



UNIVERSIDADE DA CORUÑA
DEPARTAMENTO DE COMPUTACIÓN

A CORUÑA, JULY 2001

*Learning Non-monotonic Logic Programs
to Reason about Actions and Change*

PhD. Dissertation

David Lorenzo



Universidade da Coruña
Departamento de Computación

Learning non-monotonic Logic Programs to Reason about Actions and Change

David Lorenzo Blanco

Dissertation

presented at the *Facultade de Informática
de A Coruña* following the requirements
for the degree of

Doctor en Informática

July 2001



Universidade da Coruña
Departamento de Computación

PhD. thesis

**Learning non-monotonic Logic Programs
to Reason about Actions and Change**

David Lorenzo Blanco

PhD. advisor: Ramón P. Otero

Dissertation committee: Prof. Dr. Roque Marín Morales

Prof. Dr. Stephen Muggleton

Dr. Antonis C. Kakas

Prof. Dr. Pavel Brazdil

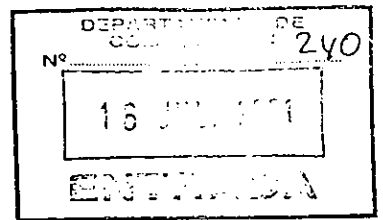
Prof. Dr. Antonio Bahamonde Rionda

Substitute members: Prof. Dr. Senén Barro Ameneiro

Dr. Alessandro Provetti



UNIVERSIDADE DA CORUÑA



DEPARTAMENTO DE COMPUTACION


Facultade de Informática
Campus de Elviña, s/n.
15071 A Coruña
Telf. 981 167 000
Fax 981 167 160

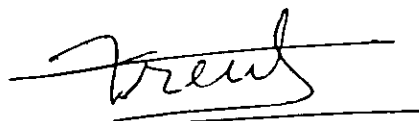
Ramón Pérez Otero, Profesor Titular del Area de Ciencias de la Computación e Inteligencia Artificial en la Universidad de A Coruña.

CERTIFICA

Que la memoria titulada "Learning non-monotonic Logic Programs to Reason about Actions and Change" ha sido realizada bajo mi dirección por *David Lorenzo Blanco* en el Departamento de Computación de la Universidad de A Coruña, y constituye la tesis que presenta para optar al grado de Doctor en Informática.

A Coruña, 16 de Julio de 2001


Ramón Pérez Otero
Director de la tesis


Vº Bº *Jose Luis Freire Nistal*
Director del Departamento de Computación

Acknowledgements

I want to thank my advisor Ramón P. Otero for his guidance and support during these years. To all the members of the AI Laboratory (AILab), since the former member, Charlie, to the most recent one, David E., for making the hours of work much more pleasant. Especially to Pedro Cabalar who was always there to provide helpful suggestions with the only help of his small notebook. I want to thank the members of my reading committee for reviewing this thesis. And, most importantly, to my family for their support and endless confidence during such a long period.

Part of this thesis was completed while I was granted by the *Xunta de Galicia*, that also allowed me to make a research stay of three months at the Department of Systems Theory and Information Engineering of the Johannes Kepler University of Linz (Austria), headed by Professor Franz Pichler. I also received support from the *Ministerio de Educación* under grant PB97-0228, the *Xunta de Galicia* under grants XUGA-10503B94 and XUGA-10503B96, and the University of A Coruña through travel grants who allowed me to attend to diverse conferences all over the world.

Abstract

Describing properties of actions and their effects on the state of the world has long been considered one of the central problems in the theory of knowledge representation [78]. Most of Artificial Intelligence applications require an almost complete description of an agent's actions and the *action laws* that describe the effects of actions that change the domain, which it is a time consuming task. Machine Learning methods can help to learn the domain specific knowledge necessary in the development of plans, or to construct the description of a robot's actions and its environment. The domain to learn is a system that changes its state when acted upon according to some set of unknown rules or functions. The learner initially knows nothing of the contexts in which actions produce changes in the environment, nor what those changes are likely to be. Then, it must infer how properties of the domain are affected by the execution of actions, or otherwise are subject to the general law of inertia.

In most previous works on action-model learning, the inferred model corresponds to a set of STRIPS-like operators [36], that was an early attempt to formalize descriptions of actions. Many extensions have been proposed since then, aiming for theories of actions that are more expressive, however, most of them are procedural representations. In this thesis we shall propose learning action models in non-monotonic formalisms for *Reasoning about Actions and Change* [78]. These formalisms constitute a formal and expressive representation for action domains, grounded on a mathematical and computational foundation, where system's behaviors are naturally viewed as appropriate logical consequences of the domain's description, so that the specification of actions and their effects is made as intuitive and natural as possible. Any attempt to learn in these formalisms must cope with various facets such as inertia (and the associated Frame Problem [81]), constraints and indirect effects (and the associated Qualification [79] and Ramification problem [58]), non-deterministic and other complex effects of actions.

Our work differs also from previous approaches in the use of *Inductive Logic Programming* [93] methods, which allow a natural integration with implementations of Action Theories based on the Logic Programming paradigm. The result is that an action theory is learned in the same way as it will be used, thus there is no a different representation for learning and another different for reasoning and planning. For this task we adopt Extended Logic Programs [45] as the form of programs to be learned, where two kinds of negation –negation as failure and classical negation– are effectively used in the presence of incomplete information.

