (probabilistic 0.33 (and (on ?x TABLE) (not (on ?x ?y)) (not (inhand NIL)))))

         (when (and (table ?y) (not (wet)))
            (probabilistic 0.5 (and (inhand ?x) (not (clear ?x)) (not (inhand NIL))
                                    (not (on ?x ?y)) (clear ?y))))

         (when (and (table ?y) (wet))
            (probabilistic 0.8 (and (inhand ?x) (not (clear ?x)) (not (inhand NIL))
                                    (not (on ?x ?y)) (clear ?y)))))

(:action puton
 :parameters (?x ?y)
 :precondition (inhand ?x)
 :effect (when (and (clear ?y) (block ?y))
            (probabilistic 0.7 (and (inhand NIL) (not (clear ?x)) (not (inhand ?y)) (on ?x ?y)
                                    (clear ?x)))
            (probabilistic 0.2 (and (on ?x TABLE) (clear ?x) (inhand NIL) (not (inhand ?x)))))

         (when (table ?y)
            (probabilistic 0.8 (and (on ?x TABLE) (clear ?x) (inhand NIL) (not (inhand ?x))))))

(:action paint
 :parameters (?x)
 :precondition (block ?x)
 :effect (probabilistic 0.6 (painted ?x))
         (probabilistic 0.1 (and (painted ?x) (wet))))

