Comenius University
Faculty of Mathematics, Physics and Informatics
Department of Applied Informatics

Project of Dissertation

Online Action Learning Techniques for Noisy and Partially Observable Domains

Author: Michal Čertický

Supervisor: doc. PhDr. Ján Šefránek CSc.

Bratislava 2011

Abstract

Knowledge about domain dynamics, describing how certain actions affect the world, is called an *action model*, and constitutes the essential requirement for planning and goal-oriented intelligent behaviour. *Action learning*, as the automatic construction of action models, has become a hot research topic in recent years, and various methods employing wide variety of AI tools have been developed. Diversity of these methods naturally renders each of them usable under different conditions and in various kinds of domains.

After the extensive analysis of related work, we have declared our goal to introduce a collection of *tractable* and *online* methods for *probabilistic* action learning in *noisy* and *partially observable* domains supporting the induction of action's *preconditions* and complex *conditional effects*.

Subsequently, we have proposed the first solution candidate, which embeds our compact representation structure called *effect formula* ($\mathcal{EF}$) and a polynomial algorithm 3SG (*Simultaneous Specification, Simplification, and Generalization*), that produces and improves action models represented by $\mathcal{EF}$.

Keywords: action learning, probabilistic action model, planning, partially-observable domain, noise

Contents

Chapter 1

Introduction

1.1 Background

Knowledge about domain dynamics, describing how certain actions affect the world, is essential for planning and intelligent goal-oriented behaviour of both living and artificial agents. Such knowledge is in artificial systems referred to as *action model*, and is usually manually constructed by domain experts.

Complete specification of action models in case of complex domains is however difficult and time consuming task. In addition to that, it is often needed to modify this action model when confronted with new information. Research on various methods of *automatic construction of action models* has therefore become a hot topic in recent years. This process of constructing and subsequent modification of action models is called *action learning*. Recent action learning methods take various approaches and employ the wide variety of AI tools.

Let us mention the heuristic *greedy search based* action learning introduced

in [Zettelmoyer-Pasula-Kaelbling, 2003], *perceptron algorithm based* method which can be found in [Mourao-Petrick-Steedman, 2010], learning through *inference over logic programs* described in [Balduccini, 2007], or two solutions based on the conversion of action learning into different classes of *satisfiability problem*, available in [Amir-Chang, 2008] and [Yang-Wu-Jiang, 2007]. In chapter 3 of this dissertation project, we provide detailed analysis of these modern methods and their properties.

1.2 Problem

The diversity of action learning methods renders each of them usable under different conditions and in different kinds of domains. Modern methods are most often evaluated and compared based on the following set of properties:

- Usability in *partially observable domains*,
- production of *probabilistic* action models,
- probabilistic estimation of *current world state*,
- usability in *noisy* domains with *action failures*,
- derivation of action's *preconditions and effects*,
- compatibility with *conditional effects*,
- and *tractability*.

Examination of available solutions has shown, that the vast majority of them has only two or three of these properties (see the table in figure 3.1 for com-

prehensive comparison, and the rest of chapter 3 for evaluation of individual methods).

Our goal is the **introduction** and **comparison** of several novel **action learning methods**, which would possess all of the aforementioned properties, and subsequent **analysis** of their **effectiveness**, **compatibility** with current representation formalisms, and **alternative uses**.

1.3 Proposed Solution Candidate

In this work we also propose the first solution candidate which embeds the following two notions:

1. Compact representation structure called the *effect formula* ($\mathcal{EF}$)
2. and the online polynomial algorithm called 3SG (*Simultaneous Specification, Simplification, and Generalization*), that produces and modifies an action model represented by $\mathcal{EF}$.

Currently, we can safely say, that 3SG algorithm has at least five out of seven desired properties: it is designed for learning *probabilistic* action models with *conditional effects* in *noisy*, *partially observable domains*, while keeping the *computational complexity* low. Additional research is still required, in order to decide whether it possesses remaining two properties.

1.4 Outline

After first three sections of chapter 2, which provide extensive description of background and motivation behind the problem of action learning, we will get to section 2.4, which contains detailed explanation of individual attributes.

Chapter 3 then provides the comprehensive analysis of five recent methods, and their evaluation/comparison based on this set of attributes.

Chapter 4 finally stipulates our goals, preceded by necessary theoretical background. After these goals, reader can find the explanation and pseudocode of the first proposed solution candidate - 3SG algorithm.

Chapter 2

Preliminaries

2.1 Planning: Why do we need a domain description?

Over past few decades, the problem of **planning** has become a significant part of artificial intelligence research. Among numerous approaches and methods that were developed and successfully used either in abstract, or real-world domains, we must first mention the pioneer work of Richard Fikes and Nils Nilsson, who established an extremely influential paradigm in early 70's [Fikes-Nilsson, 1971] by introducing a simple action representation formalism called STRIPS. Task of planning have been since then understood as finding a sequence of actions leading an agent from initial world state, to a world state satisfying its goals.

Methods for solving the task of planning are countless and make use of various techniques from other parts of AI. To mention just a few, we can start with graph-search based methods like Partial Order Planing [Sacredoti, 1975,

Tate, 1977, Wilkins, 1984], GraphPlan [Blum-Furst, 1997, Blum-Langford, 2000, Kambhampati, 2000, Lopez-Bacchus, 2003], OBBD-Base Planning [Jensen, 1999, Jensen-Veloso, 2000, Jensen, 2003], logic-based methods like SATPlan [Kautz-Selman, 1992, Baioletti-Marcugini-Milani, 1998, Rintanen, 2009] or Logic Programming [Subrahmanian-Zaniolo, 1995, Eiter et al., 2000], hierarchical network based planning like HTN [Kutluhan-Hendler-Nau, 1994], probabilistic planning with Markov Decision Processes (MDPs and POMDPs) [McMahan-Gordon, 2005, Theocharous-Kaelbling, 2003], or Genetic Planning [Farritor-Dubowsky, 2002]. Planning itself however, is not the central topic of this text and we will not explain these methods in detail. We merely recapitulate them in order to point out one thing they all have in common: they all need some kind of **domain dynamics description - action model**.

2.2 Representation Languages: How to express our action model?

Action model is simply a description of dynamics within our planning domain, and can be understood as an expression of all the **actions** that can be executed in it, among with their **preconditions** and **effects** in some kind of representation language. Given an **initial** and **goal state**, the planning is simply an algorithmic process exploiting given action model expressed by some kind of representation language to find suitable plans.

Just like there are many planning algorithms, there is a large number of representation languages (often also called **action languages**) adopted by them. Many of such languages have their roots in STRIPS [Fikes-Nilsson-1971] language, that we already mentioned, while enriching it by various features

(note, that STRIPS itself has been modified and revised since its first introduction). As an example of such languages, we can name PDDL (Planning Domain Definition Language) [McDermott et al., 1998, Fox-Long, 2003], providing features like existential and universal quantifiers, ADL (Action Description Language) [Pednault, 1989, Pednault, 1994], enabling conditional effects, or our language IK-STRIPS (STRIPS for Incomplete Knowledge) [Čertický, 2010a, Čertický, 2010b], aimed at non-monotonic representation of partial knowledge. There are also other widely-used languages, evidently deviating from original STRIPS-like pattern by representing the action models by a set of fluents (time and action-dependant variables) and laws (expressing the causality within the domain). Languages like $\mathcal{K}$ [Eiter et al., 2000], $\mathcal{C}$ [Guinchiglia-Lifschitz, 1998] or C+ [Lee-Lifschitz, 2003] can serve as appropriate examples.

2.3 Action Learning: Why do we need to obtain action models automatically?

No matter what the representation language is, the action model must be known if an agent is supposed to plan his actions. Normally, it is hand-written by a programmer or domain expert. However, describing planning domains is extraordinarily difficult and time-consuming task, especially in complex (possibly real-world) domains. This fact is the cause of our ambition - automatically generating action models for our domain (no matter what kind [1]) based on agent's observations over time (this process is referred to as

[1] As a matter of fact, automatic action learning is a necessary condition of an **environmental universality** as described in [Čertický, 2008] and [Čertický, 2009]. If an agent is

action learning).

2.4 Properties of Action Learning Methods

In this section, we will describe the set of **interesting properties**, that any action learning method might, or might not have. Those properties will then serve as the basis for next chapter, where we will evaluate the collection of current action learning approaches.

2.4.1 Usability in Partially Observable Domains

First off, as an example of a **fully observable domain** we may consider a game of chess. Both players (agents) have a full visibility of all the features of their domain - in this case the configuration of the pieces on the board. Such configuration is typically called a **world state**. On the other hand, by **partially observable domain** we understand any environment, in which agents have only limited observational capabilities - in other words, they can see only a small part of the state of their environment (world states are partially observable). Real world is an excellent example of a partially observable domain. Agents of real world (for example humans) can only observe a small part of their surroundings: they can only hear sounds from their closest vicinity (basically several meters, depending on how loud the sounds are), see only objects that are in their direct line of sight (given the light conditions are good enough), etc.

supposed to be usable in various domains, it needs an ability to learn its dynamics from a sequence of observations.

An action learning method is **usable in partially observable domains** only if it is capable of producing useful action models, even if world states are not fully observable.

2.4.2 Learning Probabilistic Action Models

There are two ways of modelling a domain dynamics (creating an action models), depending on whether we want the randomness to be present or not. An action model is **deterministic**, if actions it describes have all a unique set of always successful effects. Conversely, in case of a **probabilistic** (or stochastic) action models, action effects are represented by a probabilistic distribution over the set of possible **outcomes**. Let us clarify this concept with the help of simple "toy domain" called Blocks World (example 2.1), discussed extensively (among others) in [Nilsson-1980], [Russel-Norvig-1995], [Gupta-Nau-1992] or [Slaney-Thiebaux-2001][2].

Example 2.1

The Blocks World domain consists of a finite number of blocks stacked into towers (see figure 2.1) on a table large enough to hold them all. The positioning of towers on the table is irrelevant. Agents can manipulate this domain by moving blocks from one position to another. Action model of the simplest Blocks World versions is composed of only one action $move(B, P_1, P_2)$. This action merely moves a block B from position P_1 to position P_2 (P_1 and P_2 being either another block, or the table).

Deterministic representation of such action would look something like this:

```
Action (
```

[2]Because of its simplicity, we will return to this Blocks World domain several times throughout the text, in order to explain various aspects of action learning process.

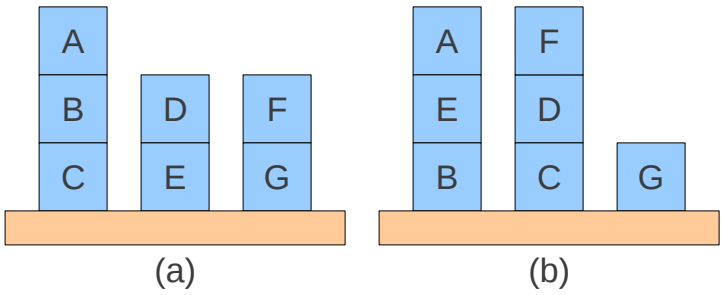

Figure 2.1: Two different world states of Blocks World.

Name & parameters:

$$move(B, P_1, P_2)$$

Preconditions:

$$\{on(B, P_1), free(P_1), free(P_2)\}$$

Effects:

$$\{\neg on(B, P_1), on(B, P_2)\}$$

)

Our action is defined by its name, preconditions, and a unique set of effects $\{\neg on(B, P_1), on(B, P_2)\}$, all of which are applied each time the action is executed. This basically means, that every time we perform an action $move(B, P_1, P_2)$, the block B will cease to be at position P_1 and appears at P_2 instead. In a simple domain like Blocks World, this seems to be sufficient.

In the real world however, the situation is not so simple, and our attempt to move a block can have different outcomes[3]:

[3]Similarly to the variation of Blocks World featuring realistic physics in a three-dimensional simulation presented in [Pasula-Zettlemoyer-Kaelbling, 2007]. We will mention this paper further in our text, since it features an interesting method for probabilistic action learning.

Action (

Name & parameters:

$$move(B, P_1, P_2)$$

Preconditions:

$$\{on(B, P1), free(P_1), free(P_2)\}$$

Effects:

$$\begin{cases} 0.8: & \neg on(B, P_1), on(B, P_2) \\ 0.1: & \neg on(B, P_1), on(B, table) \\ 0.1: & nochange \end{cases}$$

)

This representation of our action defines the following probabilistic distribution over three possible outcomes:

1. 80% chance that block B indeed appears at P_2 instead of P_1,
2. 10% chance that block B falls down on the table,
3. 10% chance that we fail to pick it up and nothing happens.

We can easily see, that probabilistic action models are better suited for real-world domains, or complex simulations of non-deterministic nature, where agent's sensors and effectors are often imprecise and actions sometimes lead to unpredicted outcomes.