The use of Action Languages [47] has a significant impact in the generality of the approach, so that diverse learning tasks can be approached in an homogeneous way and with minimal changes, for instance, the dual problem, i.e., *learning to act*, where action-selection rules for guiding the planning process are learned from solutions to instances of a planning problem. The control rules learned this way do not make explicit reference to the workings of the planner, but only refer to the solution space, so that they can be used by fundamentally different planning architectures.

Contents

Acknowledgments	i
Abstract	iii
Contents	v
1 Introduction	1
1.1 Objectives and Motivation	1
1.2 Organization of chapters	6
I Background	7
2 Non-monotonic Action Theories	9
2.1 Reasoning about actions and change	9
2.2 The Situation Calculus	10
2.3 Challenges for Action Theories	12
2.4 Logic Programs to reason about actions	14
3 Inductive Logic Programming	17
3.1 General definition of ILP	17
3.2 Testing the generality of hypotheses	19
3.3 Methods in ILP	20
3.4 Declarative bias	23
3.5 Recursive programs	25
3.6 Multiple predicate learning	27
3.7 Learning Logic programs with negation as failure	30
3.8 Learning Extended Logic Programs	31
II Learning Action Theories	33
4 Learning non-monotonic Action Theories	35
4.1 Learning action models	35
4.2 A basic definition	37
4.3 Learning Action Theories as Logic Programs	38
4.3.1 The Frame Problem	40
4.3.2 Formal definition	43

4.3.3	Undefinedness in the observations	45
4.3.4	Testing the generality of hypotheses	46
4.3.5	A three-valued setting for SC programs	49
4.4	Conclusions	51
5	Implementation: <i>LRAC</i>	53
5.1	Learning Action Theories with ILP methods	53
5.2	Description of <i>LRAC</i>	54
5.2.1	Learning the basic circuit with <i>LRAC</i>	61
5.2.2	The Blocks world	62
5.3	Conclusions	65
6	Learning Indirect Effects	67
6.1	The Ramification problem	67
6.2	Indirect effects in the Situation Calculus	68
6.3	Indirect effects = effects propagation	69
6.4	Learning Action Theories with indirect effects	71
6.4.1	A more complex circuit	74
6.4.2	Causality-based approaches	78
6.4.3	The Blocks world (contd.)	79
6.4.4	The Logistics domain	80
6.5	Relation to integrity constraints	83
6.6	Cycles	84
6.6.1	Yet another circuit	88
6.6.2	Recursive indirect effects	91
6.6.3	Negative cycles	94
6.7	Conclusions	97
7	Learning Default Action Theories	99
7.1	The Qualification Problem	99
7.2	Default theories	100
7.3	Default Action Theories	101
7.4	Learning rules with exceptions	103
7.4.1	A small example	104
7.5	Learning about exceptions	105
7.6	Conclusions	107
III	Extensions	109
8	Complex actions	111
8.1	Concurrent actions	111
8.1.1	Learning of concurrent actions	113
8.1.2	Scenarios for concurrent actions	115
8.2	Exogenous actions	120
8.3	Sensing actions	122
8.4	Actions with duration and delayed effects	123
8.5	Conclusions	125

9	Learning declarative control rules for planning	127
9.1	Introduction	127
9.2	Action-selection rules	129
9.3	Extraction of examples from plans	130
9.4	Learning action-selection rules in the Situation Calculus	131
9.4.1	Using a planner to generate the training set	133
9.4.2	Using learned control rules to speed-up planning	134
9.4.3	The Blocks world	135
9.4.4	The Logistics domain	140
9.5	Control rules based on subgoals	145
9.6	Discussion	149
9.7	Conclusions	152
IV	Related work and Conclusions	155
10	Related work	157
10.1	Learning operators effects for planning	157
10.2	Learning action models using ILP methods	161
10.3	Reinforcement Learning	162
10.4	Inference of deterministic finite-state automata	163
10.5	Learning Qualitative models	163
11	Summary and Conclusions	165
11.1	Summary	165
11.2	Areas for future work	166
A	Logic Programming concepts	169
B	The stable models semantics	171
	Bibliography	175

Chapter 1

Introduction

1.1 Objectives and Motivation

Describing properties of actions and their effects on the state of the world has long been considered one of the central problems in the theory of knowledge representation [47]. Actions performed in a world change its state, for instance, actions of a robot or an agent, updates in a database, program statements, operations in a supply chain, selections/clicks in an interactive GUI, and so on. In Artificial Intelligence (*AI*) the goal is to model and build *agents* capable of reasoning, planning and acting in a changing environment. To perform nontrivial reasoning or to take action in the world, an intelligent agent situated in a changing domain needs the knowledge of *causal laws* that describe the effects of actions that change the domain, i.e., it must be able to predict the results of its actions. For the most part, applications fall into three broad categories that correspond roughly to the need to predict, explain, and understand physical phenomena [117].