(:action dry
 :parameters ()
 :precondition ()
 :effect (probabilistic 0.9 (not (wet))))
```

Figure 8: PPDDL description of the slippery gripper domain.

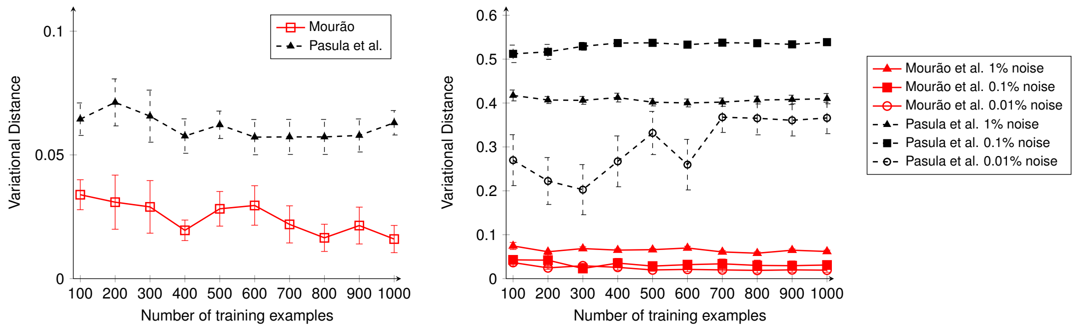
quences of random actions and resulting states were generated from the PDDL domain descriptions and used as training and testing data. We used 50 objects and locations for training, and 100 objects and locations for testing.

In line with previous work (Amir & Chang, 2008), incomplete observations were simulated by randomly selecting a fraction (99%, 95% or 90%) of fluents (including negations) from the world to observe after each action. The remaining fluents were discarded and a reduced state graph was generated from the observed fluents. Similarly, to simulate noisy observations, with some



(a) Learning without noisy observations (b) Learning with noisy observations

Figure 9: Comparison of the proposed rule learning approach with results of Pasula et al. (2007), using the drive action from the Trucks and Drivers domain.



(a) Learning without noisy observations (b) Learning with noisy observations

Figure 10: Comparison of the proposed rule learning approach with results of Pasula et al. (2007), using the walk action from the Trucks and Drivers domain.

probability p (1%, 0.1% or 0.01%), the value of each fluent in an observation was flipped to the opposite value.

7.2 Experiments

7.2.1 PROBABILISTIC DOMAINS

Pasula et al.'s approach has been implemented as the learning component of libPRADA, a C++-library for model-based relational reinforcement learning in stochastic domains (Lang & Toussaint, 2010). In our experiments we use version 1.2 (July 2012). In order to reproduce the results of Pasula et al. we used the conditional gradient descent algorithm described in their paper in place of libPRADA's gradient descent calculation.

We compare our results to those of Pasula et al.(2007), calculating the average variational distance between the true model P and the learnt model Q over sets of test examples $\mathbf{E}$:

$$VD(P,Q) = \frac{1}{|\mathbf{E}|} \sum_{E \in \mathbf{E}} |P(E) - Q(E)| \tag{1}$$

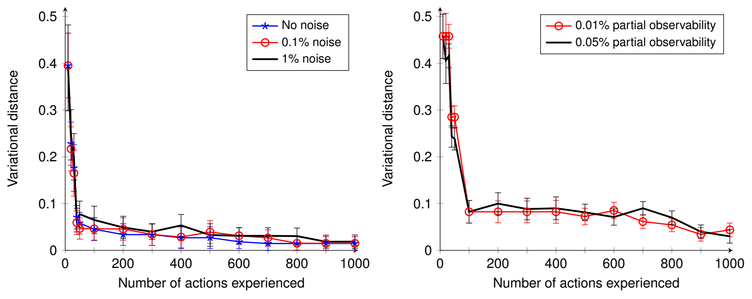


Figure 11: Briefcase results

In Figures 9 and 10 we present results from the Trucks and Drivers domain for the *drive* and *walk* actions. Without noisy observations, both approaches learn probabilistic rulesets with low variational distance from the true model. When the observations are noisy the performance of Pasula et al.'s approach degrades rapidly, even with the low levels of noise present in the experiment. In contrast, the performance of our approach only degrades slightly at low noise levels.

7.2.2 IPC DOMAINS

We additionally evaluated our method on domains taken from the International Planning Competition, again measuring performance using variational distance, and considering different levels of noise and partial observability. Our results for the Briefcase domain, a conditional effects domain, are shown in Figure 11, showing that our method is effective with noisy or incomplete observations. Although the Briefcase domain is not probabilistic, and our method generates probabilistic operators, the probabilities are sufficiently accurate under noise and incompleteness that the variational distance measure is little affected.

8. Discussion

In recent years several new approaches have been developed for the problem of learning domain models, addressing learning in a variety of different representational and environmental settings. The choice of representation includes how expressive the learnt action models can be (e.g. STRIPS versus PDDL), and whether the state is described with symbolic predicates, continuous-valued attributes or a mixture. The environmental setting determines whether observations and actions are subject to noise, incompleteness or non-determinism, and whether training data is available as plan traces, observation traces, or independent *(state,action,state)* tuples. In contrast to other approaches, the approach presented in this paper handles a wide range of environmental settings, while also supporting relatively expressive domain representations. Below we discuss how alternative approaches relate to the approach presented in this paper, in terms of the representational and environmental settings to which they apply, before considering future research directions in this area.

8.1 Related work

Very few approaches can learn domain models at the full level of expressiveness supported by PDDL (nor indeed do many planners support the full PDDL language). Most do support at least STRIPS models, but functions, logical implications, conditional effects and quantification may only be supported to some degree. In the method we have presented here, conditional effects, negative preconditions and some types of quantification are supported: we can learn effects with universally quantified fluents, and preconditions with existentially quantified fluents. Functions can be supported by setting up an extension to the graphical representation (Mourão, 2013).

Deictic reference is a fundamental aspect of our learning algorithm, and the expressiveness of the rules we learn is similar to other approaches which also use deixis (Pasula et al., 2007; Rodrigues, Gérard, & Rouveirol, 2010a; Rodrigues, Gérard, Rouveirol, & Soldano, 2010b). The approach of Pasula et al. (2007) supports many aspects of PDDL, including functions and preconditions with complex concepts such as quantification. The ILP-based approach of Rodrigues et al. (2010a) uses an extended deterministic STRIPS model, supporting disjunctive preconditions and conditional effects. LAMP (Zhuo et al., 2010) supports more expressive action models, generating action representations with conditional effects, quantifiers and logical implications, and works from partial observations, but requires noiseless plan traces to do so.

A critical aspect of learning in real-world domains is the ability of the learning method to tolerate uncertainty. Most recent work has considered some aspect of uncertainty in the world, such as noisy observations (Mourão, Petrick, & Steedman, 2010; Rodrigues et al., 2010a); incomplete observations (Yang et al., 2007; Amir & Chang, 2008; Zhuo et al., 2010; Mourão et al., 2010); noisy action effects (Pasula et al., 2007); and non-deterministic action effects (Pasula et al., 2007). However, the approach we have presented in this paper is the first to tolerate all of these forms of uncertainty, either individually or in combination.

Some research has focussed on learning action models from successful plan traces, (McCluskey, Richardson, & Simpson, 2002; Yang et al., 2007; McCluskey, Cresswell, Richardson, & West, 2009; Zhuo et al., 2010; Cresswell & Gregory, 2011; Cresswell, McCluskey, & West, 2013) while others have developed methods to learn from exploration traces (Pasula et al., 2007; Mourão et al., 2010; Rodrigues et al., 2010a). In fact all the methods which learn from exploration traces can learn from the more general case where the data consists of sets of independent action observations in the form of *(state,action,state)* tuples. Methods which learn from plan traces cannot learn from independent actions, as these methods typically rely on the fact that the actions are successful and have underlying structure in the form of a plan. These assumptions enable plan-trace methods to learn with limited state observations. Conversely, methods which learn from independent action observations cannot assume such structure and depend on the presence of negative training examples in the form of actions attempted when the preconditions are not satisfied. Our approach is designed to operate in the independent action setting, but is able to learn in both settings by converting plan traces to a set of independent actions and then augmenting the set with likely negative examples. Because the augmentation introduces potentially noisy effects to the training data, this technique can only be applied with approaches which are robust to noisy action effects, namely our approach, and the approaches of Pasula et al. (2007) and Rodrigues et al. (2010a). However, while methods learning from plan traces are often able to learn even when only the initial and goal states are given (e.g. Zhuo & Kambhampati, 2013), the independent-action approaches require intermediate state information to operate.

We extract rules from classifiers based on the intuition that more discriminative features will contribute more to the classifier's objective function. This is similar to the insight underlying feature selection methods of the type which rank each feature according to the sensitivity of the classifier's objective function to the removal of the feature (Kohavi & John, 1997; Guyon, Weston, Barnhill, & Vapnik, 2002). Our approach differs in that features are selected separately for individual examples, rather than once across the entire training set, and we define a stopping criterion which identifies when the set of selected features can no longer form a rule.

Our work also has links with earlier work in version spaces (Mitchell, 1982) and the associated greedy search, which underlie many other approaches to rule learning (e.g. Amir & Chang, 2008; Pasula et al., 2007). Our rule search benefits from extra information to guide the search, in the form of the weights associated with each hypothesis by the previously trained statistical classifiers, which can be highly robust to noise and incomplete observations.

8.2 Conclusions

Currently our approach relies on the availability of predefined symbolic predicates and actions, however, recently methods have been developed to allow robots to automatically learn their own symbolic representations of the world (Konidaris, Kaelbling, & Lozano-Perez, 2014; Jetchev, Lang, & Toussaint, 2013). A significant future step will therefore be to combine our approach with methods for learning symbolic predicates and actions from continuous-valued robot observations. Our classifier-based domain learning approach presents an opportunity to directly incorporate ideas from work on learning classifier-based symbolic representations.

Another important research direction is to align our method with the capabilities and requirements of modern planners. One aspect of this is to extend our approach to operate in more expressive domains such as those with durative actions, action costs or exogenous events none of which are addressed by existing action learning methods. A second aspect to consider is the refinement of existing domain models in order to improve planner performance, either by fixing incorrect or incomplete models, or by introducing learnt macro-actions.

9. Acknowledgements

This work was partially funded by the European Commission through the EU Cognitive Systems project Xperience (FP7-ICT-270273) and the UK EPSRC/MRC through the Neuroinformatics and Computational Neuroscience Doctoral Training Centre, University of Edinburgh.

References

Agre, P. E., & Chapman, D. (1987). Pengi: An implementation of a theory of activity. In *Proceedings of the 6th National Conference on Artificial Intelligence (AAAI 1987)*, pp. 268–272.

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

Ballard, D. H., Hayhoe, M. M., Pook, P. K., & Rao, R. P. (1997). Deictic codes for the embodiment of cognition. *Behavioral and Brain Sciences*, *20*(4), 723–742.

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

Brooks, R. A. (1990). Elephants don't play chess. *Robotics and Autonomous Systems*, *6*(1&2), 3–15.

Cresswell, S., McCluskey, T., & West, M. (2013). Acquiring planning domains using LOCM. *Knowledge Engineering Review*, *28*, 195–213.

Cresswell, S., & Gregory, P. (2011). Generalised domain model acquisition from action traces. In *Proceedings of the 21st International Conference on Automated Planning and Scheduling (ICAPS 2011)*, pp. 42–49.

Finegan, E. (1998). *Language: Its Structure and Use* (Third edition). Heinle & Heinle Publishers.

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

Gärtner, T., Flach, P., & Wrobel, S. (2003). On graph kernels: Hardness results and efficient alternatives. In *Proceedings of the 16th Annual Conference on Computational Learning Theory and 7th Kernel Workshop (COLT 2003)*, pp. 129–143, Berlin, Heidelberg.

Ghallab, M., Nau, D., & Traverso, P. (2004). *Automated Planning: Theory & Practice (The Morgan Kaufmann Series in Artificial Intelligence)* (1st edition). Morgan Kaufmann.

Goodale, M. A., & Milner, A. D. (1992). Separate visual pathways for perception and action.. *Trends in Neurosciences*, *15*(1), 20–25.

Guyon, I., Weston, J., Barnhill, S., & Vapnik, V. (2002). Gene selection for cancer classification using support vector machines. *Machine Learning*, *46*(1-3), 389–422.

Halbritter, F., & Geibel, P. (2007). Learning models of relational MDPs using graph kernels. In *Proceedings of the 6th Mexican International Conference on Advances in Artificial Intelligence (MICAI 2007)*, pp. 409–419, Berlin, Heidelberg. Springer-Verlag.

Haussler, D. (1989). Learning conjunctive concepts in structural domains. *Machine Learning*, *4*(1), 7–40.

Hommel, B. (2004). Event files: Feature binding in and across perception and action. *Trends in Cognitive Sciences*, *8*(11), 494–500.

Hurford, J. R. (2003). The neural basis of predicate-argument structure. *Behavioral and Brain Sciences*, *23*(6), 261–283.

Itti, L., Koch, C., & Niebur, E. (1998). A model of saliency-based visual attention for rapid scene analysis. *IEEE Transactions on Pattern Analysis and Machine Intelligence*, *20*(11), 1254–1259.

Jetchev, N., Lang, T., & Toussaint, M. (2013). Learning grounded relational symbols from continuous data for abstract reasoning. In *Proceedings of the ICRA Workshop on Autonomous Learning, ICRA 2013*.

Kahneman, D., Treisman, A., & Gibbs, B. J. (1992). The reviewing of object files: Object-specific integration of information.. *Cognitive Psychology*, *24*(2), 175–219.

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

Khardon, R., & Servedio, R. A. (2005). Maximum margin algorithms with Boolean kernels. *Journal of Machine Learning Research*, *6*, 1405–1429.

Koch, C., & Ullman, S. (1985). Shifts in selective visual attention: Towards the underlying neural circuitry.. *Human Neurobiology*, *4*(4), 219–227.

Kohavi, R., & John, G. H. (1997). Wrappers for feature subset selection. *Artificial Intelligence*, *97*(1-2), 273–324.

Konidaris, G., Kaelbling, L. P., & Lozano-Perez, T. (2014). Constructing symbolic representations for high-level planning. In *The Twenty-Eighth International Conference on Artificial Intelligence*, p. to appear.

Krüger, N., Geib, C., Piater, J., Petrick, R., Steedman, M., Wörgötter, F., Ude, A., Asfour, T., Kraft, D., Omrčen, D., Agostini, A., & Dillmann, R. (2011). Object-Action Complexes: Grounded abstractions of sensorimotor processes. *Robotics and Autonomous Systems*, *59*(10), 740–757.

Lang, T., & Toussaint, M. (2010). Planning with noisy probabilistic relational rules. *Journal of Artificial Intelligence Research*, *39*(1), 1–49.

McCluskey, T. L., Cresswell, S. N., Richardson, N. E., & West, M. M. (2009). Automated acquisition of action knowledge. In *Proceedings of the International Conference on Agents and Artificial Intelligence (ICAART)*, pp. 93–100.

McCluskey, T., Richardson, N., & Simpson, R. (2002). An interactive method for inducing operator descriptions. In *Proceedings of the Sixth International Conference on Artificial Intelligence Planning Systems*, AIPS 2002, pp. 121–130.

Mitchell, T. (1982). Generalization as search. *Artificial Intelligence*, *18*(2), 203–226.

Mourão, K. (2013). Learning planning models from incomplete observations. In *Proceedings of the Planning and Learning Workshop at ICAPS*, ICAPS 2013.

Mourão, K., Petrick, R. P. A., & Steedman, M. (2010). Learning action effects in partially observable domains. In *Proceedings of the 19th European Conference on Artificial Intelligence (ECAI 2010)*, pp. 973–974. IOS Press.

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

Nienhuys-Cheng, S.-H., & de Wolf, R. (1997). *Foundations of Inductive Logic Programming*. Springer, Berlin, Heidelberg.