2.4.3 Probabilistic Evaluation of Possible World States

Since we are dealing with partially observable domains, our agent typically doesn't have complete knowledge about the current world state. Based on

his limited observations, it can however compute the set of **possible world states**.

Example 2.2

Let us consider the following example of a robot R sharing an appartment with two people A and B (figure 2.2). Robot R can only see people that

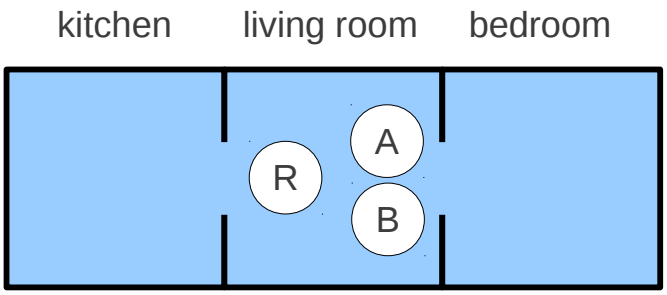

Figure 2.2: Robot R sharing a room with two people A and B at a time t.

are in the same room as he is. His observations at a given time t would be $\{in(R, livingRoom), in(A, livingRoom), in(B, livingRoom)\}$. Now if the robot leaves the living room and moves into the kitchen, his observation at subsequent time $t+1$ would only be: $\{in(R, kitchen)\}$. While the robot is in the kitchen, he doesn't know the location of A and B exactly - he can only tell, that they are not in the kitchen with him. Based on his limited observations, robot R concludes, that current world state can only be one of the following **possible states**:

- $\{in(R, kitchen), \quad in(A, livingRoom), in(B, livingRoom)\}$
- $\{in(R, kitchen), \quad in(A, bedroom), in(B, bedroom)\}$

- $\{in(R, kitchen), \quad in(A, livingRoom), in(B, bedroom)\}$
- $\{in(R, kitchen), \quad in(A, bedroom), in(B, livingRoom)\}$

Now, in order to maintain as precise knowledge as possible, our robot R can try to compute the **probabilistic distribution** over this set of possible world states. There are various methods to do so, while most of them would be based on previous observations. For example, if robot remembered, that most of the times he saw A or B they were together, his **probabilistic evaluation** of possible world states could look something like this:

$35\% \ \ldots \ \{in(R, kitchen), \quad in(A, livingRoom), in(B, livingRoom)\}$

$35\% \ \ldots \ \{in(R, kitchen), \quad in(A, bedroom), in(B, bedroom)\}$

$15\% \ \ldots \ \{in(R, kitchen), \quad in(A, livingRoom), in(B, bedroom)\}$

$15\% \ \ldots \ \{in(R, kitchen), \quad in(A, bedroom), in(B, livingRoom)\}$

Some of the action learning algorithms are capable of producing such distributions, even though it is not their main purpose. Having a probabilistic distribution over possible states is undoubtedly beneficial (among other reasons) for the impending planning process.

2.4.4 Dealing with Action Failures

As we noted in section 2.4.2, actions in non-deterministic domains can have more than one outcome. In typical situation though, each action has one outcome with significantly higher probability than the others. In case of action $move(B, P_1, P_2)$ from Blocks World (example 2.1), this **expected outcome**

was actually moving a block B from position P_1 to P_2. Then if after the execution the block was truly at position P_2, we considered the action successful. If the action had any other outcome, it was considered unsuccessful - we say that it **failed**.

From agent's point of view, action failures pose a serious problem, since it is difficult for him to decide whether given action really failed (due to some external influence), or just his expectations were wrong (if his expectations were wrong, he needs to modify his action model accordingly).

Action learning algorithms need to be able to deal with action failures[4] in order to be usable in non-deterministic domains.

2.4.5 Learning both Preconditions and Effects

Since the introduction of STRIPS [Fikes-Nilsson, 1971] in early 70's, common assumption is, that actions have some sort of **preconditions** and **effects**.

Preconditions[5] define what must be established in a given world state before an action can even be executed. Looking back at Blocks World (example 2.1), the preconditions of action $move(B, P_1, P_2)$ require both positions P_1 and P_2 to be free (meaning that no other block is currently on top of them). Otherwise, this action is considered inexecutable.

[4]Probabilistic action models, as outlined in example 2.1, provide a means for simple solution to this problem by modelling several different outcomes of a given action and defining the probability distribution over them. This method [Pasula-Zettlemoyer-Kaelbling, 2007], among others, will be discussed further in our text.

[5]Preconditions are sometimes called *applicability conditions* - especially when we formalise actions as operators over the set of world states.

Effects[6] simply specify what is established after a given action is executed, or in other words, how the action modifies the world state.

Some action learning approaches either produce effects and ignore preconditions, or the other way around. They are therefore incapable of producing complete action model from the scratch, and thus are usable only in situations when some partial hand-written action model is provided. We want to avoid the necessity to have any prior action model.

2.4.6 Learning Conditional Effects

Research in the field of action description languages has shown, that expressive power of early STRIPS-like representations is susceptible to be improved by addition of so-called conditional effects. This results from the fact, that actions, as we usually talk about them in natural language, have different effects in different world states.

Consider a simple action of person P drinking a glass of beverage B - $drink(P, B)$. Effects of such action would be (in natural language) expressed by following sentences:

- P will cease to be thirsty.
- If B was poisonous, P will be sick.

We can see, that second effect (P becoming sick) only applies *under certain conditions* (only if B was poisonous). We call effects like this a **conditional effects**.

[6] Effects are sometimes called *postconditions* - primarily in the early publications in STRIPS-related context.

STRIPS language for instance didn't support conditional effects. Of course, there was a way to express aforementioned example, but we needed two separate actions with different applicability preconditions for it: $drink\_if\_poisonous(P, B)$ and $drink\_if\_not\_poisonous(P, B)$.

Having a support for conditional effects thus allows us to specify domain dynamics by lower number of actions, making our representation less space consuming and more elegant. Several state-of-the art action languages provide the apparatus for defining conditional effects - see the following example:

Example 2.3
STRIPS extensions like Action Description Language (ADL) [Pednault, 1989, Pednault, 1994] or Planning Domain Definition Language (PDDL) [McDermott et al., 1998, Fox-Long, 2003] express the effects of $drink(P, B)$ action in the following LISP-resembling syntax:

```
:effect     (not (thirsty ?p))
:effect     (when (poisonous ?b) (sick ?p))
```

Definition of same two effects in fluent-based languages like $\mathcal{K}$ [Eiter et al., 2000] on the other hand, employs the notion of so-called *dynamic laws*:

```
caused -thirsty(P) after drink(P,B).
caused sick(P) after poisonous(B), drink(P,B).
```

2.4.7 Online Algorithms and Tractability

Algorithms that run fast enough for their output to be useful are called **tractable** [Hopcroft-Motwani-Ullman, 2007].

Algorithms whose input is served one piece at a time, and upon receiving

it, they have to take an irreversible action without the knowledge of future inputs, are called **online** [Borodin-El-Yaniv, 1998].

For the purposes of action learning we prefer using online algorithms, which run once at each time step - after the observation. Agent's newest observation is served as the input for the algorithm, while there is no way of knowing anything about future observations. Algorithm simply uses this observation to modify agent's knowledge. Since the input of such algorithm is relatively small, tractability should not be an issue here.

If we, on the other hand, decided to use offline algorithms for action learning, we would have to provide the whole history of observations on the input. Algorithms operating over such large data sets are prone to be intractable.

Since online algorithms are designed to run repeatedly during the "life" of an agent, he has some (increasingly accurate) knowledge at his disposal at all times. Offline action learning algorithms are, on the other hand, designed to run only once, after the agent's life, which makes them unusable in many applications.

Chapter 3

Evaluation of Current Methods

In this chapter we will explain five methods that are currently used to solve the task of action learning. Each of them represents a different approach and uses different set of tools and/or representation formalisms to express and generate new action models.

Conclusion of every section contains the **evaluation** of corresponding method based on the set of **properties established in the previous chapter**. For a comprehensive comparison of these methods, see figure 3.1.

Paper	Method name	Partially observable domains	Probabilistic action models	Probabilistic world states	Dealing with action failures	Both precondition s and effects	Conditional effects	Online
[Amir-Chang, 2008]	SLAF	yes	no	no	only when failure is explicitly known	no	no	yes
[Yang-Wu-Jiang, 2007]	ARMS	yes	no	no	no	yes	no	no
[Balduccini, 2007]	A-Prolog with ASP semantics + Learning module	yes	no	no	no	yes	yes	no
[Mourao-Petrick-Steedman, 2010]	Perceptron Algorithm	yes	no	no	yes	no	no	yes
[Pasula-Zettlemoyer-Kaelbling, 2007]	Greedy Search	no	yes	no	yes	yes	yes	no

Figure 3.1: Comparison of current action learning methods based on the properties from previous chapter.

3.1 SLAF - Simultaneous Learning and Filtering

First method we will take a look at is called Simultaneous Learning and Filtering (SLAF) and was published in a paper called *Learning partially observable deterministic action models* [Amir-Chang, 2008] in 2008.

This method produces the action models and possible world states in two phases:

1. Building a propositional formula φ over time by calling the SLAF algorithm once after each observed action (at each time step t).
2. Interpreting this formula φ by using satisfiability (SAT) solver algorithms (e.g., [Moskewitz et al., 2001]).

3.1.1 Phase 1: SLAF Algorithm

Representation language:

Let A be a set of all actions possible in our domain. Let P be a set of all domain's fluent literals (features that can change over time, typically by executing actions). Knowledge about action model and world states is then compactly encoded by logic formulas over a vocabulary $L = P \cup L_f^0$, where $L_f^0 = \bigcup_{a \in A} \{a^f, a^{f\circ}, a^{\neg f}, a^{[f]}, a^{[\neg f]}\}$ for every $f \in P$. The intuition behind propositions in the set L_f^0 is as follows:

a^f : *"a **causes** f"*, meaning that execution of a causes literal f to be true.

$a^{f\circ}$: *"a **keeps** f"*, meaning that execution of a doesn't affect the value of fluent f.

$a^{[f]}$: *"a **causes** FALSE **if** $\neg f$"*, meaning that a literal f is a precondition of a, and it must hold before execution of a.

Also, for a set of propositions P, let P' represent the same set of propositions, but with every proposition primed (i.e. each proposition f is annotated to become f'). Authors use such primed fluents to denote the value of unprimed fluents one step into the future after taking an action. Furthermore, let $\varphi_{[P'/P]}$ denote the same formula as φ, but with all primed fluents replaced by their unprimed counterparts. For example formula $(a \vee b')_{[P'/P]}$ is equal to $(a \vee b)$ when $b \in P$.

SLAF algorithm is called at each time step t with the following input and output:

Input:

- Most recently executed action a ($a \in A$).
- Partial observation σ from current time step ($\sigma \in Pow(P)$).
- Output of previous call of SLAF algorithm in a form of propositional formula φ.

Output:
Propositional logical formula φ over the vocabulary L. φ represents all the combinations of action models that could possibly have given rise to the observations in the input, and all the corresponding states in which the world may now be (after the sequence of time steps that were given in the input occurs). This formula is called a *transition belief formula*.

Output of SLAF algorithm after execution of action a is defined in the recursive fashion:

Definition 1 (SLAF Output)**.**

$$SLAF[a](\varphi) \equiv Cn^{L \cup P'}(\varphi \wedge \tau_{eff}(a))_{[P'/P]}$$

where $\tau_{eff}(a)$ is a propositional axiomatization of action a and $Cn^{L \cup P'}$ is appropriate *consequence finding operator* over a vocabulary $L \cup P'$.

Axiomatization $\tau_{eff}(a)$ is a propositional expression of consequences of the fact that action a was executed:

$$\begin{aligned}
\tau_{eff}(a) &\equiv \bigwedge_{f \in P} Pre_{a,f} \wedge Eff_{a,f} \\
Pre_{a,f} &\equiv \bigwedge_{l \in \{f, \neg f\}} (a^{[l]} \Rightarrow l) \\
Eff_{a,f} &\equiv \bigwedge_{l \in \{f, \neg f\}} ((a^{l} \vee (a^{f \circ} \wedge l)) \Rightarrow l') \wedge (l' \Rightarrow (a^{l} \vee (a^{f \circ} \wedge l))).
\end{aligned}$$

Here $Pre_{a,f}$ states that if l is a precondition of action a, then it must have held in the state before executing a. $Eff_{a,f}$ then states that fluents before and after execution of a must be consistent with the action models defined by propositions a^f, $a^{\neg f}$, and $a^{f\circ}$.

As an efficient consequence finding operator $Cn^{L\cup P'}(\varphi)$, it is possible to use simple *propositional resolution* [Davis-Putnam, 1960, Chang-Lee, 1973]. However, in their implementation, authors use *prime implicates finder* [Marquis, 2000], which is under certain conditions both tractable, and keeps formula φ reasonably short.