1. *predictive*, in which the objective is to predict future states of the system from observations of the past and present states of the system.
2. *diagnostic*, in which the objective is to infer what possible past states of the system might have led to the present state of the system.
3. *planning*, in which the objective is neither to predict the future nor explain the past but find a sequence of actions that lead the system to a desired state given an initial situation.

Most of these applications require an almost complete description of the agent's actions and its environment which, for moderately complex dynamic systems, it is a time consuming task. Machine Learning methods can help to learn the domain specific knowledge necessary, for instance, in the development of plans, or to construct the description of a robot's actions and its environment.

A *domain description* consists of a set of *fluent* names, a set of *action* names, and a set of causal laws that describe the effects of actions that change the domain. Fluents serve to describe situations and are properties whose truth values may change in the course of time as the result of performing actions. Let us consider a classical domain, the Blocks world.

Example 1 (Blocks world) *There is a table and some blocks, such that blocks can be moved onto other blocks or onto the table, which is large enough to hold all the blocks. For each block, either it is clear or else there is block sitting on it. The domain contains the action move and the fluents on and clear to represent the location of blocks and what blocks are clear respectively. □*

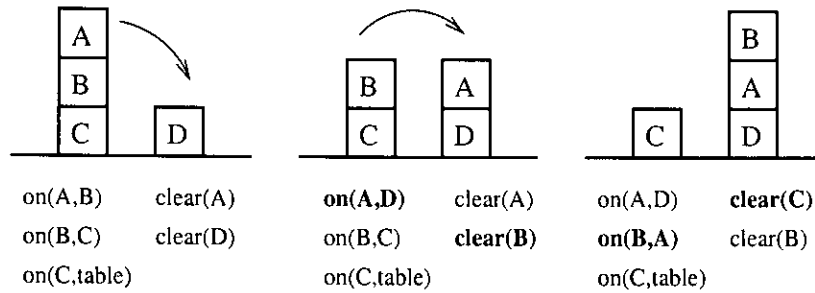


Figure 1.1: Blocks world

The domain to learn is a system that changes its state when acted upon according to some set of unknown rules or functions. The learner initially knows nothing of the contexts in which actions produce changes in the environment, nor what those changes are likely to be. An agent in such a scenario learns to predict the effects of his actions through observation, where inputs correspond to the actions executed and outputs correspond to the perceptual information available (Fig. 1.1), and constructs a dynamic system that can predict the future output of the unknown system, i.e., it must generate theories of how the actions affect the environment. In the blocks world, for instance, this means to determine the conditions under which blocks can be moved onto other blocks or onto the table, and predict the effects, i.e., the state of the blocks at the new situation after executing an action. While such predictions may not be completely accurate since they are based on incomplete experience, it is possible that they may be good enough to allow an agent to perform effectively [13]. It is important to notice that a model need not be exactly the same as the environment from the learner's point of view. In many cases, this is because the perception ability of the agent is limited.

To address this problem is a non-trivial task, taking into account the range of phenomena to accommodate, among others [111]:

- The causal laws relating actions to their effects.
- The conditions under which an action can be performed.
- Environments partially known to an agent.
- Exogenous and natural events.
- Complex actions.
- Discrete and continuous time.
- Unreliable sensors.
- Uncertainty.
- Non-Determinism.

Learning of Action Models [13] has been studied in a variety of disciplines including control theory, neural networks, and automata theory. The inferred model might correspond to a system of differential equations, a set of production rules [124], or a set of states and transition probabilities [13]. Most of previous approaches [87] were based on STRIPS [36] operators that

were an early attempt to formalize descriptions of actions, consisting of a list of state features (or postconditions) that are added and another list of features that are removed by applying the operator, without more “deductive” features. For instance, the description of the operator *move* in the blocks world is the following:

```
(move
  (params (<block> <block-from> <block-to>))
  (preconds
    (clear <block>)
    (clear <block-to>))
  (effects (
    (add (on <block> <block-to>))
    (del (on <block> <block-from>))
    (add (clear <block-from>))
    (del (clear <block-to>))))))
```

For years, researchers in the nonmonotonic reasoning community aimed for theories of actions that are more expressive than STRIPS-like systems. Many extensions have been proposed since then, aiming for theories of actions that are more expressive than STRIPS-like systems or to alleviate problems in operator representation caused by the STRIPS assumptions, however, most of them rely on procedural representations.

In this thesis we focus on learning representations of environments that can be characterized by non-monotonic formalisms for *Reasoning about Actions and Change*. These formalisms constitute an effort towards Commonsense Reasoning grounded on a mathematical and computational foundation. Features as indirect effects of actions (ramifications), implied action preconditions (qualifications), and concurrent actions are now easily represented. These formalisms are intended to precisely capture the effects that an *action* produces given the current description of the world and the *preconditions* that specify when the action can be executed. However, they are interested in temporal reasoning of a more general kind, e.g., explain observations about the state of the world in terms of what actions might have taken place, or what state the world might have been before –when the values of some fluents in one or more situations are given and the goal is to derive other facts about the values of fluents– to make plans and so on.