O'Regan, J. K. (1992). Solving the "real" mysteries of visual perception: The world as an outside memory.. *Canadian Journal of Psychology*, *46*(3), 461–488.

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

Pylyshyn, Z. (1989). The role of location indexes in spatial perception: A sketch of the FINST spatial-index model.. *Cognition*, *32*(1), 65–97.

Pylyshyn, Z. W. (2000). Situating vision in the world. *Trends in Cognitive Sciences*, *4*(5), 197–207.

Pylyshyn, Z. W. (2001). Visual indexes, preconceptual objects, and situated vision. *Cognition*, *80*(1-2), 127–158.

Pylyshyn, Z. W. (2009). Perception, representation and the world: The FINST that binds. In Dedrick, D., & Trick, L. M. (Eds.), *Computation, Cognition and Pylyshyn*, pp. 3–48. MIT Press.

Rodrigues, C., Gérard, P., & Rouveirol, C. (2010a). Incremental learning of relational action models in noisy environments. In *Proceedings of the International Conference on Inductive Logic Programming (ILP 2010)*, pp. 206–213.

Rodrigues, C., Gérard, P., Rouveirol, C., & Soldano, H. (2010b). Incremental learning of relational action rules. In *International Conference on Machine Learning and Applications (ICMLA 2010)*, pp. 451–458, Los Alamitos, CA, USA. IEEE Computer Society.

Sadohara, K. (2001). Learning of Boolean functions using support vector machines. In *Proceedings of the 12th International Conference on Algorithmic Learning Theory (ALT 2001)*, pp. 106–118. Springer.

Shervashidze, N., Schweitzer, P., van Leeuwen, E. J., Mehlhorn, K., & Borgwardt, K. M. (2011). Weisfeiler-Lehman graph kernels. *Journal of Machine Learning Research*, *12*, 2539–2561.

Van Rijsbergen, C. J. (1979). *Information Retrieval* (2nd edition). Butterworth-Heinemann.

Walsh, T. J., & Littman, M. L. (2008). Efficient learning of action schemas and web-service descriptions. In *Proceedings of the 23rd National Conference on Artificial Intelligence (AAAI 2008)*, pp. 714–719.

Whitehead, S. D., & Ballard, D. H. (1991). Learning to perceive and act by trial and error. *Machine Learning*, *7*(1), 45–83.

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

Zhuo, H. H., & Kambhampati, S. (2013). Action model acquisition from noisy plan traces. In *Proceedings of the Twenty-Third International Joint Conference on Artificial Intelligence*, IJCAI'13, pp. 2444–2450. AAAI Press.

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

AN APPLICATION PROGRAMMING INTERFACE TO HIGH-LEVEL PLANNING WITH PKS

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

2015-01-15

Technical Report

Abstract

The task of integrating planners on robot platforms often centres around the problem of representation, and how to abstract the capabilities of a robot and its working environment into a suitable form for use by the planner. Integration also typically requires the ability to communicate information between system components and, thus, requires a consideration of certain engineering-level concerns, to ensure proper interoperability with components that aren't traditionally considered in theoretical planning settings. This report presents a snapshot of UEDIN's contribution to the work of integrating high-level planning on robot platforms in the Xperience project. In particular, this document focuses on the current state of an application programming interface (API) designed to provide an abstract, general-purpose 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 meant to be generic and any planner which supports the API can be used in its place as an alternative backend. This document presents details of the latest version of the interface (Version 0.85, 2014-10-09). An example planning domain is also described, to demonstrate certain features of the representation and API.

Revision history

2015-01-15 :	A completely rewritten and extended version of the planning report, building on the previous version, but reflecting changes to the API since the last report was released. Notable differences in this report include a completely rewritten description of the API and the inclusion of an example planning domain. The document has also been restructured to include new motivation, implementation, and discussion sections. This version of the report was included as part of the Month 49 deliverable D3.2.4 (Transfer of Structural Bootstrapping for Planning).
2013-12-18 :	An initial report presenting a snapshot of the 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). This version of the report was included as part of the Month 37 deliverable D3.2.3 (Structural Bootstrapping for Planning: Extended Reasoning and Indexical Information).
2012-01-25 :	A report describing the role of the PKS planner and its derivative technologies which will be used and extended throughout the project. This version of the report was included as part of the Month 13 deliverable D3.2.1 (Structural Bootstrapping for Planning: Informed Search for Knowledge-Level Planning).

Contents

1 Introduction

In this document we focus on the state of UEDIN's high-level automated planning work 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 3.2.2 (Learning knowledge-level control rules), 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)* framework 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 domains as part of WP3 (with some connections to WP4). 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 on a robot platform.

2 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.

In this paper, we explore the application of planning with incomplete information and sensing actions to the problem of planning in robotics domains. In particular, building models of realistic domains which can be used with general-purpose planning systems often involves working with incomplete (or uncertain) perceptual information arising from real-world sensors. Furthermore, this task may be complicated by the difficulties of bridging the gap between geometric and symbolic representations: robot systems typically reason about joint angles, spatial coordinates, and continuous spaces, while many symbolic planners work with discrete representations in represented in logic-like languages (Geib et al., 2006; Krüger et al., 2011).

Our approach uses the PKS (Planning with Knowledge and Sensing) planner (Petrick and Bacchus, 2002, 2004) as the high-level reasoning tool for task planning in robotics domains. PKS is a general-purpose contingent planner that operates at the *knowledge level* (Newell, 1982), by reasoning about how its knowledge changes due to action. PKS can represent known and unknown information, and model sensing actions using concise but rich domain descriptions, making it well suited for structured, partially-known environments of the kind that arise in many robot scenarios, such as those we consider on the Xperience project.

Like most high-level 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.

While PKS has previously been used successfully in robot domains (Petrick et al., 2009), the basic planning system lacks some of the features mentioned above which could improve its applicability to a wider range of robotics tasks. The most notable of these is an application programming interface (API) that abstracts the planner's internal structures and functions into a form that enables it to be more easily integrated as a component in a larger system. In this paper, we describe such an interface for PKS using the *Internet Communications Engine (ICE)*[1] which provides the network-level middleware.

3 Planning with Knowledge and Sensing (PKS)

PKS (Planning with Knowledge and Sensing) is a contingent planner that builds plans using 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 changes due to action. This differs from some contingent planners that attempt to instead model how the world state changes when actions are performed. PKS works with a restricted first-order logical language, and limited inference. Thus, unlike planners that reason with possible worlds or belief states, PKS works directly with formulae representing its knowledge state. This representation supports features like functions and variables that arise in real-world planning scenarios; however, as a trade-off, some types of knowledge cannot be modelled.

PKS is based on a generalisation of STRIPS (Fikes and Nilsson, 1971). 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. Each database's contents have a formal interpretation in a modal logic of knowledge. Actions can modify the databases, which has the effect of updating the planner's knowledge. To ensure efficient inference, PKS restricts the type of knowledge (especially disjunctions) the databases can represent:

$\mathbf{K_f}$: This database is like a standard STRIPS database except that both positive and negative facts are stored and the closed world assumption (Reiter, 1978) is not applied. K_f is primarily 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. For instance, `objAt(glass1,sideboard)` $\in K_f$ could be used to represent the fact that the planner knows that the object `glass1` is on the sideboard.

$\mathbf{K_w}$: This database models the plan-time effects of sensing actions with binary outcomes. $\phi \in K_w$ 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 information is important since PKS can make use of such knowledge to construct conditional plans (i.e., plans with context-dependent branches). For instance, `objAt(plate2,sideboard)` $\in K_w$ could be used to represent the fact that the

[1] `http://www.zeroc.com/ice.html`

planner knows at plan time that it will come to know whether `objAt(plate2,sideboard)` is actually true or false at run time.

$\mathbf{K_v}$: This database stores function values that will become known at execution, such as the effects of sensing actions that return constants. 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. Thus, K_v terms can act as *run-time variables* (Etzioni et al., 1992) in plans. For instance, `colour(cup3)` $\in K_v$ could be used to represent the fact that, at plan time, the planner knows the colour of `cup3` and could use the term `colour(cup3)` as an alternative reference to the actual colour. At run time, the planner will learn the actual denotation of `colour(cup3)`, e.g., `colour(cup3) = blue`.

$\mathbf{K_x}$: This database models the planner's *exclusive-or* knowledge. 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 arises in many planning scenarios, namely that "exactly one of the ℓ_i is true." For instance, `objAt(glass4,sideboard) | objAt(glass4,cupboard)` $\in K_x$ could be used to represent the fact that `glass4` is either on the sideboard or in the cupboard, but not both. Unlike K_w and K_v knowledge, there is no guarantee that K_x will be resolved at run time and subsequent sensing might be required to distinguish between the alternatives.

$\mathbf{LCW}$: This database stores *local closed world* information (Etzioni et al., 1994), i.e., instances where the planner has complete information about the state of the world. Normally the planner's lack of knowledge about a fact means that it will conclude that it is "unknown". With LCW, the planner can conclude that the state of certain facts is "false" in the absence of other information to the contrary. For instance, `objAt(?x,sideboard)` $\in LCW$ can be used to denote the idea that the planner has local closed world information about all objects on the sideboard. If the planner is then faced with a query about a particular object, e.g., is `objAt(bowl5,sideboard)` true or false, failure to establish that the fact is true means the planner can conclude that `objAt(bowl5,sideboard)` is false (instead of unknown).

PKS databases are inspected using *primitive queries* that ask simple questions about PKS's knowledge. Basic knowledge assertions are tested with a query $K(\phi)$ which asks: "is a formula ϕ true?" A query $K_w(\phi)$ asks whether ϕ is known to be true or known to be false. A query $K_v(t)$ asks "is the value of function t known?" The negation of the above queries can also be used. An inference procedure evaluates queries by checking the contents of the databases, and the interaction between different types of knowledge.

An action in PKS is modelled by its *preconditions* that query the agent's knowledge state, and its *effects* that update the state. Action preconditions are simply a list of primitive queries. Action effects are described by a set of STRIPS-style "add" and "delete" operations that modify 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 conditional effects (Pednault, 1989), where the secondary preconditions of an effect are described by lists of primitive queries. Simple forms of quantification, $\forall^K x$ and $\exists^K x$, that range over known instantiations of x can be used, along with certain types of simple program-like constructs (e.g., while loops, if-else structures, and temporary variables).

PKS constructs plans by forward-chaining: if an action's preconditions are satisfied in a knowledge state, then the action's effects can be applied to produce a new knowledge state. Planning then continues from this state. PKS can also build contingent plans with branches, by considering the possible outcomes of K_w and K_v knowledge. For instance, if $\phi \in K_w$ then PKS can construct two branches in a plan: along one branch (the K^+ branch) ϕ is assumed to be known (ϕ is added to K_f), while along the other branch (the K^- branch), $\neg\phi$ is assumed to be known ($\neg\phi$ is added to K_f). A similar type of multi-way branching plan can be built for restricted K_v information. Planning continues along each branch until the *goal*, a list of primitive queries, is satisfied.

Example PKS actions and plans are shown in Section 6.

4 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.

The task of integrating planners on robot platforms often centres around the problem of representation, and how to abstract the capabilities of a robot and its working environment into a suitable form for use by the planner. Integration also typically requires the ability to communicate information between system components and, thus, requires a consideration of certain engineering-level concerns, to ensure proper interoperability with components that aren't traditionally considered in theoretical planning settings.

To facilitate the task of providing software-level planning services to complex systems (e.g., robot platforms), we created an *application programming interface (API)* which abstracts many common planning operations into a set of functions which provide direct, programme-level access to these services. For instance, the API includes methods for manipulating domain representations and controlling certain aspects of the plan generation process (e.g., selecting goals, generation strategies, or planner-specific settings). Plans can also be manipulated as first-class entities, e.g., for replanning purposes. 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.[2]

Overall, the API is designed to be generic and is not tied to a particular planner. For instance, the configuration methods are meant to provide a way to set properties of the underlying planner, and provide access to features needed for debugging. The domain configuration functions provide the means for defining planning domain models, either from traditional domain/problem files, or via string-based descriptions. A key idea behind the API is that it offers the possibility of specifying domains to the planner incrementally, using function calls alone, rather than specifying a single monolithic domain file. This means that an initial domain could be specified and then later revised, for instance using information discovered during execution (e.g., new objects, revised action models, additional properties, etc.). Finally, the plan generation and iteration functions provide ways of controlling certain aspects of plan generation, and provide a way for external processes to control monitoring and replanning activities, including goal change.

The PKS API is implemented as a C++ library, but also supports a network-based interface using the Internet Communications Engine (ICE). (Alternative interfaces being developed are discussed below in Section 7.) We discuss the main components of the interface below.

4.1 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),
```