3.1.2 Phase 2: Interpreting Results

When we interpret the transition belief formula, we assume the existence of three logical axioms, that disallow *inconsistent* or *impossible* models:

1. $a^f \vee a^{\neg f} \vee a^{f\circ}$
2. $\neg(a^f \wedge a^{\neg f}) \wedge \neg(a^{\neg f} \wedge a^{f\circ}) \wedge \neg(a^f \wedge a^{f\circ})$
3. $\neg(a^{[f]} \wedge a^{[\neg f]})$

for all possible $a \in A$ and $f \in P$. First two axioms mean that action a either **causes** f, **causes** $\neg f$, or **keeps** f. Third axiom says, that action cannot have both f and $\neg f$ as its preconditions.

To find out, if a world state S (or action model M) is possible, we can use any of numerous SAT solvers. If a conjunction $\varphi \cup S \cup AXIOMS$ (resp. $\varphi \cup M \cup AXIOMS$) is satisfiable, then S (M) is possible, and vice versa.

3.1.3 Evaluation

Amir and Chang's method is designed only for **deterministic domains**, where our observations are always true, and **action failures do not occur**[1]. Transition belief formula is then *exact* in a sense, that it represents every possible and no impossible *action model - world state* combinations (given previous observations). In fact, number of such *model - state* combinations is decreasing over time when new observations are added, making our knowledge more precise. However, there is **no probabilistic distribution** over these possible *model - state* combinations.

Authors have shown that their solution is **tractable**, since SLAF algorithm called once in each time step runs in polynomial time, when implemented correctly (for actual pseudo-code of SLAF algorithm see [Amir-Chang, 2008]). Answering certain queries about learned states and action models is more time-consuming problem, since deciding the satisfiability of a propositional formula is NP-complete problem[2]. Current SAT solvers however deal with this problem quite efficiently.

3.2 ARMS - Action-Relation Modelling System

This next method was proposed in a paper called *Learning action models from plan examples using weighted MAX-SAT* [Yang-Wu-Jiang, 2007]. It is

[1] Authors suggested a solution for situations with possible action failures, but it depended on unrealistic assumption, that agent always knew whether given action was successful, even when he still didn't know what action was supposed to do.

[2] Cook-Levin theorem [Cook, 1971].

closely related to the SLAF method by Amir et al. (described in section 3.1) due to similar objectives (learning action models) and usage of satisfiability solvers. Input needed for this technique is however different. Amir et al.'s method needs each observed action to be followed by a **state observation**. In the absence of these observations, SLAF algorithm wouldn't work. ARMS method, on the other hand, takes only a set of so-called **plan examples** as an input, and doesn't need any state observations between individual actions. Also, resulting action models are *approximations*, in contrast to SLAF, which aimed to find the *exact solutions*.

The ARMS method consists of running a single recursive algorithm, consisting of the following six steps:

1. Convert all the observed action instances to their **schema forms** by replacing the constants by corresponding variables (actual representation language used here is PDDL - it will be described in the following subsection).

2. For all the unexplained actions, build the set of logical clauses of three types called **action**, **information**, and **plan constraints**. Also use a *frequent-set mining algorithm* [Agrawal, 1994] to find a pairs of actions (a, b), such that a frequently coexists with b in our observed plans.

3. Assign weights to all clauses (constraints) generated in preceding step.

4. Solve the weighted MAX-SAT problem over this set of clauses with assigned weights using external MAX-SAT solver. In their experiments, authors used both http://www.nmt.edu/ brochers/maxsat.html and the MaxWalkSat solver [Kautz-Selman-Jiang, 1997, Selman-Kautz-Cohen, 1993].

5. Update our current knowledge with the set of action models with highest weights. If we still have some unexplained actions, go to step 2.

6. Otherwise terminate the algorithm.

Let us now take a closer look at the ARMS algorithm, its input, output, and individual steps.

3.2.1 ARMS Algorithm

Input:
Plan example is a triple consisting of:

1. **initial state**,

2. **goal state**, both described by a list of propositions,

3. and a **sequence of actions**, represented by an *action name* and *instantiated parameters*.

Each plan in the input of ARMS algorithm must be *successful* in that it achieves the goal from the initial state. In the input instances to ARMS, the plan examples may only contain **action instances** - expressions like $move(block01, pos01, pos02)$, where $block0$, $pos01$ and $pos01$ are constants. These plan examples do not provide action's preconditions and effects. These are to be learned by the algorithm.

Representation Language and Output:
Resulting action model is expressed in Planning Domain Definition Language - PDDL [McDermott et al., 1998, Fox-Long, 2003]. This action language provides us, among other useful features, with type definitions for

variables/constants. To describe the PDDL language, we will once again use the $move(B, P_1, P_2)$ action of Blocks World (example 2.1):

```
(:action move
  :parameters (?b - block ?p1 ?p2 - position)
  :precondition (and (on ?b ?p1) (free ?p1) (free ?p2))
  :effect (and (not (on ?b ?p1) (on ?b ?p2))
)
```

Symbols **?b**, **?p1**, and **?p2** (or any lowercase letter preceded by a question mark) are used to denote variables. In the **:parameters** section of this action expression, we have the **type definitions**. Expression "?b - block" simply says, that the variable **?b** can only be substituted by a constant c, if c is a block - that is, if a "(block c)" clause holds in our domain.

Converting Actions to their Schema Forms:
A plan examples consist of a sequences of *action instances* (with constant parameters). ARMS algorithm *converts* all such plans by substituting all occurrences of instantiated object in every action instance with the variable of the same type. That of course means, that the algorithm needs to have access to domain specific knowledge about the types of individual constants.

Building Constraints:
Each iteration of ARMS algorithm generates the set of logical clauses (called *constraints*), which later serve as an input for weighted MAX-SAT solver. There are three types of such clauses:

1. **Action Constraints** are imposed on individual actions. They are derived from the *general axioms of correct action representations*. Let pre_i, add_i, and del_i be preconditions, add list, and delete list of action

a_i respectively. Generated *action constraints* are then following two clauses:

(A.1) $pre_i \cap add_i = \emptyset$

The intersection of precondition and add list of all actions must be empty.

(A.2) $del_i \subseteq pre_i$

In addition, for every action we require that the delete list is a subset of the precondition list.

2. **Information Constraints** are used to explain *why the (optionally) observed intermediate state exists in a plan.* Suppose we observe p to be true between actions a_n and a_{n+1}. Also let $\{a_{i_1} \dots a_{i_k}\}$ be the set of actions from our current plan (appearing in that order), which share the parameter type with p. Then:

(I.1) $p \in add_{i_1} \vee p \in add_{i_2} \vee \dots \vee p \in add_{i_k}$

p must be generated by an action $a_{i_k} (a \leq i+k \leq n)$, which means that it is in the add list of a_{i_k}.

(I.2) $p \notin del_{i_k}$

Last action a_{i_k} must not delete p, which means that it must not occur in delete list of a_{i_k}.

3. **Plan Constraints** represent the relationship between actions in a plan that *ensures that the plan is correct* and that the *actions' add list clauses are not redundant*:

(P.1) Every precondition p of every action b must be in the add list of preceding a and is not deleted by any actions between a and b.

(P.2) At least one clause r in the add list of action a must be useful for achieving a precondition of some later action.

While constraints P.1 and P.2 provide the general guide principle for ensuring the correctness of a plan, in practice there are *too many instantiations* of these constraints. To ensure the efficiency, authors suggest to generate them not for every possible action pair, but only for pairs of actions that frequently coexist in input plan examples. They employ the *frequent-set mining algorithm* [Agrawal, 1994] to find such pairs.

Assigning Weights

To solve a weighted MAX-SAT problem in Step 3 of ARMS algorithm, each constraint clause must be associated with a weight value between zero and one. The higher the weight, the higher the priority in satisfying the clause. To assign the weights, authors apply the following heuristics:

1. **Action Constraints** receive an empirically determined constant weight $W_A(a)$ for an action a.

2. **Information Constraints** also receive empirically determined constant weight $W_I(r)$ for a clause r. This assignment is generally the highest in all constraint's weights.

3. **Plan Constraints** take the weight value $W_P(a, b)$ equal to prior probability of the occurrence of action pair (a, b) in plan examples (while taking into account predetermined *probability threshold* or *minimal support-rate*).

3.2.2 Evaluation

The method of Yang et al. is unique among current action learning approaches in a fact, that it doesn't require the world state observations between individual actions (while still being able to use them if they are present), which can prove particularly useful in some situations. As its input however, it requires successful plan examples, which not only means that it **cannot deal with action failures**, but also that we need to have extra information about initial and goal states.

The fact that resulting action models are represented by PDDL language with type definitions means, that subsequent planning is potentially faster. The downside of this is, that domain-specific background knowledge (about the types of individual constants) is needed for successful learning. **Conditional effects** are **not produced** by ARMS algorithm, even though such construct is allowed in PDDL. As for **probabilistic representation** of actions, they are **not supported** by PDDL at all.

In conclusion, ARMS method is designed for learning **deterministic** action models (with both **preconditions** and **effects**) from the successful plan examples instead of *action-state* observations. Learning process is in its essence highly heuristic, which is necessary, since it's based on the NP-hard weighted MAX-SAT problem.

3.3 Learning through Logic Programming and A-Prolog

In a paper called *Learning Action Descriptions with A-Prolog: Action Language* $\mathcal{C}$ [Balduccini, 2007], Marcello Balduccini demonstrated how action models can be learned in a purely declarative manner, using non-monotonic logic programming with stable model semantics.

The prior goal of his paper was to show, that a representation language A-Prolog [Gelfond-Lifschitz, 1988, Gelfond-Lifschitz, 1991] is a powerful formalism for knowledge representation and reasoning, usable (besides planning and diagnosis) also for learning tasks. Action learning here is based solely on the semantics of A-Prolog, and includes both *addition* of new, and *modification* of existing knowledge about domain dynamics.

Representation language chosen for the expression of actual action models is however not A-Prolog, but $\mathcal{C}$ [Guinchiglia-Lifschitz, 1998] ($\mathcal{C}$ is more suitable, since it was originally designed as an action language).

Balduccini's method can be viewed as a simple three-step process:

1. **Translation** of current knowledge from $\mathcal{C}$ to a set of **A-Prolog facts**.
2. Using so-called **learning module** (also a set of A-Prolog facts) in conjunction with observation **history** to compute the set of stable models by an external ASP solver[3].

[3]ASP (or Answer Set Programming) solvers (for example DLV [Leone et al., 2006] or Smodels [Niemela-Simons-Syrjanen, 2000]) take logic program as an input and return the set of *stable models* (also called *answer sets*) on the output. Stable models represent all the possible interpretations of input logic program under the *stable moel semantics*.

3. Translation of resulting stable models back to action language $\mathcal{C}$.

3.3.1 Translation to A-Prolog

Overview of $\mathcal{C}$

$\mathcal{C}$ is one of the languages, that quite strongly deviate from the typical STRIPS-like pattern - it doesn't encapsulate individual actions. Instead, it conveniently describes the whole domain dynamics by a set of **laws**. In general, there are two types of such laws:

1. **Static laws** of the form: **caused** r **if** $l_1, \ldots, l_n$
2. **Dynamic laws** of the form: **caused** r **if** $l_1, \ldots, l_n$ **after** $p_1, \ldots, p_m$

Here r (the *head* of the law) is a literal or $\bot$, $l_1, \ldots, l_n$ (the *if-preconditions*) are all literals, and $p_1, \ldots, p_m$ (the *after-preconditions*) are either literals, or actions. More detailed definition of a $\mathcal{C}$ language can be found (with slightly different terminology) in [Lifschitz-Turner, 1999].

To illustrate how this language is used in practice, take one more look at our $move(B, P_1, P_2)$ action of Blocks World (example 2.1), this time encoded in $\mathcal{C}$:

caused $on(B, P_2)$ if $\emptyset$ after $move(B, P_1, P_2), on(B, P_1), free(P_1), free(P_2)$
caused $\neg on(B, P_1)$ if $\emptyset$ after $move(B, P_1, P_2), on(B, P_1), free(P_1), free(P_2)$

Note that "if $\emptyset$" part can be omitted in this simple example, since the set of *if-preconditions* is empty.

Overview of A-Prolog

A-Prolog is a knowledge representation language that allows formalisation of

various kinds of common sense knowledge and reasoning. The language is a product of research aimed at defining a formal semantics for logic programs with default negation [Gelfond-Lifschitz, 1988], and was later extended to allow also a classical (or *explicit*) negation [Gelfond-Lifschitz, 1991].

A **logic program** in A-Prolog is a set of **rules** of the following form:

$$h \leftarrow l_1, \ldots, l_m,\ not\ l_{m+1}, \ldots\ not\ l_n.$$

where h and $l_1, \ldots, l_n$ are literals and *not* denotes a default negation. Informally, such rule means that "if you believe $l_1, \ldots, l_m$, and have no reason to believe any of $l_{m+1}, \ldots, l_n$, then you must believe h". Part to the right of the "$\leftarrow$" symbol ($l_1, \ldots, l_m,\ not\ l_{m+1}, \ldots\ not\ l_n$) is called a *body*, while part to the left of it (h) is called a *head* of the rule. Rules with an empty body are called *facts*. Rules with empty head are called *constraints*.