We focus on Logic Programming (LP) theories of actions, i.e., theories of actions that either use logic programming directly to formalize some aspects of reasoning about actions or those that provide translations to logic programs [8]. In particular we deal with a logical approach to modeling dynamical systems based on a dialect of first order logic called the *Situation Calculus* [81]. Unlike other formalisms, where observations describe the pre- and post-execution states of actions, i.e., transitions are considered as independent entities and the concept of situation is not mentioned explicitly, the Situation Calculus represents explicitly the situation where an action is executed, i.e., the sequence of actions that led to that situation. This particular feature is essential for learning in action domains, as we will see in the next chapters. For instance, the sequence of actions of Fig. 1.1 is represented as the following facts, among others, where s_0 is a constant that represents the initial situation:

$$\begin{aligned} & \text{holds}(\text{on}(a, d), \text{do}(\text{move}(a, d), s_0)) \\ & \text{holds}(\text{on}(b, a), \text{do}(\text{move}(b, a), \text{do}(\text{move}(a, d), s_0))) \end{aligned}$$

The use of Logic Programming makes it feasible to study the integration of Inductive Logic Programming (ILP) with logic programming theories of actions. Inductive Logic Programming,

i.e., Machine Learning methods that use Logic Programming as the representation knowledge— is an emerging field in the area of Machine Learning. The ability to use a Logic Programming representation both for examples and rules has contributed to this success. As a result, an action theory is learned in the same way as it will be used, thus there is no a different representation for learning and another different for reasoning, planning and so on. Additionally these logic-based methods for learning provide a high expressivity with respect to propositional methods, the possibility of including arbitrary logic programs as background knowledge, and other special techniques, e.g., noise handling techniques, while learning without losing representational power.

We can translate dynamic data into a Logic Programming representation that ILP methods can manage, where the temporal relationships among components and their states are partly expressed by the introduction of the variable *Time* in the training data. However, the use of first-order features to merely represent the sequence in a set of examples have restricted ability to model dynamic domains. For instance, non-monotonic reasoning is needed to avoid the explicit representation of unchanged properties from one particular situation to another (*Frame Problem* [81]). In the blocks world, after moving block *A* onto *D* in Fig. 1.1, we have that $on(C, table)$ is not affected by the action, so that approaches that do not consider the inertia assumption have to explicitly assert it in the input data. As a consequence, non-affected values of fluents must be part of the information given to the learning algorithm, thus having to deal with large datasets from which only a small fraction corresponds to effects of actions. Unlike this, with the inertia assumption, observations need only be explicitly given for those situations where a fluent changes, whereas inertia propagates non-affected truth values from one situation to the next one. Thus, the learner must be able to infer how properties of a domain are affected by the execution of actions, or otherwise are subject to the general law of inertia.

Furthermore, any attempt to learn in these formalisms must cope with other facets apart from inertia (and the associated Frame Problem), such as constraints and indirect effects (and the associated *Qualification* [79] and *Ramification* [58] problem), nondeterministic and other complex effects of actions. For instance, previous approaches to learning action models are restricted to predicting a single outcome or effect of an action. This forces the explicit representation of all the effects of an action as direct effects, producing the so-called Ramification problem, which makes the descriptions of actions cumbersome and difficult for complex domains. In many cases, the effects of an action are not caused directly by the execution of the action but indirectly through other changes. Thus, the learner should infer how properties of a domain are (directly/indirectly) affected by the execution of actions, or otherwise are subject to the general law of inertia. Previous approaches also dealt with effects of actions on a world where these are strictly specified. We show how to incorporate *defeasibility* into the specifications, where negation-as-failure is used to represent absence of information about exceptions. The explicit use of exceptions allows to learn rules that are more generally applicable, mainly when there are occasional qualifications or noise in the observations which may decrease the quality of the learning results, or when a theory must be specialized minimally.

On the other hand, we need to learn a definition for allowed actions, forbidden actions and actions with an unknown outcome, and therefore we need to learn in a richer three-valued setting. For this task we adopt Extended Logic Programs (ELP) as the form of programs to be learned, where two kinds of negation—negation as failure and classical negation— are effectively used in the presence of incomplete information. Let us consider an autonomous agent that has to select its own actions on the basis of acquired knowledge [63]. As pointed by Lamma et al., if the agent learns in a two-valued setting, it will not know the difference between what is true and what is unknown and, therefore, it can try actions with an unknown outcome. Rather, by learning in a

three-valued setting, the agent will know which part of the domain needs to be further explored and will not try actions with an unknown outcome unless it is trying to expand its knowledge.

The learning task studied in this thesis is an old problem in the Machine Learning field. We argue that this problem is best framed and solved by non-monotonic action theories where system's behaviors are naturally viewed as appropriate logical consequences of the domain's description, hence the specification of actions and their effects is made as intuitive and natural as possible [122]. Furthermore, a language that makes the notion of *situation* more central can provide a more convenient hypothesis space for learning, and a more compact description of observations, where observed time traces of property values are represented in the Situation Calculus in an homogeneous and natural way.