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

```
// Abstract state property description
struct StateProperty {
    string     name;
    StringList args;
    bool       sign;
    string     value;
};

// Abstract state description
sequence<StateProperty> StatePropertyList;

// Abstract plan step description
struct PlanStep {
    string     name;
    string     type;
    StringList args;
};

// Abstract linear plan sequence
sequence<PlanStep> PlanStepList;

// Abstract state change description
struct StateChange {
    StatePropertyList addList;
    StatePropertyList deleteList;
};

// Main planning functions
interface PlannerController {
    // Planner configuration and debugging
    void              reset();
    string            getPlannerProperty(string s);
    bool              setPlannerProperty(string s, string t);

    // Domain configuration
    void              clearDomain();
    void              clearDomainSymbols();
    void              clearDomainActions();
    void              clearDomainProblems();
    void              clearDomainInitialState();

    bool              loadDomain(string s);
    bool              loadDomainSymbols(string s);
    bool              loadDomainActions(string s);
    bool              loadDomainProblems(string s);
    bool              loadDomainInitialState(string s);
    bool              loadDomainInitialStateFacts(string s);

    bool              defineDomain(string s);
    bool              defineDomainSymbols(string s);
    bool              defineDomainActions(string s);
    bool              defineDomainProblems(string s);
    bool              defineDomainInitialState(string s);
    bool              defineDomainInitialStateFacts(string s);
    bool              defineDomainInitialStateFactsFromList(StatePropertyList s);

    // Plan generation and plan iteration
    void              clearPlan();
    void              resetPlan();
    bool              buildPlan();
    PlanStepList      getPlan();
    StatePropertyList getPlanState();
    PlanStep          getPlanAction();
    bool              isPlanDefined();
    bool              isEndOfPlan();
    void              advancePlan();
    bool              setPlanProblem(string s);

    // Plan inspection
    bool              checkPlanActionPreconditions(StatePropertyList s);
    StateChange       getPlanActionEffects(StatePropertyList s);
    StatePropertyList getPlanActionEffectsState(StatePropertyList s);
    StatePropertyList getPlanInitialState();
    StatePropertyList getDomainInitialState();
    StatePropertyList getDomainProblemInitialState();
    StateChange       getStateDifference(StatePropertyList s, StatePropertyList t);

    // Conversion functions
    StringList        statePropertyListToString(StatePropertyList s);
    string            statePropertyToString(StateProperty s);
    StringList        planStepListToString(PlanStepList p);
    string            planStepToString(PlanStep p);
};
```