Semantics of A-Prolog is built on the concept of **stable models**. Consider a logic program Π and a consistent set of literals S. We can then get a subprogram called *program reduct of* Π *w.r.t. set* S (denoted Π^S) by removing each rule, that contains *not* $l; l \in S$ in its body. Also, we say that a set of literals L is *closed* under a rule a, if holds $body(a) \in L \Rightarrow head(a) \in L$.

Definition 2 (Stable Model)**.** Any minimal set of literals *closed* under all the rules of Π^S is called a *stable model* of Π w.r.t. set S.

Intuitively, a stable model represents one possible interpretation of knowledge encoded by logic program Π and a set of literals S.

Translation

Even though it is possible to translate each $\mathcal{C}$ law into *one* A-Prolog rule [Lifschitz-Turner, 1999], Balduccini rather decided to translate it **into a set of facts**.

Let us now take a $\mathcal{C}$ law and denote it $w(v_1, \ldots, v_k)$, where $v_1, \ldots, v_k$ is a list of variables appearing in it (in previous example it would be B, P_1, P_2) and prefix w is just any name assigned to it. Balduccini's method calls this this prefix $w^{\mathcal{P}}$, and its variable list $w^{\mathcal{V}}$.

Static law w is now encoded into following set of A-Prolog facts:

$$s\_law(w^{\mathcal{P}}).$$

$$head(w^{\mathcal{P}}, r^w).$$

$$vlist(w^{\mathcal{P}}, \lambda(w^{\mathcal{V}})).$$

$$if(w^{\mathcal{P}}, \langle l_1^w, \ldots, l_n^w \rangle).$$

First two facts say, that symbol $w^{\mathcal{P}}$ is reserved for a *static law* and that a head of this law is literal r^w. Third fact describes how variables of this law are *groundified* (subsequently substituted by constants[4]). Fourth fact enumerates all the literals representing *if-conditions* of this law.

Semantic meaning of any static $\mathcal{C}$ law is preserved by addition of the following A-Prolog rule (uppercase letters are used here to denote variables):

$$\begin{aligned} h(H,T) \leftarrow\ & s\_law(W), vlist(W, VL), \\ & head(W, H_g), gr(H, VL, H_g), \\ & all\_if\_h(W, VL, T). \end{aligned}$$

Here $gr(H, VL, H_g)$ intuitively means, that H can be groundified to H_g using the variable mapping from VL, and $all\_if\_h(W, VL, T)$ means, that all the if-preconditions of a law W hold in time step T w.r.t. the same variable

[4]In A-Prolog, we call rules that do not contain variables *ground*. Note, that non-ground rules are semantically equivalent to the set of their ground instances.

mapping. Intuitive meaning of literal $h(H,T)$ is that head H holds in the step T.

Dynamic law w is translated into A-Prolog in a similar manner:

$d\_law(w^{\mathcal{P}})$.

$head(w^{\mathcal{P}}, r^w)$.

$vlist(w^{\mathcal{P}}, \lambda(w^{\mathcal{V}}))$.

$if(w^{\mathcal{P}}, \langle l_1^w, \ldots, l_n^w \rangle)$.

$after(w^{\mathcal{P}}, \langle p_1^w, \ldots, p_m^w \rangle)$.

First fact means, that symbol $w^{\mathcal{P}}$ is reserved for a *dynamic law*. Second, third and fourth law are all identical to previous case. Last fact is added to enumerate all the *after-preconditions* of law w.

Semantic meaning of dynamic laws is expressed by the following rule:

$$\begin{aligned} h(H, T+1) \leftarrow\ & d\_law(W), vlist(W, VL), \\ & head(W, H_g), gr(H, VL, H_g), \\ & all\_if\_h(W, VL, T), all\_after\_h(W, VL, T). \end{aligned}$$

Meaning of this rule is similar to previous simpler case, except it describes the causal dependence between two time steps T and $T+1$.

In addition to the ones that we mentioned, there are some more A-Prolog rules defining the complete semantics of language $\mathcal{C}$. In order to save space, we must omit them.

In this manner, we are able to translate all the static and dynamic $\mathcal{C}$ laws into A-Prolog. Additionally, **observations** are encoded by statements of

the form $obs(l, t)$ or $hpd(a, t)$, meaning that either a literal l was observed, or an action a happened at time step t (respectively). The collection of such literals over a time period is called **history**.

Deciding if the Modification is Needed

The next step of Balduccini's method is detection of the need for learning. When given a history, an agent should check if it is consistent with its current action description. If there is at least one stable model, they are consistent, and we say that a history can be explained by our description. Otherwise we need to rebuild our action description by **appending a learning module** to the semantic definitions and history, and **re-calculating the stable models**.

3.3.2 Learning Module

Learning module is simply the following set of fifteen rules:

1. $\{if(W, N, L_g)\}$
2. $\leftarrow if(W, N, L_{g_1}), if(W, N, L_{g_2}), L_{g_1} \neq L_{g_2}.$
3. $\leftarrow has\_if(W, N), N > 1, not\ has\_if(W, N-1).$
4. $\leftarrow if(W, N, L_g), not\ valid\_gr(W, N, L_g).$
5. $\{d\_law(W), s\_law(W)\} \leftarrow available(W).$
6. $\leftarrow d\_law(W), s\_law(W).$
7. $\{head(W, H_g)\} \leftarrow newly\_defined(W).$
8. $\leftarrow newly\_defined(W), not\ has\_head(W).$

9. $\leftarrow head(W, H_{g_1}), head(W, H_{g_2})$.

10. $\leftarrow head(W, H_g), not\ valid\_gr(W, N, H_g)$.

11. $\{after(W, N, A_g)\} \leftarrow d\_law(W)$.

12. $\leftarrow after(W, N, A_{g_1}), after(W, N, A_{g_2}), A_{g_1} \neq A_{g_2}$.

13. $\leftarrow has\_after(W, N), N > 1, not\ has\_after(W, N-1)$.

14. $\leftarrow after(W, N, A_g), not\ valid\_gr(W, N, A_g)$.

15. $vlist(W, VL) \leftarrow newly\_defined(W), avail\_list(W, VL)$.

Space limitations prevent us from explaining all of the rules here, but we can clearly see how new laws are generated when we take a look at rules (5) and (6). Rule (5) intuitively says, that any available constant can be used as prefix of a new dynamic or static law, while rule (6) says, that it cannot be used for both. For the full explanation of the learning module, see [Balduccini, 2007].

Individual rules of this learning module together with definitions of $\mathcal{C}$ semantics cause ASP solver to produce stable models representing all the reasonable models of domain dynamics. Models, that are in addition to that consistent with observation history, represent our valid models. Naturally, when the history grows over time, number of such models is decreasing, thus making our knowledge more precise.

3.3.3 Evaluation

The main advantage of Balduccini's method is its declarative character, meaning that basically nothing new needs to be implemented in order to

use it. Problem is, that proposed A-Prolog encoding of knowledge is too robust, and using ASP solvers to compute stable models for non-monotonic logic programs of this length is **intractable**. In addition to that, the **whole observation history** needs to be checked at each time step, making the computational time grow exponentially.

Interestingly for us, the non-monotonic character of A-Prolog allows the method to be used with **partial observations**. The action language $\mathcal{C}$, chosen to represent the output, implicitly allows and encourages the use and generation of **conditional effects**.

On the other hand, **deterministic character** of this representation **disallows** the production of **probabilistic action models** and makes dealing with **action failures impossible**.

In conclusion, we need to say that this method was never intended to be used in complex practical applications. It was rather designed to analyse and demonstrate certain properties of A-Prolog formalism. However, like many other AI techniques that bring into play ASP solvers, it is still usable in domains, that are simple enough, and computation process doesn't need to be quick.

3.4 Kernelised Voted Perceptrons and Deictic Coding

This next method described by Mourao, Petrick and Steedman in a paper called simply *Learning action effects in partially observable domains*[5] [Mourao-Petrick-Steedman, 2010] is based on well known **Perceptron Algorithm** introduced by F. Rosenblatt in 1958 [Rosenblatt, 1958]. What is interesting about this method is, that it links the high-level reasoning with the low-level robot/vision system.

It aims at learning only the effects of STRIPS-like actions (not their preconditions) and treats this problem as a **set of binary classification problems**, which are solved using the **kernelised voted perceptrons** [Freund-Schapire, 1999, Aizerman-Braverman-Rozoner, 1964].

Since all the effects here are in the form of conjuntions of literals, it is sufficient to learn the rule for each effect fluent separately (one binary classification problem per fluent). Authors have chosen to use **perceptrons** [Rosenblatt, 1958] to address this particular learning problem, since they are simple, yet fast binary classifiers.

Specifically, having a strictly given set P of perceptrons, where $\forall p \in P : p$ corresponds to one possible effect fluent, we perform the following routine at each time step:

1. Choosing a relevant subset of fluents using the **deictic coding** approach.

[5] This paper extends the earlier method proposed in [Mourao-Petrick-Steedman, 2008] by providing the better support for incomplete observations and noisy domains.

2. Representing currently observed world state as a **vector over** $\{-1, 0, 1\}$.

3. Using this vector as an input for each $p \in P$ and **adjust the internal weight** w of p accordingly (training).

This way, after a sufficient number of observations, each $p \in P$ can be used to predict whether its corresponding fluent will change, given a *state-action* pair on the input.

3.4.1 Deictic Coding

Before doing anything else, authors compute a reduced form of the input *state-action* pair using an approach called deictic coding [Agre-Chapman, 1987]. For a given action instance A, they construct the set of **objects of interest** $\mathcal{O}^{\mathcal{A}}$ by combining a *primary set of objects* (given by parameters of A) and *secondary set of objects*, which are directly *connected* to any of the objects in the primary set. We say, that two objects c_i and c_j are connected in a state s, iff $\exists\ l(c_1, \ldots, c_n) \in s$ such that $c_i, c_j \in \{c_1, \ldots, c_n\}$.

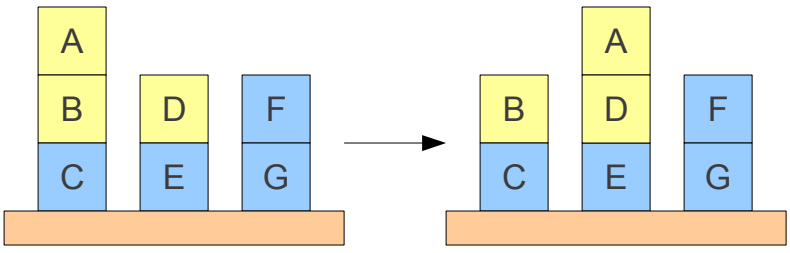

Figure 3.2: Action instance $move(A, B, D)$ in the Blocks World domain.

Let us take a look at the example configuration of the Blocks World (figure 3.2). Given an action $move(A, B, D)$ (moving a block A from block B to

D), our primary set of objects would be $\{A, B, D\}$, secondary set would be $\{C, E\}$ (since we have literals $on(B, C)$ and $on(D, E)$ in our knowledge base), and we would ignore all the other objects ($\{F, G, table\}$). $\mathcal{O}^{\text{move(A,B,D)}}$ is then a set $\{A, B, D, C, E\}$.

Intuitively, deictic coding is used to heuristically filter out all the irrelevant (w.r.t. current action) objects from the input world state. Authors have shown, that it not only speeds up the learning process significantly, but also allows this method to be scaled up to a very large domains. On the other hand it prevents learning such effects, that are seemingly not connected to our action at the moment (just because of the lack of knowledge about this connection). Consequently, it makes us more dependant on the prior background knowledge.

3.4.2 Input/Output Representation

As we mentioned before, the **input** state-action pairs are translated into the vectors over $\{-1, 0, 1\}$. This translation is done as follows:

Each possible action and each 0-ary (object-independent) fluent is represented by one digit. Then for each object $o \in \mathcal{O}^{\mathcal{A}}$, all the possible relations between o and all other objects in $\mathcal{O}^{\mathcal{A}}$ must be represented by an additional digit. This digit equals to 1 if corresponding fluent is observed *true*, or if corresponding action is the *current* action. It equals to -1 if fluent is observed *false*, or if the action is *not* the *current* one. If a digit corresponds to a fluent which was *not currently observed*, its value is set to 0.

Vector representation of a state-action pair from figure 3.2 follows. Note however, that in this simplistic case we have only one action, no object-

independent fluents, and we can observe the whole world state:

Actions:
1 move(A,B,D)
Properties of A:
1 free
-1 on-A
1 on-B
-1 on-C
-1 on-D
-1 on-E
Properties of B:
-1 free
-1 on-A
-1 on-B
1 on-C
-1 on-D
-1 on-E
... and similarly for objects D,C and E.