Our goal in this thesis is to provide solutions to several representative problems that can be characterized with these formalisms. While we admit that the real world cannot be modeled by such theories to any high degree of predictive accuracy, given appropriate percepts and actions, it makes sense to model system identification in terms of non-monotonic action theories. Actually, very simple scenarios become significantly complex without an appropriate level of abstraction. Even if it were possible to construct such a detailed model, this would be absurdly and uselessly large. Despite these simplifying assumptions, we believe that our models are relevant to a variety of interesting tasks and environments, for instance, those involving agents with higher level cognitive functions that involve reasoning about goals, actions, collaborative task execution, etc..

On the other hand, the use of Action Languages allows that diverse learning tasks can be approached in an homogeneous way, for instance, learning the effects of actions but also the dual problem, i.e., *learning to act*. In AI Planning, a planner is given an initial state and a goal, and finds a sequence of actions that maps the state into the goal. For instance, turning a configuration of blocks into another is a planning problem in the Blocks world (Fig. 1.2).

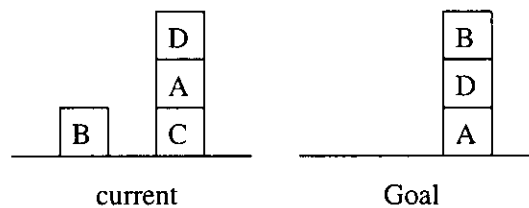


Figure 1.2: A planning problem in the Blocks world

Planners generally search through a list of domain actions until a correct sequence of those actions has been found that can achieve the desired goals. This problem has been tackled by a number of algorithms and in recent years substantial progress has been made [133]. However, the problem is still computationally hard and the best algorithms are bound to fail on certain classes of instances. An alternative that has been proposed is to use knowledge of the planning domain for guiding the planning process [113, 2, 51] via a set of additional constraints that do not make explicit reference to the workings of the planner, but only refer to the solution space like the constraints that define the original problem instance. Search control information for the blocks-world may say things like:

'pick up a *misplaced* block if clear'

'put current block on destination if destination block is clear and well placed'

etc.

The above control information for the blocks world is not included in the domain description, because it depends on a planning problem, e.g., *well_placed* is not used to predict the effects of moving blocks. Recent results have shown that it is possible to automatically acquire such high level declarative rules for action selection in planning in a purely declarative way. This problem has been very recently tackled by Kautz and Selman [52], Geffner [76] and Kharon [60], among others, using different schemes for representation and learning. More precisely, they deal with the problem of learning how to solve a planning problem in a domain, given solutions to a number of small instances of it. Learned rules select the action to be executed based on the current situation where the action will be executed and both the initial and the goal situation, so that control rules encode the solutions to a class of problems that differ on both the initial and the goal situations.

In the last part of the thesis, we consider the task of learning action-selection rules where both plans and the control rules are represented in the Situation Calculus.

1.2 Organization of chapters

This dissertation is organized as follows. In part I, we give a brief overview of formalisms for reasoning about actions and change (chapter 2), focusing on the Situation Calculus, as well as of Inductive Logic Programming methods (chapter 3). In part II we show what learning means in formalisms for reasoning about actions and change (chapter 4) and show that learning rules about actions and change may be possible using a Logic Programming implementation of the Situation Calculus and Inductive Logic Programming methods. A prototype to learn action theories in the Situation Calculus in the form of Extended Logic programs by using ILP methods is described in chapter 5. In chapter 6 we extend the framework of the previous chapter to deal with indirect effects of actions as well as with cyclical and recursive dependences, whereas in chapter 7 we learn *default theories* of actions, where rules are allowed to have exceptions. In part III, we propose some extensions to cope with complex effects of actions (chapter 8) and we adapt the previous framework for learning declarative control rules that help improving planning methods by reducing the search (chapter 9). In the last part, we review the most relevant related work in the field (chapter 10), present some conclusions and outline future work (chapter 11). Appendix A and B give a brief overview of basic Logic Programming concepts and the stable model semantics respectively.

Part I

Background



Chapter 2

Non-monotonic Action Theories

Action languages are formal models that are used for talking about the effects of actions [47]. These formal theories of actions are needed to describe dynamic behavior of programs, databases, robots, and other reasoning and actions agents, build agents capable of performing actions and reasoning in the dynamic world and reason about such agents. There are many existing disciplines that focus on modeling dynamical systems, including Petri nets, dynamic logic, temporal logic, finite automata, Markov decision processes, differential equations, STRIPS operators, etc, that solve problems that arise in sometimes narrowly circumscribed fields of specialization. However, as pointed by R. Reiter [111], there should be a unifying theory of dynamics, one that subsumes the many special purpose mechanisms that have been developed in these different disciplines, and that accommodates all the features of dynamical systems. In this chapter, we give a brief overview of formalisms for reasoning about actions and change, focusing on the Situation Calculus [81].

2.1 Reasoning about actions and change

One of the goals in reasoning about actions has been to contribute towards the development of ‘autonomous agents’ that can ‘perform’ in a dynamic environment. For this task the agents have to sense, reason, plan, and execute actions. For instance, the task might correspond to the agent being in one of many locations in an office building, in which locations correspond to junctions where hallways meet [11]. In this case, the observed outputs might correspond to the number of hallways incident on a junction, and the inputs to actions for traversing incident hallways. As another example, consider the structure of a voice-mail system. Here the states might correspond to various menus and services, actions to keys pressed by the user, and outputs to the announcements made at each state.