Figure 1: API for high-level planning services (Version 0.85, 2014-10-09)

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 a list of `StateProperty` definitions, denoted in the API as a `StatePropertyList`. 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 should also be noted that since this structure is simply a container, it does not force any particular interpretation of what the contents actually represent with respect to various types of state encodings (i.e., open world properties versus positive literals combined with a closed world assumption). Thus, this structure can be used to embed complete planned states or partial observed states as returned by a set of sensors.

4.2 Plan steps and plan sequences

A `PlanStep` can be thought of as an 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 `grasp(left,sideboard,glass1)` (grasp `glass1` from the `sideboard` using the robot's `left` hand), which denotes a type of manipulation action, could be encoded in this structure as:

```
name     :  grasp
type     :  manipulation
args[0]  :  left
args[1]  :  sideboard
args[2]  :  glass1.
```

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.

A `PlanStepList` is simply a sequence of `PlanSteps` (i.e., instantiated actions) which can also be thought of as a simple linear plan, of the form that most classical and conformant 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 any 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++). More details about this future extension are discussed in Section 7.

4.3 State changes

A `StateChange` structure is used to denote the difference between two states (for this discussion, S_1 and S_2) as represented by two sets of properties: an `addList` denoting the properties that occur in S_2 but not S_1, and a `deleteList` denoting the properties that occur in S_1 but not S_2. The `addList` and `deleteList` sets are themselves simply instances of `StatePropertyList`, i.e., state containers. This notion of state change captures an important inertial frame property underlying STRIPS: properties that are unchanged between states are not represented in the `StateChange` structure.

For instance, if

$$S_1 = \{\texttt{objAt(glass1,sideboard)}, \texttt{objAt(plate2,sideboard)}, \texttt{handEmpty(left)}\}$$

and

$$S_2 = \{\neg\texttt{objAt(glass1,sideboard)}, \texttt{objAt(plate2,sideboard)}, \texttt{inHand(glass1,left)}\}$$

are two states (assumed to be encoded in proper StatePropertyList structures), then the difference between S_1 and S_2 could be captured by a StateChange structure where

$$\texttt{addList} = \{\neg\texttt{objAt(glass1,sideboard)}, \texttt{inHand(glass1,left)}\}$$
$$\texttt{deleteList} = \{\texttt{objAt(glass1,sideboard)}, \texttt{handEmpty(left)}\}$$

(again, encoded in proper StatePropertyList structure).

4.4 Planner configuration and debugging

The main set of planning functions are divided into various categories. 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 default settings.
- `getPlannerProperty(string s)`: This function returns the status of the planner property variable `s`. The current set of supported properties for PKS includes:
  - `SearchMethod`: the underlying search strategy used by the planner.
  - `SearchDepth`: the maximum depth the planner should search for a plan.
  - `UseFunctionsAsActionArgs`: allow known function terms to be bound as action parameters.
  - `UseDomainProblemInitialState`: use the initial state defined in a `problem` block.
  - `UseDomainInitialState`: use the initial state definition specified by a prior call to the `loadDomainInitialState` or `defineDomainInitialState` functions.

  The precise set of accessible properties is defined by the underlying planner.
- `setPlannerProperty(string s, string t)`: This function sets the status of planner property `s` to value `t`. (The current set of supported properties for PKS is given above in `getPlannerProperty`.) The precise set of accessible properties and associated values is defined by the underlying planner.

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.

4.5 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()`, `clearDomainSymbols()`, `clearDomainActions()`, `clearDomainProblems()`, `clearDomainInitialState()`: These functions direct the planner to delete any domain (similarly, symbols, actions, problems, or initial states) that are currently defined.
- `loadDomain(string s)`: This function directs the planner to load a domain from the specified local file `s`. The actual format of the domain is specified by the backend planner.
- `loadDomainSymbols(string s)`: This function directs the planner to load a set of symbol definitions from the specified local file `s`. Symbol definitions typically involve a specification of the allowable objects, types, and properties in a planning domain.

- loadDomainActions(string s): This function directs the planner to load a set of action definitions from the specified local file s. Actions are defined in a language supported by the backend planner.
- loadDomainProblems(string s): This function directs the planner to load a set of problem definitions from the specified local file 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.
- loadDomainInitialState(string s): This function allows the planner to load an initial state definition from the specified local file s. Such a state can be used by the planner as an alternative initial state to the state defined in a problem definition, 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. The function assumes that the state is specified in the standard format used by the backend planner.
- loadDomainInitialStateFacts(string s): This function allows the planner to load an initial state definition from from the specified local file s, where the state is defined as a comma-separated list of properties. Otherwise, this function is identical to loadDomainInitialState.
- defineDomain(string s), ..., defineDomainInitialStateFacts(string s): These functions are analogous to the functions loadDomain(string s), ..., loadDomainInitialStateFacts(string s), as defined above, except that rather than loading definitions from a specified local file, 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. With the exception of loadDomainInitialStateFacts, which assumes a comma-separated list of state properties, the precise format of the string s depends on the backend planner.
- defineDomainInitialStateFactsFromList(StatePropertyList s): This function is similar to its defineDomainInitialState counterpart except that rather than specifying a state definition in a string s, it is defined using the abstract 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 produced by 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 direct solution to this problem.