The form of the **output** vectors representing the action's effect on a state are identical to input vectors, except that the actions themselves are excluded from it, and digits are set to 1 if the corresponding fluent changes, -1 if it stays the same, and 0 if it wasn't observed neither before, nor after the action.

The **ordering** of the digits is of course constrained, so that two objects with the same role in two instances of the same action must share the same

position in the vector.

3.4.3 Training the Perceptrons

The perceptron **maintains a weight vector** w which is adjusted at each time step. The i-th input vector x_i is usually[6] classified by a perceptron using the decision function $f(x_i) = sign(\langle w \cdot x_i \rangle)$. Then if $f(x_i)$ is not a correct class, weight w is set either to $w + x_i$ or to $w - x_i$ (depending on whether the mistake was negative or positive). If $f(x_i)$ is correct, w stays unchanged.

This *perceptron algorithm* is guaranteed to converge on a solution in a finite number of steps only if the data is *linearly separable* [Minsky-Papert, 1969]. Unfortunately, this is in general not our case, and we need to deal with this problem.

One solution for non-linearly separable data is to **map input feature space into a higher-dimensional space**, where the data is linearly separable. Since explicit mapping is problematic, due to a massive expansion in the number of features, authors decided to apply **kernel-trick** [Freund-Schapire, 1999], that allows an *implicit mapping*.

Note, that the decision function can be written in terms of the dot product of the input vectors:

$$f(x_i) = sign(\langle w \cdot x_i \rangle) = sign(\sum_{j=1}^{n} \alpha_j y_j \langle x_j \cdot x_i \rangle)$$

where α_j is the number of times the j-th example has been misclassified by the perceptron, while $y_j \in \{-1, 1\}$ says whether the mistake was positive or negative. By replacing the typical dot product with a *kernel function*

[6] In original Perceptron Algorithm.

$k(x_i, x_j)$ which calculates $\langle \phi(x_i) \cdot \phi(x_j) \rangle$ for some mapping ϕ, the perceptron algorithm can be applied in higher dimensional spaces without ever requiring the mapping to be explicitly calculated. In their implementation, authors used the polynomial kernel of degree 3: $k(x, y) = (x \cdot y + 1)^3$

3.4.4 Evaluation

Mourao et al.'s method aims to learn action effects of **partially observable** STRIPS domains. For this purpose, authors decided to use a *voted kernel perceptron learning model*, while encoding the knowledge about domain into a binary vectors. The choice of perceptrons as the basis for learning allows elegant handling of **noise** and **action failures**, while keeping the computational complexity relatively low. However, even though this learning method is in principle probabilistic, binary character of used vector-based representation **doesn't allow production of probabilistic** action models.

Unfortunately, this solution lacks a certain degree of versatility, since it cannot generate **neither preconditions, nor the conditional effects** of the actions.

Usability of this solution in **large domains** is achieved by applying the deictic coding to filter out irrelevant parts of world states. On the downside, using this approach intensifies our dependency on prior background knowledge, and prevents finding the effects affecting seemingly unrelated parts of the world.

3.5 Greedy Search for Learning Noisy Deictic Rule Sets

Last section of this chapter describes the action learning method introduced by Pasula, Zettlemoyer, and Kaelbling, which falls into the category of algorithmic techniques (in contrast to Balduccini's purely declarative approach (section 3.3)). The paper was published in 2007 and is called *Learning Symbolic Models of Stochastic Domains* [Pasula-Zettlemoyer-Kaelbling, 2007].

Authors apply the **greedy search algorithms** to the problem of modelling **probabilistic actions** in **noisy** domains. For the purpose of evaluation they designed a simulation of three-dimensional Blocks World with realistic physics.

On the input, their algorithm takes a training set E consisting of examples of a form (s, a, s'), where a denotes an action instance, and s, s' are preceding and following (fully observed) world states. It then searches for an action model A that maximizes the likelihood of the action effects seen in E, while penalizing candidate models based on their complexity.

For representing generated action models, the notion of *Noisy Deictic Rules (NDRs)* was introduced. Since, as authors claim, learning the sets of these rules is in general NP-hard [Zettelmoyer-Pasula-Kaelbling, 2003], the choice of greedy approach is adequate. Their search algorithm is hierarchically structured into three levels of greedy search:

1. **LearnRules** method represents the outermost level. It searches the space of possible rule sets, often by constructing new rules, or altering existing ones.

2. **InduceOutcomes** method is repeatedly called by the *LearnRules* method, thus representing the middle level. It generates the set of possible outcomes for a given rule.

3. **LearnParameters** method, called by the *InduceOutcomes*, then finds a probabilistic distribution over these outcomes, while optimizes the likelihood of examples covered by this rule.

After the description of used representation language, we will discuss these three levels starting from inside out, so that each subroutine is described before the one, that depends on it.

3.5.1 Representation by NDRs

Each *Noisy Deictic Rule* specifies a small number of action **outcomes**, representing individual ways how the action can affect the world, along with the probabilistic distribution over them. It consists of four components:

1. **Name** of the action with **parameters**,

2. a list of **deictic references** introducing additional variables and defining their types,

3. description of **context** in which the rule applies,

4. and a set of **outcomes** with assigned **probabilities** summing up to 1.0 (including a compulsory *noise* outcome).

Our traditional action $move(B, P_1, P_2)$ of Blocks World (example 2.1) can be represented as a set of two NDR rules quite easily, as depicted in figure

3.3. Deictic reference part ($\{T : table(T)\}$) of these rules serves simply to introduce additional variable T and define its type. We can see, that two rules are applicable in different *contexts* - first rule applies if a position P_1 is not too high[7], while the second rule describes the situation when we are trying to put B on top of a tall pile of blocks.

$move(B, P_1, P_2) : \{T : table(T)\}$
$free(P_1), free(P_2), on(B, P_1), height(P_1) < 10$

$$\left\{ \begin{array}{ll} 0.7: & \neg on(B, P1), on(B, P2) \\ 0.1: & \neg on(B, P1), on(B, T) \\ 0.1: & no\ change \\ 0.1: & noise \end{array} \right\}$$

$move(B, P_1, P_2) : \{T : table(T)\}$
$free(P_1), free(P_2), on(B, P_1), height(P_1) \geq 10$

$$\left\{ \begin{array}{ll} 0.3: & \neg on(B, P1), on(B, P2) \\ 0.2: & \neg on(B, P1), on(B, T) \\ 0.1: & no\ change \\ 0.4: & noise \end{array} \right\}$$

Figure 3.3: Action $move(B, P_1, P_2)$ of the Blocks World domain represented as a *noisy deictic rule*.

Intuitively, outcomes of the first rule mean, that with probability of 70%, we succeed in moving the block B to P_2, while with 10% probability it either falls on the table, or nothing changes (for example if we failed to pick it up).

[7]Suppose, that $height(P_1)$ is a function counting how many blocks are stacked under the position P_1.

Last "*noise*" outcome sums up all the other things, that could happen, but are so unlikely, that we don't want to model them individually.

Situations covered by the second rule are less stable. If our block is on top of a big stack of blocks, the chance that we successfully move it is only 30%, while the chance that it falls on the table is 20%. There is a high probability here, that the big stack of blocks topples. All the possible configurations of blocks after that happening are included in the "*noise*" outcome.

Probability Calculation

Since the effects of noise are not explicitly modelled, we cannot precisely calculate **the probability** $P(s'|s,a,r)$ that a noisy deictic rule r assigns to moving from state s to state s' when action a is taken. We can however bound the probability as:

$$\begin{aligned}\hat{P}(s'|s,a,r) &= p_{min}P(\Psi'_{noise}|s,a,r) + \sum_{\Psi'_i \in r} P(s'|\Psi'_i,s,a,r)P(\Psi'_i|s,a,r) \\ &\leq P(s'|s,a,r)\end{aligned} \tag{3.1}$$

where $P(\Psi'_i|s,a,r)$ is a probability assigned to outcome Ψ'_i and the outcome distribution $P(s'|\Psi'_i,s,a,r)$ is a deterministic distribution that assigns all of its mass to the relevant s'. The p_{min} constant assigns a small ammount of probability mass to every possible next state s'. To ensure that the probability model remains well-defined, p_{min} multiplied by the number of possible states should not exceed 1.0.

Note, that in this representation it is possible for a resulting state s' to be covered by more than one outcome. The probability of s' occurring after the execution of a given action is the sum of probabilities associated with relevant outcomes.

Compulsory Default Rule

In this formalism, any valid **rule set** representing an action model needs to contain one additional rule - *default rule* which has an **empty context** and two possible outcomes: **no change** and **noise**. This rule expresses the probability of change, if no other rule applies.

3.5.2 Scoring Metric

A greedy search algorithm needs some kind of **scoring metric** to **judge** which parts of the search space are more desirable.

Definition 3 (Scoring metric for *rules* and *rule sets*)**.** Let E_r denote the set of input examples covered by rule r. Scoring metric $S(r)$ over a rule r is then computed as

$$S(r) = \sum_{(s,a,s') \in E_r} log(\hat{P}(s'|s,a,r)) - \alpha PEN(r)$$

where $PEN(r)$ is a complexity penalty applied to rule r, and α is the scaling parameter[8]. Scoring metric $S(R)$ over a rule set R is then simply a sum of scores of all the rules $r \in R$:

$$S(R) = \sum_{r \in R} S(r)$$

We can see that $S(R)$ favours the rule sets that maximize the probability bound, while penalizing those, that are overly complex.

[8] Authors used the scaling parameter $\alpha = 0.5$ in their experiments.

3.5.3 Learning Parameters

The *LearnParameters* method takes an incomplete rule consisting of an *action name with parameter list*, a set of *deictic references*, a *context*, and a set of *outcomes*, and **learns the distribution** P that maximizes the score of r on the examples E_r covered by it. The optimal distribution is simply the one that maximizes the log likelihood of E_r. In our case this will be:

$$L = \sum_{(s,a,s') \in E_r} log(\hat{P}(s'|s,a,r))$$

fortunately, even though estimating the maximum-likelihood parameters is nonlinear programming problem, it is an instance of well-studied problem of maximizing a convex function over a probability simplex. Several gradient ascent algorithms with guaranteed convergence are known for this problem [Bertsekas, 1999]. The *LearnParameters* algorithm uses the **conditional gradient method**, which works by, at each iteration, moving along the axis with the maximal partial derivative. The step sizes are chosen using the **Armijo rule** discussed in [Armijo, 1966].

3.5.4 Inducing Outcomes

Input for the *InduceOutcomes* algorithm is an incomplete rule lacking the whole outcomes part. Its purpose is to find the optimal way to **fill the set of outcomes** and its associated probability distribution that maximizes the score according to metric from definition 3.

InduceOutcomes performs a greedy search through a restricted subset of possible outcome sets which cover at least one training example.

It moves through this space until there are no more immediate moves that

improve the rule score. To generate outcome sets, two operators are used:

- *Add* operator picks a pair of two non-contradictory outcomes and creates a new one, that is their conjunction.
- *Remove* operator drops an outcome from the set, if it is overlapping with another outcome on all the covered examples.

For each set of outcomes it considers, *InduceOutcomes* calls *LearnParameters* to supply the best possibility distribution it can.

3.5.5 Learning Rules

Now that we know how to fill in incomplete rules, we will describe *LearnRules*, the outermost level of our learning algorithm, which takes a set of examples E and performs a greedy search through the space of *proper* rule sets. We say, that a rule set is *proper* w.r.t. E, if it includes at most one rule that is applicable to every example $e \in E$ (in which a change occurs) and if it doesn't include inapplicable rules.

Search starts with a rule set containing only the *default rule* and applies some of the following set of **operators** to obtain new rule sets:

1. *ExplainExamples*
2. *DropRules*
3. *DropLits*[9]
4. *DropRefs*

[9]Here *"Lits"* stands for *literals*.

5. *GeneralizeEquality*
6. *ChangeRanges*
7. *SplitOnLits*
8. *AddLits*
9. *AddRefs*
10. *RaiseConstants*
11. *SplitVariables*

Due to relative complexity of these operators and limited space, we will not describe them here. See [Pasula-Zettlemoyer-Kaelbling, 2007] for their detailed specification.

For assigning the score to newly produced rule sets, score metric from definition 3 is used again.

3.5.6 Evaluation

This method proposed by Pasula et al. is a typical example of algorithmic approach to the problem of action learning. Authors experimented with using relatively simplistic *greedy search* algorithm in conjunction with an appropriate representation language for the purpose of learning **probabilistic action models** including both **preconditions** (in a form of *contexts* in this case) and **effects**.

Even though the method effectively deals with **action failures** or any other kind of **noise**, there are some significant downsides that we need to mention.

The most restricting disadvantage of this technique is the **lack of support for partially observable domains**.