There are basic formalisms (referred to as STRIPS-like) –that was an early attempt to formalize descriptions of actions– where the knowledge T simply codes the possible answer pairs (action, effect) without more “deductive” features. For years, researchers in the nonmonotonic reasoning community aimed for theories of actions that are more expressive than STRIPS-like systems. Research in *non-monotonic Action Theories* constitutes an effort towards Common-sense Reasoning grounded on a strong mathematical foundation, where system’s behaviors are viewed as appropriate logical consequences of the domain’s description.

Most of these formalizations define an entailment relationship (\models) between the specifications (of effects of actions and relations among objects of the world). This ability allows to make plans that will take us to particular kind of worlds and explain observations about the state of

the world explain observations about the state of the world in terms of what actions might have taken place, or what state the world might have been before, or both [11]. Features, as indirect effects of actions (ramifications), implied action preconditions (qualifications), and concurrent actions are easily represented in non-monotonic action theories. Until recently, however, these theories were of theoretical interest only because of their high computational complexity. The situation has since changed substantially [67, 128, 101]. In fact, recent works have shown that a competitive planner can be built directly on top of one of such formalisms.

Formalisms for reasoning about actions and change are intended to precisely capture the effects that an *action* produces given the current description of the world and the *preconditions* that specify when the action can be executed. That is, they allow to do *temporal projections* to predict the state of the world after the execution of a sequence of actions. In a temporal projection problem, we are given a description of the initial state of the world, and use properties of actions to determine what the world will look like after a series of actions is performed. Current formalizations are interested in temporal reasoning of a more general kind, for instance, the cases when we want to use information about the current state of the world for answering questions about the past, i.e., when the values of some fluents in one or more situations are given, and the goal is to derive other facts about the values of fluents.

Recently there has been a lot of progress in formulating theories of actions, particularly in progressing from simple and/or restricted theories and ‘example centered approaches’, to general theories that incrementally consider various specification aspects [122]. This allows that more complex problems can be managed. For instance, *Cognitive Robotics* is an emerging field aimed at the construction of robots whose architecture is based on the idea of representing the world by sentences of formal logic and reasoning about it by manipulating those sentences [122, 11]. Action formalisms are also used for the diagnosis of dynamic systems [100], where the system behavior is described in an action formalism. Once diagnosis is defined in an action formalism, faults are viewed as any other evolving property of the domain, and the solution to the diagnosis problem provides an explanation for what components failed and when they did. Reiter [69] proposed specifying transactions in databases describing the evolution of a database under the effect of an arbitrary sequence of update transactions, like a dynamic system. Expressed in this way it is possible to pose and answer historical queries, to determine legal or illegal sequences of updates to the database and so on. Recently it has been shown that action formalisms are appropriate for Natural Language understanding due to the inferential and representational capabilities [102].

In this chapter, we give a brief overview of these formalisms, focusing specially on the Situation Calculus because it is the formalism we have selected for learning.

2.2 The Situation Calculus

The most classical formalisms for actions and change are the Situation Calculus and the Event Calculus. Situation Calculus [81] is a logical approach to modeling dynamical systems based on a dialect of first order logic. A formal framework to reason about actions and change requires the basic notion of a *situation*. Situation Calculus conceives of the world as consisting of a sequence of *situations*, each of which is a snapshot of the state of the world. Situations are generated from previous situations by *actions*. Any aspect of a system which can change as the result of an action or event is indexed by a situation and represented as a *fluent*. The next step is to represent how the world changes from one situation to the next. An action, if executed in some

situation, leads to a “resulting” situation. Situation Calculus uses the function

$$do(action, situation)$$

or $Result(action, situation)$ to denote the situation that results from performing an action in some situation. Fluents serve to describe situations and enable to reason about changes in the truth value of observations as the result of performing actions.

$$Holds(fluent, do(action, situation))$$

A situation argument may be in the form of a constant s_i or else a situation resulting after executing an action $do(action, sit)$. An special situation s_0 is included to represent the initial situation. A sequence of situations is encoded in the form of chains:

$$do(action_n, do(action_{n-1}, \dots, do(action_1, s) \dots))$$

starting from the initial situation. The concept of situation of the Situation Calculus differs from the concept of state used in automata-based approaches. An state S is a set of fluent literals, i.e., expressions of the form f or \bar{f} , such that each fluent occurs either affirmatively (f) or negatively (\bar{f}) in S . Thus, an state can be mapped to multiple situations.