4.6 Plan generation and plan iteration

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

- clearPlan(): This function directs the planner to clear any existing plan it has saved, if one exists.
- resetPlan(): This function resets the state of an existing plan to the first step.

- `buildPlan()`: This function directs the planner to generate a plan using the current settings, domain, and the current or default planning problem. The return value of the function (a `bool`) specifies whether a plan was found or not.
- `getPlan()`: This function directs the planner to return the current plan (if one exists) as a `PlanStepList` structure. If no plan exists, an empty `PlanStepList` structure is returned.
- `getPlanState()`: This function returns the current expected state of the plan as a `StatePropertyList` structure, which may be empty.
- `getPlanAction()`: This function directs the planner to return the next action in a plan as a `PlanStep` structure.
- `isPlanDefined()`: This function returns a `bool` specifying whether or not a plan is currently defined in the planner.
- `isEndOfPlan()`: This function returns a `bool` specifying whether ot not the end of the plan has been reached.
- `advancePlan()`: This function directs the planner to advance the plan to the next step, if one exists.
- `setPlanProblem(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.

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, so that simple plan execution monitoring activities can be supported without reliance on the backend planner. As a result, a client using these services can control when a plan should be generated, and can iteratively ask for individual plan steps, advancing the plan one step at a time. 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.

4.7 Plan inspection

This set of functions provides access to certain state-related aspects of a generated plan.

- `checkPlanActionPreconditions(StatePropertyList s)`: This function checks if the preconditions of the current action in the plan are satisfied in the specified state `s` (a `StatePropertyList` structure). A `bool` is returned indicating the success or failure of the function.
- `getPlanActionEffects(StatePropertyList s)`: This function returns the effects of the current action in the plan when applied to state `s`. The effects are returned as a `StateChange` structure.
- `getPlanActionEffectsState(StatePropertyList s)`: This function is similar to the function `getPlanActionEffects` except that the resulting state structure is returned as a `StatePropertyList`.
- `getPlanInitialState()`: This function returns the initial state of the current plan as a `StatePropertyList` structure.
- `getDomainInitialState()`: This function returns the initial state previously defined by a `loadDomainInitialState` or `defineDomainInitialState` function. The state is returned as a `StatePropertyList` structure.
- `getDomainProblemInitialState()`: This function returns the initial state defined by the currently selected `problem`. The state is returned as a `StatePropertyList`.

- `getStateDifference(StatePropertyList s, StatePropertyList t)`: This function returns the difference between state `s` and state `t` (specified as `StatePropertyList` structures). The state difference is returned as a `StateChange` structure.

These functions allow an external client access to the state-level effects of a plan and provide support for producing the state changes that result from applying planned actions to alternative states. As a result, these functions are potentially useful for constructing an external plan execution monitor.

4.8 Conversion functions

The final set of functions provides a set of helper methods for converting abstract properties, states, and plans to strings, for instance for display purposes.

- `statePropertyListToString(StatePropertyList s)`: This function converts a state specified as a `StatePropertyList` to a string.
- `statePropertyToString(StateProperty s)`: This function converts a `StateProperty` structure to a string.
- `planStepListToString(PlanStepList p)`: This function converts a plan specified as a `PlanStepList` structure to a string.
- `planStepToString(PlanStep p)`: This function converts a plan step specified as a `PlanStep` structure to a string.

5 Implementation

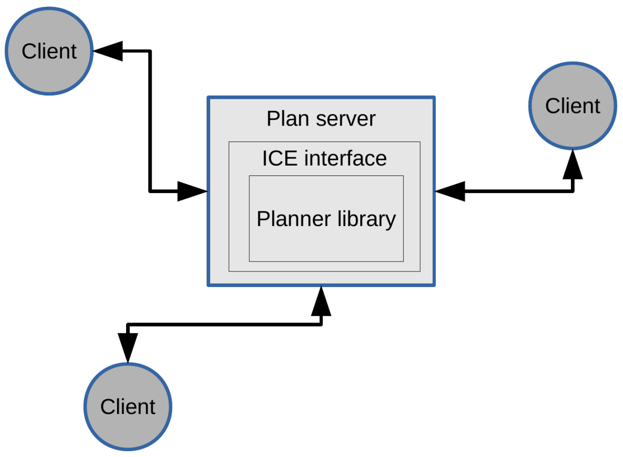


Figure 2: A client-server model for accessing planner services via ICE

This document is not meant to provide precise details concerning the implementation of the API, however, we note here the decisions which has guided our current implementation.

Client-server structure: The general structure of the current API implementation is a client-server model as shown in Figure 2. In particular, planner services are supplied by a plan server which could be accessed by possibly multiple clients via the Internet Communications Engine (ICE) which provides the necessary network-level middleware. In its minimal form, the plan server is structured as an ICE interface wrapped around a backend planning library which provides the necessary planning services. In our current implementation, the ICE interface is a “thin” layer which simply passes function calls to an instance of the PKS planner implemented as a C++ library which appropriately responds to each client request.

Internet Communication Engine (ICE) interface: One important design feature to note is that, in keeping with standard ICE behaviour, the plan server's ICE interface acts as the external-facing interface to the planning library's functions. As a result, it simply reflects the domain independent nature of the underlying planner and is not tied to a particular deployment environment (e.g., a robot platform). Instead, all application specific code (e.g., connecting it to a robot) by design exists outwith the plan server, either at the client level or elsewhere, with the client providing the necessary bridge between the plan server and the deployment environment. On the client side, ICE provides an object-oriented interface whereby the plan server functions can be executed as remote procedure calls using a locally-defined object which communicates to the server through the ICE middleware.

Support for multiple backends: Another design feature incorporated into the structure of the API is that it is meant to provide future support for multiple planning backends. While the current API implementation was adapted from the underlying interface to the PKS planner, it has been abstracted to avoid PKS-specific representations and syntax, at least at the interface function level. (I.e., 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. Section 7 outlines current work aimed at adapting this interface to off-the-shelf PDDL planners via a conversion library.

Backend-dependent syntax: One part of the current implementation that is dependent on the specific underlying planner is the syntax used by the various domain definition functions (e.g., defining symbols, actions, problems, etc.). Since the plan server simply passes function calls from the ICE layer to the backend planner, the precise syntactic form is currently left to that planner. As a result, this means that the content of certain function calls may need to change if the backend is changed. However, this also means that all planning domain entities do not need to be standardised in order to support the API. Indeed, it is unlikely this standardisation will occur in the future, even with the adoption of a language like PDDL (McDermott et al., 1998) which itself supports certain variants and diverse sets of features. The difficulties of producing a standard language suitable for all planners is well known and planners like PKS continue to support a set of features at the representation level that are not present in many planners. In addition, the introduction of new languages like RDDL (Sanner, 2011) has provided even more choice in recent years, providing even more evidence that the question of finding a standardised planning language hasn't been completely answered.

The role of plan execution monitoring: The set of functions provided by the planning API goes beyond traditional operations for simply controlling the planner itself. Instead, it provides access to the actions and states that make up a plan, allowing them to be treated as first class entities and examined by external clients. Moreover, the set of supported functions provides enough primitive operations to build a simple plan execution monitor which can make decisions concerning how a plan is executed and whether replanning operations are deemed to be necessary.