Usability in real-world domains (even if we overlooked the *full-observation* requirement) is diminished also by the fact, that the algorithm they used is a three-level search over the increasingly large dataset - the set of all previously observed **examples** is **needed** on the input, making the computational complexity a big issue.

Chapter 4

Project Proposal

In the first section of this chapter, we will gradually form the notion of **action learning** until we reach its formal definition. At the same time we will explain all the necessary **terminology**, and introduce some novel **representation structures**, which may be used in our future solutions.

When all this is done, we will be able to review the set of **desired properties** described earlier in section 2.4 using newly clarified terminology, and clearly specify the **goals** of this dissertation.

Last section will then describe our first **example of solution candidate** - so called 3SG algorithm.

4.1 Theory

4.1.1 Basic Terminology

We will be using the notion of **action model**, formally defined as a double $\langle \mathcal{D}, \mathcal{P} \rangle$, where $\mathcal{D}$ is a *description of domain dynamics* and $\mathcal{P}$ is a *probability function* over $\mathcal{D}$. **Action learning** will then be understood as an algorithmic process of improving $\mathcal{D}$, while using $\mathcal{P}$ to do so (we will present more formal definition in subsection 4.1.5).

Description of domain dynamics $\mathcal{D}$ explains how execution of individual **actions** leads from one discrete **world state** to another. To describe a world state, we use the collection of **fluent literals**: a fluent literal is either an atom f or its negation $\neg f$. Every fluent represents one individual property of our domain. The set of all fluents of our domain is denoted by $\mathcal{F}$. Complement of a fluent f (denoted by $\overline{f}$) is defined to be $\neg f$, while complement of $\neg f$ (denoted by $\overline{\neg f}$) is f.

World state s is any set of fluents such that every fluent literal is contained in s either in *positive* (f) or *negative* ($\neg f$) form. We will denote the set of all possible world states $\mathcal{S}$. Formally:

$$\mathcal{S} = \{s \mid \forall f, \neg f \in \mathcal{F} : (f \in s \wedge \neg f \notin s) \vee (f \notin s \wedge \neg f \in s)\}$$

Action instance is any expression consisting of **action name** and (possibly empty) set of constant **parameters** reffering to objects of our domain. Typical example of **action instance** is "$move(block01, table)$" expression of Blocks World (example 2.1). The set of all action instances possible in our domain is called $\mathcal{A}$. Throughout this chapter, we will use action instances to express observed actions, and therefore refer to them simply as *"actions"*.

Having established the meaning of $\mathcal{F}, \mathcal{S}$ and $\mathcal{A}$, we can move on to formal definitions of domain dynamics description $\mathcal{D}$. Throughout this chapter, we will explain **three different structures** that can be used to **represent** $\mathcal{D}$ (one of them was presented in related literature, and other two are our own improvements). Then we will define a probability function $\mathcal{P}$ over each of those structures.

4.1.2 Representing $\mathcal{D}$

Transition Relation $\mathcal{TR}$

Transition relation ($\mathcal{TR}$) is probably the most typical way of describing domain dynamics. Term has been used for example in [Amir-Russel, 2003] or [Amir-Chang, 2008] and is similar to structures used for instance in [Eiter et al., 2005].

Definition 4 (Transition Relation)**.** Having the sets $\mathcal{F}, \mathcal{S}$ and $\mathcal{A}$ defined as above, **transition relation** is any relation $\mathcal{TR} \subseteq \mathcal{S} \times \mathcal{A} \times \mathcal{S}$.

Intuitive meaning of every $(s, a, s') \in \mathcal{TR}$ is that *"execution of action* ***a*** *in a world state* ***s*** *causes a world state* ***s'*** *to hold in the next time step"*.

Note that the space complexity of a transition relation $\mathcal{TR}$ is $O(|\mathcal{A}| \cdot |\mathcal{S}|^2)$ which is equal to $O(|\mathcal{A}| \cdot (2^{|\mathcal{F}|})^2)$.

Effect Relation $\mathcal{ER}$

It is evident, that representing the domain dynamics by transition relation $\mathcal{TR}$ is relatively robust in terms of space requirements. In order to allow for faster, tractable solutions, we would like to come up with more compact representation of domain dynamics.

First improvement of $\mathcal{TR}$ introduced in this work is called **effect relation** $\mathcal{ER}$. As you will see, its structure is quite similar to that of $\mathcal{TR}$:

Definition 5 (Effect Relation)**.** Having the sets $\mathcal{F}, \mathcal{S}$ and $\mathcal{A}$ defined as above, **effect relation** is any relation $\mathcal{ER} \subseteq \mathcal{S} \times \mathcal{A} \times \mathcal{F}$.

Here the meaning of each triple $(s, a, f) \in \mathcal{ER}$ is *"execution of action* $\boldsymbol{a}$ *in a world state* $\boldsymbol{s}$ *causes a fluent* $\boldsymbol{f}$ *to be true in the next time step"*. Let us now take a look at the relationship between $\mathcal{TR}$ and $\mathcal{ER}$ representations.

Transformation 1. *Any* $(s, a, f) \in \mathcal{ER}$ *can be expressed by transition relation* $\mathcal{TR}$*.*

We can simply translate (s, a, f) into a set $\{(s, a, s') \mid s' \in \mathcal{S} \wedge f \in s' \wedge \overline{f} \in s\}$. In other words, we translate (s, a, f) into a set of all (s, a, s') triples where f *began to hold* after a was executed.

Transformation 2. *Any* $(s, a, s') \in \mathcal{TR}$ *can be expressed by an effect relation* $\mathcal{ER}$*.*

To prove the existence of $\mathcal{ER}$ representing (s, a, s'), we need to consider two distinct cases:

$s \neq s'$: This means, that there exists some $f \in s'$ such that $\overline{f} \in s$. We just need to have (s, a, f) included in $\mathcal{ER}$ for every such f.

$$s \neq s' \Leftrightarrow \forall f : \overline{f} \in s \wedge f \in s' : \exists (s, a, f) \in \mathcal{ER}$$

$s = s'$: This means, that action a has no effect in world state s. This fact is expressed simply by the **absence of** (s, a, f) in $\mathcal{ER}$.

$$s = s' \Leftrightarrow \nexists (s, a, f) \in \mathcal{ER}$$

Using transformations 1 and 2 we can conclude, that the **expressive power** of both representations is equal. Notice however, that the **space complexity** of $\mathcal{ER}$ is only $O(|A| \cdot |F| \cdot 2^{|F|})$ which is lower than in previous case.

Effect Formula $\mathcal{EF}$

Let us now introduce the final structure (that we will actually use in our algorithms) called **effect formula** $\mathcal{EF}$. Unlike in previous two cases, this new representation is not structured as a relation. It is rather a finite set[1] of **propositional atoms** over a vocabulary $\mathcal{L}_{\mathcal{EF}} = \{a^f \mid a \in \mathcal{A} \wedge f \in \mathcal{F}\} \cup \{a^f_c \mid a \in \mathcal{A} \wedge f, c \in \mathcal{F}\}$. The intuitive meaning of atoms from $\mathcal{EF}$ follows:

- a^f: *"action* $\boldsymbol{a}$ *causes* $\boldsymbol{f}$*"*
- a^f_c: *"*$\boldsymbol{c}$ *must hold in order for* $\boldsymbol{a}$ *to cause* $\boldsymbol{f}$*"* (c is a condition of a^f)

For example $\mathcal{EF} = \{drink^{\neg thirsty}, drink^{sick}, drink^{sick}_{poisoned}\}$ means, that *"if we drink a glass of water, we will not be thirsty"* and that *"if the water was poisoned, we will get sick"*.

Let us now take a look at the expressiveness of such representation:

Transformation 3. *Any* a^f *or* $a^f_c \in \mathcal{EF}$ *can be expressed by an effect relation* $\mathcal{ER}$*.*

The a^f atom is represented by any $\mathcal{ER}$ that contains at least one (s, a, f). The a^f_c atom is represented by any $\mathcal{ER}$ where for every $(s, a, f) : c \in s$.

[1] Even though $\mathcal{EF}$ is formally defined as a *set*, it can be equivalently understood as a *conjunction* of all its elements.

Transformation 4. *Any* $(s, a, f) \in \mathcal{ER}$ *can be expressed by an effect formula* $\mathcal{EF}$*.*

Any (s, a, f) triple is expressed simply by the presence of a^f in $\mathcal{EF}$ together with the absence of all the a^f_c, such that $c \notin s$.

We have now shown, that the expressive power of $\mathcal{EF}$ is equal to that of both $\mathcal{ER}$ and $\mathcal{TR}$. Even without losing any expressiveness, this new candidate for representing domain dynamics is considerably more compact. Upper bound of its space complexity is only $O(|\mathcal{A}| \cdot (|\mathcal{F}| + |\mathcal{F}|^2))$, which is exponentially lower than in previous cases. This makes it a suitable candidate for our action learning algorithms.

4.1.3 Probability Function

Having defined various set-based structures for description of domain dynamics, we can now understand the process of action learning as **iterative modification** of $\mathcal{D}$ by **adding** and **deleting** some of its elements **based on the observations**.

Remember however, that we want our methods to be usable in **probabilistic domains** with the possibility of **action failures** and **noise**. This prevents us from deleting elements from $\mathcal{D}$ based on just one observation.

Example 4.1

Imagine for example that our $\mathcal{D}$ says, that *"if we move block b1 to a free block b2, it will be on top of it"*. Such knowledge can be represented by a simple effect formula $\mathcal{EF} = \{move(b1, b2)^{on(b1,b2)}, move(b1, b2)^{on(b1,b2)}_{free(b2)}\}$. Now imagine, that we try to move it, but our hand slips and nothing happens. If

this happens rarely, deleting corresponding elements from $\mathcal{EF}$ is not a good idea (because they are correct most of the time).

To decide which elements of $\mathcal{D}$ are correct most of the time, and which are mostly false (and therefore should be deleted), we need a **probability function** $\mathcal{P}$.

Regardless of how $\mathcal{D}$ is represented, function $\mathcal{P}$ assigns every element $i \in \mathcal{D}$ a real number $p \in [0, 1]$ representing its probability:

$$\mathcal{P}(i) = \frac{pos(i)}{pos(i) + neg(i)}$$

where $pos(i)$ is a number of positive examples and $neg(i)$ is a number of negative examples w.r.t. i.

The definition of $pos(i)$ and $neg(i)$ differs based on the representation of $\mathcal{D}$.

4.1.4 Positive and Negative Examples

Definition 6 (Training Set)**.** Let E be a set of triples (o, a, o') where $a \in \mathcal{A}$ and o, o' are *consistent* (not containing f and $\neg f$ at the same time) non-empty subsets of $\mathcal{F}$. Each (o, a, o') triple means, that in certain successive time steps $\langle t-1, t\rangle$ we made following observation:

1. $\forall f \in o : f$ was true in time $t-1$,
2. $\forall f \in o' : f$ was true in time t,
3. a was an action executed in time t.

Set E is then called a **training set**, and its elements (o, a, o') are called **examples**. Note, that unlike world states, observations can be **incomplete**.

Let us now define a set $\Delta(o, o') = \{f \mid \overline{f} \in o \wedge f \in o'\}$ of all fluents that **changed their value** between times $t-1$ and t.

Intuitively, we say that (o, a, o') is a **positive example** w.r.t. $i \in \mathcal{D}$ if it **confirms** the meaning of i. Conversely, (o, a, o') is a **negative example** w.r.t. i, if it **denies** the meaning of i. Since the semantics of individual elements $i \in \mathcal{D}$ depends on our choice of representation, we need to separately define positive and negative examples for all three of $\mathcal{TR}, \mathcal{ER}$ and $\mathcal{EF}$.

Note 1. We say that $\mathcal{D}$ **covers** an example (o, a, o') only if (o, a, o') is *positive* for at least one $i \in \mathcal{D}$.

Recall, that $pos(i)$ and $neg(i)$ denote the **number of** positive and negative **examples** w.r.t. element $i \in \mathcal{D}$.