A *narrative* is a course of actions about which we might have incomplete information. Formalisms for reasoning about actions may be divided into those which are narrative-based, such as the Event Calculus and those which reason on the level of sequences of actions, e.g., the Situation Calculus. In narrative formalisms, e.g., the Event Calculus [121], situations are usually represented by natural numbers, and a predicate $happens/2$ is added to indicate the actions that happened at each time point in the narrative. In this case, narratives correspond to a course of real actions.

$$\begin{aligned} &Holds(e, n) \\ &Happens(a_n, n), \dots, Happens(a_1, 1) \end{aligned}$$

Situation Calculus operates at a more abstract level than that of actual actions. It allows to reason about the hypothetical situation which results from the performance of a sequence of hypothetical actions [85]. Questions such as “is e an effect after the sequence of actions (a_1, a_2, \dots, a_n) ?” are expressed as formulas by means of the predicate $Holds(e, s)$ and the function $Result(a, s)$:

$$Holds(e, Result(a_n, (\dots (Result(a_2, (Result(a_1, s_0)))) \dots)))$$

Actions are described by stating their effects, that is, we specify the properties of the situation that results from doing the action. For instance, consider a robot r that can hold, drop and repair objects. The effect on the fluent *broken* of dropping and repairing something can be expressed in the form:

$$\begin{aligned} Holds(holding(r, x), s) \wedge fragile(x) &\Rightarrow Holds(broken(x), do(drop(r, x), s)) \\ Holds(hasglue(r), s) \wedge Holds(broken(x), s) &\Rightarrow \neg Holds(broken(x), do(repair(r, x), s)) \end{aligned}$$

where s, x, y and z are universally quantified variables, and *fragile* is an static property of objects. These axioms are called *effect axioms* or *action laws*, and capture the causal laws of a domain describing the changes in the values of fluents as a result of performing actions. In this example, the first axiom states that when the robot drops a fragile object x that he was holding, x will

be broken in the resulting situation, whereas the second axiom states that repairing a broken object x is possible provided the robot has glue.

Given the description of a domain and an initial situation s_0 , it is possible to solve *projection problems*, where we are interested in the value of a fluent at a particular situation, for instance:

$$\text{Holds}(\text{broken}(x), \text{do}(\text{repair}(r, x), \text{do}(\text{drop}(r, x), s_0)))?$$

In a *planning* problem, we are given the initial situation and a goal condition and have to find a sequence of actions that lead from the initial situation to a situation where the goal condition holds, for instance:

$$\text{Find } A \text{ and } B \text{ such that } \text{Holds}(\text{broken}(x), \text{do}(A, \text{do}(B, s_0)))$$

In an *explanation* problem, we are given one or more observations at one or more situations, for instance, $\neg\text{Holds}(\text{broken}(x), s_1)$, and the objective is to explain observations about the state of the world in terms of what actions might have taken place, or what state the world might have been before.

Situation Calculus has several problems that limit its applicability [48]. When there are multiple agents in the world or when the world can change spontaneously, or there are actions that have different durations, or whose effects depend on duration, then the Situation Calculus in its intended form cannot be used at all. Advantages and disadvantages of the Situation Calculus have been the subject of many debates in the knowledge representation community. It is clear that at least in simple cases –when actions are deterministic and are not executed concurrently– the language of the Situation Calculus is very attractive. Furthermore, several extensions have been added that allow to deal with ramifications, qualifications, concurrent actions, exogenous actions, durative actions and so on [104].

There are other formalisms to reason about actions apart from the Situation Calculus and the Event Calculus, among others, the family of \mathcal{A} languages [47], Causal Explanation [77], Temporal Action Logic [50] based on the concept of occlusion, Pertinence Action Language [101, 21] based on the concept of pertinence [101], the GOLOG language [65], and so on. Some of them are based on the Situation Calculus [67, 65], however most of them are narrative formalisms and use circumscription [80] or completion as the non-monotonic inference method. Many of them can be implemented in Logic Programming, however this is classically used for the Situation Calculus and the Event Calculus.

2.3 Challenges for Action Theories

The use of logic-based formalisms raises specific challenges for action theories, of which the most famous are, in historical order, the Frame Problem, the Qualification Problem, and the Ramification Problem. The *Frame Problem* [81] is the problem of how to express the facts about the effects of actions and other events in such a way that it is not necessary to explicitly state for every event, the fluents it does not affect. The dynamics of the world are specified by effect axioms which specify the effect of a given action on the truth value of a given fluent. As has been long recognized, axioms other than effect axioms are required for formalizing dynamic worlds. These are called *frame axioms* and they specify the fluents unaffected by the execution of an action. For instance, in the example of the previous section, we have the following frame axiom:

$$\begin{aligned} &\text{Holds}(\text{holding}(r, x), s) \wedge \neg\text{Holds}(\text{broken}(y), s) \wedge [y \neq x \vee \neg\text{fragile}(y)] \\ &\Rightarrow \neg\text{Holds}(\text{broken}(y), \text{do}(\text{drop}(r, x), s)) \end{aligned}$$

to mean for the conditions under which the fluent *broken* is not affected by the action *drop*.

Together, effect axioms and frame axioms provide a complete description of how the world evolves in response to an agent's actions. The problem associated with the need for frame axioms is that normally there will be a vast number of them. Normally only relatively few actions will affect the truth value of a given fluent. All other actions leave the fluent invariant, and will give rise to frame axioms, one for each such action. This is the *Frame problem*. When we specify action laws, we are only interested in describing changes, assuming that all the rest will remain unaffected. Solutions to the Frame problem involve the application of the inertia assumption for non-changing fluents.

On the other hand, actions often have other effects than those we are immediately inclined to put in the axioms concerned with the particular action. The *Ramification Problem* [58] is the problem of how to express the effects without forcing the explicit representation of all the effects of an action as direct effects. Let us consider the following example due to Thielscher [128].