A redeployable domain-independent planning component: We conclude this section by restating our overarching goal behind the API: the design of a set of abstract planning functions supported by a domain-independent backend planning engine. As a result, many of the design decisions, including the choice of a client-server model combined with the ICE interface are meant to provide the necessary separation of the planning system from the deployment environment, in order to produce a planning component that can be installed and used in a variety of contexts. To this end, the planning API will continue to be maintained as a distinct project, with domain-dependent code separated from the main API.

6 Example planning domain: table setting

In this section we consider a simple example of a PKS planning domain and how it can be used to generate plans using the API. This example is adapted (and simplified) from one of the Xperience demonstration domains from WP5, featuring table setting with a robot.

6.1 Domain description

```
domain table-setting {
    symbols {
        types:
            robotHand, location, placesetting, obj,
            fork, knife, plate, glass;

        predicates:
            handEmpty/1,
            robotAt/1,
            objAt/2,
            inHand/2;

        constants:
            robotHand left, right;
            location table_ps1, table_ps2, sideboard;
            placesetting table_ps1, table_ps2;

            obj fork1, fork2, fork3, knife1, knife2, knife3,
                plate1, plate2, plate3, glass1, glass2, glass3;

            fork  fork1, fork2, fork3;
            knife knife1, knife2, knife;
            plate plate1, plate2, plate3;
            glass glass1, glass2, glass3;
    }

    action grasp(?x : robotHand, ?y : location, ?z : obj) {
        preconds:
            K(handEmpty(?x)) &
            K(robotAt(?y)) &
            K(objAt(?z, ?y))

        effects:
            add(Kf, inHand(?z, ?x)),
            del(Kf, handEmpty(?x)),
            del(Kf, objAt(?z, ?y))
    }

    action putdown(?x : robotHand, ?y : location, ?z : obj) {
        preconds:
            K(inHand(?z, ?x)) &
            K(robotAt(?y))

        effects:
            add(Kf, handEmpty(?x)),
            add(Kf, objAt(?z, ?y)),
            del(Kf, inHand(?z, ?x))
    }
    action move(?x : location, ?y : location) {
        preconds:
            K(robotAt(?x))

        effects:
            del(Kf, robotAt(?x)),
            add(Kf, robotAt(?y))
    }

    problem table-setting-example {
        initdb {
            Kf:
                objAt(fork1,  sideboard),
                objAt(fork2,  sideboard),
                objAt(fork3,  sideboard),
                objAt(knife1, sideboard),
                objAt(knife2, sideboard),
                objAt(knife3, sideboard),
                objAt(plate1, sideboard),
                objAt(plate2, sideboard),
                objAt(plate3, sideboard),
                objAt(glass1, sideboard),
                objAt(glass2, sideboard),
                objAt(glass3, sideboard),

                handEmpty(left),
                handEmpty(right),

                robotAt(sideboard)
        }

        goal:
            (forallK(?p : placesetting) (
                (existsK(?x : fork)  K(objAt(?x, ?p))) &
                (existsK(?x : knife) K(objAt(?x, ?p))) &
                (existsK(?x : plate) K(objAt(?x, ?p))) &
                (existsK(?x : glass) K(objAt(?x, ?p)))
            ))
    }
}
```

The above example illustrates a simple domain encoded in PKS's representation language. The domain describes a simple table setting scenario, where the robot has to retrieve particular items from the sideboard (e.g., forks, knifes, plates, and glasses) and place them appropriately on the table at individual place settings.

The first definition block, `symbols`, describes the basic types, properties, and objects that make up the planner's knowledge state description. The `types` block identifies eight different types (e.g., robotHand, location, ...) that can be used to classify objects. The `predicates` block defines the names of the relational properties that are used in the domain description along with their arity (e.g., `handEmpty` takes one argument). The `constants` block specifies lists of objects or labels of particular types (e.g., `left` and `right` are of type `robotHand`). These definitions specify the basic vocabulary that the planner uses for specifying states, actions, and problems.

Each action block provides a description of a single action that the robot can perform. Three actions are defined in the domain: grasp, putdown, and move. Each action is parameterised and subdivided into a preconds block specifying the action's preconditions, and an effects block specifying the action's effects. For instance, the grasp action takes three parameters (?x, ?y, ?z), denoting the behaviour that grasp will pick up an object ?z from location ?y using hand ?x. Each parameter is also defined to have a particular type (e.g., the grasp action's parameter ?x is of type robotHand). The preconditions of grasp capture two properties that must hold before the action can be applied: the robot's hand ?x must be empty (i.e., K(handEmpty(?x)) and object ?z must be at location ?y (i.e., K(objAt(?z,?y))). The action also three effects that specifies the changes it makes to the planner's knowledge state: the object ?z will be in robot hand ?x (i.e., add(Kf,inHand(?z,?x))), robot hand ?x will no longer be empty (i.e., del(Kf,handEmpty(?x))), and object ?z will no longer be at location ?y (i.e., del(Kf,objAt(?z,?y))).

The problem block defines a particular planning problem to be solved, consisting of two main components: initdb specifies the initial knowledge state of the problem, and goal specifies the goal conditions to be achieved by the plan. In the table-setting-example problem, the initdb block defines the initial contents of the K_f database as a list of known facts, e.g., the initial location of the table objects, the state of the robot's hands as empty, and the initial location of the robot. The goal is described using a quantified formula that specifies that a fork, knife, plate, and glass should be located at each placesetting.

6.2 Generating plans

Using the above domain description, a plan can be built to solve the table-setting-example problem. As a first step, the domain description must be communicated to the plan server. Using the API this can be done in one of two main ways. First, if the domain description is contained within a local file, for instance domain.pks, then we can use the following statement in a client:

```
planner->loadDomain("domain.pks");
```

This directs the plan server to attempt to load the domain from the specified file. Alternatively, we could pass the domain description directly to the plan server as a string using the statements:

```
string domain;
// populate domain string with description from Section 6.1
domain = "domain table-setting...";
planner->defineDomain(domain);
```

In both cases, we assume that we have already established communication with the plan server via ICE, with planner denoting the appropriate ICE communications object.

Once the domain has been successfully loaded, we can direct the planner to generate a plan. At a minimum, this can be done as follows:

```
bool success = planner->buildPlan();
```

with the variable success indicating whether or not a plan was successfully built. By default, the first problem in the domain description is used by the planner (in the above case, table-setting-example). On success, the plan can be retrieved using:

```
StatePropertyList plan = planner->getPlan();
```

For instance, one possible solution generated by the planner for the above problem is the linear plan:

```
grasp(left, sideboard, fork1)
grasp(right, sideboard, knife1)
move(sideboard, table_ps1)
```

```
putdown(left, table_ps1, fork1)
putdown(right, table_ps1, knife1)
move(table_ps1, sideboard)
grasp(left, sideboard, fork2)
grasp(right, sideboard, knife2)
move(sideboard, table_ps2)
putdown(left, table_ps2, fork2)
putdown(right, table_ps2, knife1)
move(table_ps2, sideboard)
grasp(left, sideboard, plate1)
grasp(right, sideboard, glass1)
move(sideboard, table_ps1)
putdown(left, table_ps1, plate1)
putdown(right, table_ps1, glass1)
move(table_ps1, sideboard)
grasp(left, sideboard, plate2)
grasp(right, sideboard, glass2)
move(sideboard, table_ps2)
putdown(left, table_ps2, plate2)
putdown(right, table_ps2, glass2).
```

In this plan, the robot is able to carry two items at a time (one in each hand) to a particular place setting. The goal specifies that two places at the table must be set, each involving four objects. With no additional constraints in the problem, the planner is free to choose the order in which objects are selected and placed.[3] Using a call to `getPlan()`, the above plan would be returned as a `PlanStepList` structure with each individual action in the plan encoded as a single `PlanStep`.