$\mathcal{TR}$:

In case of $\mathcal{D}$ represented by **transition relation** $\mathcal{TR}$, we define them in a following manner:

$$pos((s, a, s')) = |\{(o, a, o') \mid o \subseteq s \wedge o' \subseteq s'\}|$$

$$neg((s, a, s')) = |\{(o, a, o') \mid o \subseteq s \wedge \exists f \in o' : \overline{f} \in s'\}|$$

$\mathcal{ER}$:

If we represent $\mathcal{D}$ by **effect relation** $\mathcal{ER}$, we say that an example (o, a, o') is *positive* w.r.t. (s, a, f) only if we are sure that f changed its value from $\overline{f}$ after a was executed. *Negative* examples are those, where $\overline{f}$ still holds after the execution of a.

$$pos((s, a, f)) = |\{(o, a, o') \mid o \subseteq s \wedge f \in \Delta(o, o')\}|$$

$$neg((s,a,f)) = |\{(o,a,o') \mid o \subseteq s \wedge \overline{f} \in o'\}|$$

$\mathcal{EF}$:

When we use an **effect formula** $\mathcal{EF}$ to represent $\mathcal{D}$, we need to define positive and negative examples for both kinds of elements: a^f and a^f_c. Definitions for a^f elements are quite similar to previous case.

$$pos(a^f) = |\{(o,a,o') \mid f \in \Delta(o,o')\}|$$

$$neg(a^f) = |\{(o,a,o') \mid \overline{f} \in o'\}|$$

In compliance with the meaning of a^f_c, that treats c as a *condition* of a causing f, we define the *positive* examples as those, where c was observed when f started to hold. Conversely, any example where $\overline{c}$ was observed when a caused f, is considered *negative*.

$$pos(a^f_c) = |\{(o,a,o') \mid c \in o \wedge f \in \Delta(o,o')\}|$$

$$neg(a^f_c) = |\{(o,a,o') \mid \overline{c} \in o \wedge f \in \Delta(o,o')\}|$$

4.1.5 Action Learning

We have already mentioned, that action learning is an iterative process, where we add or delete some elements of $\mathcal{D}$.

In machine learning terms, this modification either **specifies** or **generalizes** our domain description $\mathcal{D}$, depending on the set of examples covered by it.

We say that $\mathcal{D}'$ is a **generalization** of $\mathcal{D}$ ($\mathcal{D}$ is a **specification** of $\mathcal{D}'$), if $\mathcal{D}'$ **covers** every example (o,a,o') covered by $\mathcal{D}$ and at least one more example. Using this terminology, we can define action learning in a following fashion:

Definition 7 (Action Learning)**. Action learning** is a process of iterative *specification* and *generalization* of $\mathcal{D}$ in order to:

1. **Maximize** the set of examples **covered** by $\mathcal{D}$.
2. **Minimize** the set of **negative** examples w.r.t. its elements.

Note 2. When dealing with probabilistic domains, we will need to maintain probability function $\mathcal{P}$ and use it throughout this process.

4.2 Goals of Dissertaion

4.2.1 Desired Properties

Before we enumerate our goals, we need to take one more look at the set of **desired properties** of action learning methods/algorithms from the section 2.4. Generally, will be interested in action learning algorithms that would satisfy following requirements:

- They are **tractable online** algorithms.
- They work in **probabilistic** domains with **action failures** and **noise**.
- Domains can be **partially-observable**.
- Produced action models support **conditional effects**.
- Produced results express not only action's **effects**, but also their **pre-conditions**.
- They can be used to probabilistically asses **current world state**.

Method name	Partially observable domains	Probabilistic action models	Probabilistic world states	Dealing with action failures	Both preconditions and effects	Conditional effects	Online
UNKNOWN	yes	yes	yes	yes	yes	yes	yes

Figure 4.1: Ideal solution candidate should have all the properties established in section 2.4.

We should explain the following collection of notions, in order to clarify aforementioned requirements:

Notion 1. We say that an algorithm is **tractable** if it runs fast enough for its solution to be practically useful [Hopcroft-Motwani-Ullman, 2007].

Notion 2. Algorithms whose input is served *one piece at a time*, and upon receiving it, they have to take an irreversible action without the knowledge of future inputs, are called **online** [Borodin-El-Yaniv, 1998]. In case of online action learning algorithms, this piece of input is the most recent observation. They run repeatedly, during agent's existence, while improving its action model[2].

Notion 3. Domain is called **probabilistic**, if the execution of the same action in the same world state can have different outcomes. Some of unexpected outcomes are referred to as **action failures** or **noise**.

Notion 4. Domain is **fully observable**, if $\forall(o, a, o') \in E$: both o and o' meet the definition of a *world state* (in that they are complete). Otherwise we say that domain is **partially-observable**.

[2]In contrast, offline action learning algorithms need the whole training set at their disposal. Therefore they are not suitable for gradual learning during agent's life.

Notion 5. We say that a model supports **conditional effects** if one action can have different effects when executed in different world states.

4.2.2 Specification of Goals

Now we can briefly summarize the goals of this dissertation project as follows:

1. Design the **collection of** action learning **methods** that would have all the **desired properties** formulated in section 2.4 (as depicted in figure 4.1).
2. **Compare** these methods and select the most promising candidates. We also need to use **competitive analysis** [Borodin-El-Yaniv, 1998] for the comparison of individual online algorithms.
3. Investigate their **compatibility** with commonly used **action languages** and **planners**.
4. Study their potential **alternative uses** (activity recognition, data mining, etc.).

4.3 3SG Algorithm as the First Solution Candidate

We will now present our first example of action-learning algorithm candidate called **3SG** (*Simultaneous Specification, Simplification and Generalization*). Input of 3SG consists of action model $\langle \mathcal{EF}, \mathcal{P} \rangle$ and current observation (o, a, o').

Note 3. We have chosen **effect formula** as the representation structure for domain dynamics because of its **compactness** and unrestricted **expressive power**. This allows us to present tractable algorithm capable of learning complex conditional effects.

In this section we will first formally define the output of 3SG, and then we will explain its pseudocode (presented in figure 4.3). This algorithm possesses at least 5 out of 7 desired properties described in the previous chapter (it hasn't been concluded whether remaining 2 goals are, or aren't satisfied yet).

4.3.1 Output of 3SG formally

3SG algorithm runs repeatedly **after every observation** (o, a, o') and modifies both *domain description* $\mathcal{EF}$ and *probabilistic function* $\mathcal{P}$ over elements of $\mathcal{EF}$ in a following way:

Theorem 1 (Output of 3SG algorithm)**.**

$$3SG(\langle \mathcal{EF}, \mathcal{P} \rangle)[(o, a, o')] = \langle (\mathcal{EF} \cup N_{GEN} \cup N_{SP}) \setminus R, P' \rangle$$

where N_{GEN} *are* ***new*** a^f *atoms (they* ***generalize*** $\mathcal{EF}$*),* N_{SP} *are* ***new*** a^f_c *atoms (they* ***specify*** $\mathcal{EF}$*), and* R *are elements that are considered* ***redundant*** *based on* $\mathcal{P}$*.*

As we can see, 3SG both **adds** and **deletes** certain elements from $\mathcal{EF}$. We delete those elements, that represent very improbable action effects, in order to **simplify** $\mathcal{EF}$ and keep the computational complexity as low as possible.

Definitions of N_{GEN}, N_{SP}, R, and P' follow:

$$N_{GEN} = \{a^f \mid f \in \Delta(o, o')\}$$

Here we assume that action a can be responsible for any change of fluent value that we have just observed.

$$N_{SP} = \{a_c^f \mid a^f \in \mathcal{EF} \wedge \overline{f} \in o' \wedge \overline{c} \in o\}$$

For any $a^f \in \mathcal{EF}$ inconsistent with current observation we assume that a^f effect has a condition c that is not met in o.

$$R = \{a^f, a_c^f \mid (\mathcal{P}(a^f) < min) \wedge (\forall a_c^f \in \mathcal{EF} : \mathcal{P}(a_c^f) < min)\}$$

Here min is a constant probability threshold. If a^f and all of its conditions[3] are improbable, we consider this group of elements redundant and delete them from $\mathcal{EF}$.

Let $pos(i)$ and $neg(i)$ denote the number of positive and negative examples so far w.r.t. $i \in \mathcal{EF}$. New probability function $\mathcal{P}'$ is then computed as

$$\mathcal{P}' = \{(i, \frac{pos'(i)}{pos'(i) + neg'(i)}) \mid i \in \mathcal{EF}\}$$

where $pos'(i) = pos(i) + 1$ iff (o, a, o') is a positive example, and $neg'(i) = neg(i) + 1$ iff (o, a, o') is a negative example w.r.t. i. Otherwise $pos'(i) = pos(i)$ and $neg'(i) = neg(i)$.

4.3.2 3SG Algorithm Description

Now that we have formally defined the output of 3SG, let us take a look at the algorithm itself (figure 4.3).

[3] Improbable effects a^f with low probability still provide some useful information if we have at least one probable a_c^f condition (we know, that even though a causes f rarely in general, it causes f often in situations where c holds).

We can see, that it runs in **polynomial** time in the size of the input observation. It cycles through observed fluents f and a^f elements of $\mathcal{EF}$, while distinguishing two cases:

1. If f is a **positive** example w.r.t. a^f, we increase $\mathcal{P}(a^f)$ and modify $\mathcal{P}(a^f_c)$ of all related a^f_c conditions. If there is no $a^f \in \mathcal{EF}$ that can be confirmed by f, we add it.

2. If f is a **negative** example w.r.t. a^f, we decrease $\mathcal{P}(a^f)$ and generate new conditions a^f_c such that $\overline{c} \in o$ (this represents our assumption, that a didn't cause f because there is some unknown condition for that effect, that was not met).
   Then we simplify $\mathcal{EF}$ by deleting all the elements corresponding to improbable effects with no probable conditions, as they don't provide any useful information (their presence in $\mathcal{EF}$ is a result of noise).

Using the optics established in 2.4, we can sum up the properties of our 3SG algorithm as depicted in figure 4.2.

Method name	Partially observable domains	Probabilistic action models	Probabilistic world states	Dealing with action failures	Both preconditions and effects	Conditional effects	Online
3SG	yes	yes	?	yes	?	yes	yes

Figure 4.2: First attempt to evaluate 3SG algorithm based on the properties defined in 2.4.

While it is too soon to conclude whether produced $\langle \mathcal{EF}, \mathcal{P} \rangle$ output of 3SG algorithm will be usable for probabilistic estimation of **current world state** or for reliable production of action **preconditions**, we can safely say, that

it can successfully deal with **action failures/noise** and **incomplete observations**. It also naturally produces **probabilistic action models** with **conditional effects**, and runs **online**.

4.3.3 Using the Output and Future Work

The output of 3SG algorithm is an **effect formula** $\mathcal{EF}$ with **probability function** over its elements. Next logical step in our future work concerning this solution candidate is the **translation** of $\mathcal{EF}$ **into** some of more widely used **action languages**. Then we will be able to use it in conjunction with available state-of-the-art planners.

Thanks to high expressive power of $\mathcal{EF}$ we will be able to formulate translations into wide variety of commonly used languages (some of which we already mentioned in section 2.2).

Note 4. It is important to clarify, that even though 3SG algorithm runs in polynomial time, we cannot guarantee this about actual translation algorithms.

Since 3SG is merely one of the solution candidates, we need to investigate its weaknesses, and consider alternatives. Implementation of individual solutions and their mutual comparison in various domains can also prove useful. Its attributes also need to be formally defined and proven, in order to establish theoretical background for our experiments.

Note 5. In order to complement our theoretical analysis, we may test the effectiveness and universality of our solutions by trying them in several domains with different properties and dynamics. According to [Buro-Furtak, 2004],

popular simulation games are ideal test applications for real–time AI research. We will use such games as a testbed for our algorithms instead of creating our own simulations.

Finally, we will need to study potential alternative uses of our action learning method. Even though its primary usage is allowing artificial agents (simulated or robotic) to learn and understand dynamics of their environment, we believe that, after certain modifications it can be used for different purposes. Let us for instance mention a discovery of typical behaviour of humans or so-called activity recognition, which has various applications within the area of intelligent house solutions, assistive technologies, or game AI development.

Input: Most recent example (o, a, o')

Input: Domain description represented by $\mathcal{EF}$

Input: Probability function $\mathcal{P}$

foreach $f \in \Delta(o, o')$ **do**

  `# Generalization of` $\mathcal{EF}$;

  if $a^f \notin \mathcal{EF}$ **then** add a^f to $\mathcal{EF}$;

  `# Modifying` $\mathcal{P}$ `after` f `changed value`;

  else if $a^f \in \mathcal{EF}$ **then**

    increase $\mathcal{P}(a^f)$ by incrementing $pos(a^f)$;

    foreach $c \in o$ **do**

      if $a^f_c \in \mathcal{EF}$ **then** increase $\mathcal{P}(a^f_c)$ by incrementing $pos(a^f_c)$;

      if $a^f_{\overline{c}} \in \mathcal{EF}$ **then** decrease $\mathcal{P}(a^f_{\overline{c}})$ by incrementing $neg(a^f_{\overline{c}})$;

    end

  end

end

foreach f *such that* $\overline{f} \in o'$ **do**

  if $a^f \in \mathcal{EF}$ **then**

    `# Modifying` $\mathcal{P}$ `after negative example`;

    decrease $\mathcal{P}(a^f)$ by incrementing $neg(a^f)$;

    `# Specification of` $\mathcal{EF}$;

    foreach c *such that* $\overline{c} \in o$ **do**

      if $a^f_c \notin \mathcal{EF}$ **then** add it;

    end

    `# Simplification of` $\mathcal{EF}$;

    if $\mathcal{P}(a^f) < min$ *and every* $a^f_c \in \mathcal{EF}$ *has* $\mathcal{P}(a^f_c) < min$ **then**

      delete a^f and every a^f_c from $\mathcal{EF}$;

    end

  end

end

Figure 4.3: Pseudocode of 3SG algorithm.

Bibliography

[Agrawal, 1994] R. Agrawal, R. Srikant. *Fast algorithms for mining association rules*. In Proceedings of the 20th International Conference on Very Large Data Bases (VLDB): 487-499. 1994.

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

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

[Amir-Chang, 2008] E. Amir, A. Chang. *Learning partially observable deterministic action models*. Journal of Artificial Intelligence Research, Volume 33 Issue 1: 349-402. 2008.

[Amir-Russel, 2003] E. Amir, S. Russel. *Logical Filtering*. In Proceedings of 18th International Joint Conference on Artificial Intelligence (IJCAI'03). 2003.

[Armijo, 1966] L. Armijo. *Minimization of functions having Lipschitz continuous first partial derivatives*. Pacific Journal of Mathematics, volume 16. 1966.

[Baioletti-Marcugini-Milani, 1998] M. Baioletti, S. Marcugini, A. Milani. *C-SATPlan: A SATPlan-based Tool for Planning with Constraints*. In Proceedings of AIPS-98 Workshop on Planning as Combinatorial Search. 1998.

[Balduccini, 2007] M. Balduccini. *Learning Action Descriptions with A-Prolog: Action Language $\mathcal{C}$*. In Proceedings of Logical Formalizations of Commonsense Reasoning, 2007 AAAI Spring Symposium. 2007.

[Bertsekas, 1999] D. P. Bertsekas. *Nonlinear Programming*. Athena Scientific. 1999.

[Blum-Furst, 1997] A. Blum, M. Furst. *Fast Planning Through Planning Graph Analysis*. Artificial Intelligence - Volume 90:281-300. 1997.

[Blum-Langford, 2000] A. Blum, J. Langford. *Probabilistic Planning in the Graphplan Framework*. Lecture Notes in Computer Science. Volume 1809. 2000.

[Borodin-El-Yaniv, 1998] A. Borodin, R. El-Yaniv. *Online Computation and Competitive Analysis*. Cambridge University Press, New York. 1998.

[Buro-Furtak, 2004] M. Buro, T. M. Furtak. *RTS games and real-time AI research*. In Proceedings of the Behavior Representation in Modeling and Simulation Conference (BRIMS). 2004.

[Chang-Lee, 1973] . C. Chang, R. C. T. Lee. *Symbolic Logic and Mechanical Theorem Proving*. Academic Press. 1973.

[Cook, 1971] S. A. Cook. *The complexity of theorem-proving procedures*. In proceedings of the third annual ACM symposium on Theory of computing (STOC '71). 1971.

[Čertický, 2008] M. Čertický. *An Architecture for Universal Knowledge-based Agent*. MEi:CogSci 2008 Proceedings. 2008.

[Čertický, 2009] M. Čertický. *An Architecture for Universal Knowledge-based Agent*. Master Thesis, Faculty of Mathematics and Physics, Comenius University, Bratislava. 2009.

[Čertický, 2010a] M. Čertický. *IK-STRIPS Formalism for Fluent-free Planning with Incomplete Knowledge*. Technical Reports, Comenius University, Bratislava, 2010.

[Čertický, 2010b] M. Čertický. *Fluent-free Action Representation with IK-STRIPS Planning Formalism*. Student Research Conference 2010 Proceedings. 2010.

[Davis-Putnam, 1960] M. Davis, H. Putnam. *A computing procedure for quantification theory*. Journal of the ACM, 7: 201-215. 1960.

[Eiter et al., 2000] T. Eiter, W. Faber, N. Leone, G. Pfeifer, A. Polleres. *Planning Under Incomplete Knowledge*. Lecture Notes in Computer Science, Volume 1861: 807-821. 2000.

[Eiter et al., 2005] T. Eiter, E. Erdem, M. Fink, and J. Senko. *Updating action domain descriptions*. Proc. 19th Int. Joint Conf. on Artificial Intelligence (IJCAI'05): 418–423. 2005.

[Farritor-Dubowsky, 2002] S. Farritor, S. Dubowsky. *A Genetic Planning Method and its Application to Planetary Exploration.* ASME Journal of Dynamic Systems, Measurement and Control, 124(4): 698-701. 2002.

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

[Fox-Long, 2003] M. Fox, D. Long. *PDLL2.1: An Extension to PDLL for Expressing Temporal Planning Domains.* Journal of Artificial Intelligence Research, Volume 20. 2003.

[Freund-Schapire, 1999] Y. Freund, R. Schapire. *Large margin classification using the perceptron algorithm.* Machine learning 37: 277-296. 1999.

[Gelfond-Lifschitz, 1988] M. Gelfond, V. Lifschitz. *The stable model semantics for logic programming.* In Proceedings of ICLP-88: 1070-1080. 1988.

[Gelfond-Lifschitz, 1991] M. Gelfond, V. Lifschitz. *Classical negation in logic programs and disjunctive databases.* New Generation Computing: 365-385. 1991.

[Guinchiglia-Lifschitz, 1998] E. Guinchiglia, V. Lifschitz. *An Action Language Based on Causal Explanation: Preliminary Report.* In Proceedings of 15th National Conference of Artificial Intelligence (AAAI'98). 1998.

[Gupta-Nau, 1992] N. Gupta, D. S. Nau. *On the Complexity of Blocks-World Planning.* Artificial Intelligence 56(2-3): 223-254. 1992.

[Hopcroft-Motwani-Ullman, 2007] J. E. Hopcroft, R. Motwani, J. D. Ullman. *Introduction to Automata Theory, Languages, and Computation.* Addison Wesley, Boston/San Francisco/New York: 368. 2007.

[Jensen, 1999] R. M. Jensen. *OBDD-based Universal Planning in Multi-Agent, Non-Deterministic Domains*. Master's Thesis, Technical University of Denmark Lyngby, DK-2800, Department of Automation. 1999.

[Jensen, 2003] R. M. Jensen. *Efficient BDD-Based Planning for Non-Deterministic, Fault-Tolerant, and Adversarial Domains*. School of Computer Science Carnegie Mellon University Pittsburgh, PA 15213. 2003.

[Jensen-Veloso, 2000] R. M. Jensen, M. M. Veloso. *OBDD-based deterministic planning using the UMOP planning framework*. In Proceedings of the AIPS-00 Workshop on Model-Theoretic Approaches to Planning: 26-31. 2000.

[Kambhampati, 2000] S. Kambhampati. *Planning Graph as a (Dynamic) CSP: Exploiting EBL, DDB and other CSP Search Techniques in Graphplan*. Journal of Artificial Intelligence Research. 2000.

[Kautz-Selman, 1992] H. A. Kautz, B. Selman. *Planning as Satisfiability*. In Proceedings of the Tenth European Conference on Artificial Intelligence (ECAI'92): 359-363. 1992.

[Kautz-Selman-Jiang, 1997] H. Kautz, B. Selman, Y. Jiang. *A general stochastic approach to solving problems with hard and soft constraints*. The Satisfiability Problem: Theory and Applications. 1997.

[Kutluhan-Hendler-Nau, 1994] E. Kutluhan, J. Hendler, D. S. Nau. *HTN planning: Complexity and expressivity*. In AAAI-94 Proceedings. 1994.

[Lee-Lifschitz, 2003] J. Lee, V. Lifschitz. *Describing Additive Fluents in Action Language C+*. In Proceedings of IJCAI-03. 2003.

[Leone et al., 2006] N. Leone, G. Pfeifer, W. Faber, T. Eiter, G. Gottlob, S. Perri, F. Scarcello. *The DLV system for knowledge representation and reasoning*. ACM Trans. Comput. Log. 7: 499–562. 2006.

[Lifschitz-Turner, 1999] V. Lifschitz, H. Turner. *Representing transition systems by logic programs*. In Proceedings of the 5th International Conference on Logic Programming and Non-monotonic Reasoning (LPNMR-99). 1999.

[Lopez-Bacchus, 2003] A. Lopez, F. Bacchus. *Generalizing GraphPlan by Formulating Planning as a CSP*. In Proceedings of the 18th International Joint Conference on Artificial Intelligence (IJCAI'03). 2003.

[Marquis, 2000] P. Marquis. *Consequence finding algorithms*. In D. Gabbay, P. Smets (Eds.), Handbook of Defeasible Reasoning and Uncertainty Management Systems, Volume 5: Algorithms for defeasible and uncertain reasoning. Kluwer. 2000.

[McDermott et al., 1998] D. McDermott, M. Ghallab, A. Howe, C. Knoblock, A. Ram, M. Veloso, D. Weld, D. Wilkins. *PDDL - The Planning Domain Definition Language*. Draft. Available at http://www.cs.yale.edu/ dvm. 1998.

[McMahan-Gordon, 2005] H. B. McMahan, G. J. Gordon. *Fast exactplanning in Markov decision processes*. In Proceedings of ICAPS. 2005.

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

[Moskewitz et al., 2001] M. W. Moskewitz, C. F. Madigan, Y. Zhao, L. Zhang, S. Malik. *Chaff: engineering an Efficient SAT Solver*. In Proceedings of the 38th Design Automation Conference (DAC'01). 2001.

[Mourao-Petrick-Steedman, 2008] K. Mourao, R. P. A. Petrick, M. Steedman. *Using kernel perceptrons to learn action effects for planning*. In Proceedings of the International Conference on Cognitive Systems (CogSys 2008). 2008.

[Mourao-Petrick-Steedman, 2010] K. Mourao, R. P. A. Petrick, M. Steedman. *Learning action effects in partially observable domains*. In Proceeding of the 2010 conference on ECAI 2010: 19th European Conference on Artificial Intelligence. 2010.

[Niemela-Simons-Syrjanen, 2000] I. Niemela, P. Simons, T. Syrjanen. *Smodels: A System for Answer Set Programming*. In Proceedings of the 8th International Workshop on Non-Monotonic Reasoning. 2000.

[Nilsson, 1980] N. J. Nilsson. *Principles of Artificial Intelligence*. Tioga, Palo Alto. 1980.

[Pasula-Zettlemoyer-Kaelbling, 2007] H. M. Pasula, L. S. Zettlemoyer, L. P. Kaelbling. *Learning Symbolic Models of Stochastic Domains*. Journal of Artificial Inteligence Research, Volume 29: 309-352. 2007.

[Pednault, 1989] E. P. D. Pednault. *ADL: exploring the middle ground between STRIPS and the situation calculus*. Proceedings of the first international conference on Principles of knowledge representation and reasoning. 1989.

[Pednault, 1994] E. P. D. Pednault. *ADL and the state-transition model of action*. Journal of Logic and Computation 4:467–512. 1994.

[Rintanen, 2009] J. Rintanen. *Planning and SAT*. Handbook of Satisfiability: 483-504, IOS Press. 2009.

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

[Russel-Norvig, 2003] S. Russel, P. Norvig. *Artificial Intelligence: A Modern Approach.* Prentice Hall. 2003.

[Sacredoti, 1975] E. D. Sacredoti. *The Nonliner Nature of Plans.* IJCAI'75 Proceedings of the 4th international joint conference on Artificial intelligence - Volume 1. 1975.

[Selman-Kautz-Cohen, 1993] B. Selman, H. Kautz, B. Cohen. *Local search strategies for satisfiability testing.* Cliques, Coloring, and Satisfiability: Second DIMACS Implementation Challenge 26. 1993.

[Slaney-Thiebaux, 2001] J. Slaney, S. Thiebaux. *Blocks World revisited.* Artificial Intelligence 125: 119-153. 2001.

[Subrahmanian-Zaniolo, 1995] V. S. Subrahmanian, C. Zaniolo. *Relating Stable Models and AI Planning Domains.* Logic Programming, Proceedings of 12th ICLP: 233-246. 1995.

[Tate, 1977] A. Tate. *Generating Project Networks.* IJCAI'77 Proceedings of the 5th international joint conference on Artificial intelligence - Volume 2. 1977.

[Theocharous-Kaelbling, 2003] G. Theocharous, L. Kaelbling. *Approximate planning in POMDPs with macro-actions.* In Proceedings of Advances in Neural Information Processing Systems 16. 2003.

[Wilkins, 1984] D. E. Wilkins. *Domain-Independent Planning Representation and Plan Generation*. Artificial Intelligence - Volume 22, Issue 3. 1984.

[Yang-Wu-Jiang, 2007] Q. Yang, K. Wu, Y. Jiang. *Learning action models from plan examples using weighted MAX-SAT*. Artificial Intelligence, Volume 171: 107-143. 2007.

[Zettelmoyer-Pasula-Kaelbling, 2003] L. S. Zettlemoyer, H. M. Pasula, L. P. Kaelbling. *Learning probabilistic relational planning rules*. MIT TR. 2003.