We now consider the situation where the domain is changed by adding a new placesetting `table_ps3`, and the planner directed to build a plan for the same goal as before. First, the additional symbol can be specified to the plan server as follows:

```
planner->defineDomainSymbols("constants: placesetting table_ps3;");
```

The planner can then be invoked as before and the plan retrieved:

```
bool success2 = planner->buildPlan();
StatePropertyList plan2 = planner->getPlan();
```

In this case, one possible plan is the linear sequence (returned as a `StatePropertyList`):

```
move(table_ps2, sideboard)
grasp(left, sideboard, fork3)
grasp(right, sideboard, knife3)
move(sideboard, table_ps3)
putdown(left, table_ps3, fork3)
putdown(right, table_ps3, knife3)
move(table_ps3, sideboard)
grasp(left, sideboard, plate3)
grasp(right, sideboard, glass3)
move(sideboard, table_ps3)
putdown(left, table_ps3, plate3)
putdown(right, table_ps3, glass3).
```

In this plan, since two of the place settings already contain the necessary objects, the planner need only build a plan which sets the necessary objects at `table_ps3`. Again, the order in which objects are moved is left up to the planner.

[3] The order in which objects are selected also depends on the underlying search strategy used by the planner.

7 Discussion

Applications of automated planning to robotics have a long history, going back to robots like Shakey (Nilsson, 1984) and Handey (Lozano-Pérez et al., 1989). Recently, the field has made substantial progress, especially in the area of robot task planning (Kaelbling and Lozano-Pérez, 2011), with many approaches proposed including probabilistic AI techniques (Kaelbling and Lozano-Pérez, 2013), closed-world symbolic planning (Cambon et al., 2009; Plaku and Hager, 2010; Dornhege et al., 2009), formal synthesis (Kress-Gazit and Pappas, 2008; Cheng et al., 2012), and sampling-based manipulation planning (Zacharias et al., 2006; Barry, 2013).

This work focuses on a very specific problem regarding the integration of automated planning on a robot platform: the definition and implementation of a planning API. In particular, our planning API can be thought of as a set of abstract planning services which are implemented by an underlying "black box" planner. As in other complex software systems, such an interface removes the need for the programmer to know how such services are actually implemented, but instead allows the designer to build more complex components that simply use these services.

While this report was not meant to provide comprehensive documentation for the planning API, it did present a snapshot of our current implementation and provide some future directions for the work. 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 if extra layers of abstraction are needed to properly integrate a planner on a specific robot platform, it can be done on the client side using the existing interface as the "core" planning services. In other words, the 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.

One important feature of our current approach is that, 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 API to a PDDL-based planner, to show it can work with our approach. As a first step, we plan to implement a simple conversion library that works with ordinary PDDL files, before adapting the interface to the planner directly.

We are also exploring extensions to the API to enable references to probabilistic representations, temporal constraints, and cost-based encodings. Moreover, we are considering extensions 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. Finally, a version of the planning API based on the Robot Operating System (ROS)[4] is also currently under development, with a prototype release expected later this year.

Acknowledgements

This research was supported in part by the European Commission's Seventh Framework Programme under grant no. 270273 (Xperience, `http://xperience.org/`).

[4] `http://www.ros.org/`

References

J. L. Barry. *Manipulation with Diverse Actions*. PhD thesis, Massachusetts Institute of Technology, 2013.

S. Cambon, R. Alami, and F. Gravot. A hybrid approach to intricate motion, manipulation and task planning. *International Journal of Robotics Research*, 28(1):104–126, 2009.

C. Cheng, M. Geisinger, H. Ruess, C. Buckl, and A. Knoll. Game solving for industrial automation and control. In *IEEE Int. Conf. on Robotics and Automation*, pages 4367–4372, 2012.

C. Dornhege, M. Gissler, M. Teschner, and B. Nebel. Integrating symbolic and geometric planning for mobile manipulation. In *IEEE International Workshop on Safety, Security & Rescue Robotics*, pages 1–6, 2009.

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.

O. Etzioni, K. Golden, and D. Weld. Tractable closed world reasoning with updates. In J. Doyle, E. Sandewall, and P. Torasso, editors, *Proceedings of the Fourth International Conference on Principles of Knowledge Representation and Reasoning (KR-94)*, pages 178–189, Bonn, Germany, May 1994. 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.

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

L. P. Kaelbling and T. Lozano-Pérez. Hierarchical task and motion planning in the now. In *IEEE International Conference on Robotics and Automation (ICRA)*, pages 1470–1477, 2011.

L. P. Kaelbling and T. Lozano-Pérez. Integrated task and motion planning in belief space. *International Journal of Robotics Research*, 32(9–10):1194–1227, 2013.

H. Kress-Gazit and G. Pappas. Automatically synthesizing a planning and control subsystem for the darpa urban challenge. In *Automation Science and Engineering, 2008. CASE 2008. IEEE International Conference on*, pages 766–771, 2008.

N. Krüger, C. Geib, J. Piater, R. Petrick, M. Steedman, F. Wörgötter, A. Ude, T. Asfour, D. Kraft, D. Omrčen, A. Agostini, and R. Dillmann. Object-action complexes: Grounded abstractions of sensorimotor processes. *Robotics and Autonomous Systems*, 59(10):740–757, 2011. doi:10.1016/j.robot.2011.05.009.

T. Lozano-Pérez, J. Jones, E. Mazer, and P. O'Donnell. Task-level planning of pick-and-place robot motions. *Computer*, 22(3):21–29, 1989.

D. McDermott, M. Ghallab, A. Howe, C. Knoblock, A. Ram, M. Veloso, D. Weld, and D. Wilkins. PDDL – The Planning Domain Definition Language (Version 1.2). Technical Report CVC TR-98-003/DCS TR-1165, Yale Center for Computational Vision and Control, Oct. 1998.

A. Newell. The Knowledge Level. *Artificial Intelligence*, 18(1):87–127, 1982.

N. Nilsson. Shakey the robot. Technical Report 323, AI Center, SRI International, Apr. 1984.

E. P. D. Pednault. ADL: Exploring the middle ground between STRIPS and the situation calculus. In R. J. Brachman, H. J. Levesque, and R. Reiter, editors, *Proceedings of the First International Conference on Principles of Knowledge Representation and Reasoning (KR-89)*, pages 324–332, San Mateo, CA, 1989. Morgan Kaufmann Publishers.

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

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.

E. Plaku and G. Hager. Sampling-based motion planning with symbolic, geometric, and differential constraints. In *IEEE Int. Conference on Robotics and Automation*, pages 5002–5008, 2010.

R. Reiter. On closed world data bases. In H. Gallaire and J. Minker, editors, *Logic and Data Bases*, pages 55–76. Plenum Press, New York, 1978.

S. Sanner. Relational dynamic influence diagram language (RDDL): Language description, 2011.

F. Zacharias, C. Borst, and G. Hirzinger. Bridging the gap between task planning and path planning. In *IEEE/RSJ International Conference on Intelligent Robots and Systems*, pages 4490–4495, 2006.