Consider the action of toggling a switch, which in the first place causes nothing but a change of the switch's position. However, the switch is probably part of an electric circuit so that, say, some light bulb is turned off, as side effect, which in turn may cause someone to hurt himself in a suddenly darkened room by running against a chair that, as a consequence, falls into a television set whose implosion activates the fire alarm and so on.

The fact that the fire alarm becomes active is not caused directly by toggling a switch, however it started the sequence of events. As a consequence, there will be an effect axiom that connects the fire alarm and every possible action that eventually may cause the television to implode. This makes the descriptions of actions cumbersome and difficult for complex domains. With respect to the robot's example, if the object is a light bulb, the bulb will be definitely off when it is dropped. Then we need an effect axiom for the fluent *light_is_off* and for every action that eventually breaks the bulb, apart from dropping it. And however, the state of the light does not depend on the particular action that broke the bulb. Solutions to the Ramification Problem are based on the use of indirect effects, where *domain constraints*, i.e., general laws describing dependences between components of the world description, are used apart from the effect axioms.

The *Qualification Problem* [79] concerns how to express the preconditions for actions without having to account for the many conditions which, albeit being unlikely to occur, may prevent the successful execution of an action. McCarthy poses the following example: that it is necessary to have a ticket to fly on a commercial airplane is rather common to express. That it is necessary to be wearing clothes needs to be kept inexplicit unless it somehow comes up. The Qualification problem is still one of the least understood and with few satisfactory solutions. The main problem is that in general it is infeasible to represent complete specifications of all the preconditions to actions which would include all possible qualifications [30]. Solutions to the Qualification problem are based mainly on the use of *defeasible* specifications, so that if a new condition is observed where the action, for instance, $drop(r, x)$, is disqualified, that is not considered in the effect axiom, this can be retracted, in the spirit of non-monotonic reasoning, by adding new postulates.

Finally, one of the motivations for action formalisms is to find representations that are *elaboration tolerant* [79], i.e., representations that allow scenario descriptions to be modified and extended with additional information, to take into account new phenomena or changed circumstances, without extensive changes. In formalisms that do not fit with the elaboration tolerant condition, a single change in the description of the domain leads in many cases to a completely new representation.

2.4 Logic Programs to reason about actions

In this section, we focus on logic-based formalisms for *reasoning about actions and change*. By logic programming theories of actions we refer to those theories of actions that either use logic programming directly to formalize some aspects of reasoning about actions or those that provide translations to logic programs [46, 9]. Logic programming can be used to represent the effects of actions by importing the ontology of the Situation Calculus [121], representing effect axioms as logic program clauses and using negation-as-failure (NAF) as a means of overcoming the frame problem.

Most previous work on Logic Programming consider definite Horn programs. However, research work on Reasoning about actions has shown that such programs are not adequate to represent the effects of actions [46]. First formalizations of actions in LP were adequate for only the simplest kind of temporal reasoning, “temporal projection”, and only in the cases when the given description of the initial state is complete. The reason for that is that these formalizations use the semantics of logic programming which automatically apply the “closed world assumption” to each predicate. For these reasons, properties of actions are represented by means of Extended Logic Programs (*ELP*), i.e., logic programs that use both classical negation and negation as failure [46, 9]. Negative information is thus represented explicitly to allow the derivation of the negative value of fluents, whereas negation-as-failure is used for inertia.

Situation Calculus programs are logic programs with a fixed clausal structure. Formally we have:

Definition 2.1 (Situation Calculus Program) A Situation Calculus program is the conjunction of [121]:

- A finite set of general clauses

$$[\neg]Holds(f, s_0) \quad (2.1)$$

where s_0 denotes the initial situation.

- A finite set of clauses in the form

$$Holds(f, do(a, s)) \leftarrow \pi^+ \quad (2.2)$$

$$\neg Holds(f, do(a, s)) \leftarrow \pi^- \quad (2.3)$$

where π^+ and π^- does not mention the *Affects* predicate and every occurrence of the *Holds* predicate in π is of the form $[\neg]Holds(F', s)$. The description states that, in any situation, if the precondition holds then the effect will hold in the resulting situation. These axioms are called *effect axioms* or *action laws*.

- A finite set of *Affects* clauses of the form

$$Affects(a, f, s) \leftarrow \pi \quad (2.4)$$

where π does not mention the *Affects* predicate and every occurrence of the *Holds* predicate in π is of the form $[\neg]Holds(f', s)$.

- The universal frame axiom describes how the world stays the same (as opposed to how it changes).

$$Holds(f, do(a, s)) \leftarrow Holds(f, s) \wedge \text{not } Affects(a, f, s) \quad (2.5)$$

$$\neg Holds(f, do(a, s)) \leftarrow \neg Holds(f, s) \wedge \text{not } Affects(a, f, s) \quad (2.6)$$

□

This Logic Programming implementation of the Situation Calculus uses an special predicate *affects/3* and negation as failure to implement inertia. Inertia is tackled in the following way: a fluent may only change if a cause for the change can be derived from the theory. This ability to succinctly represent all the actions which leave a given fluent invariant is precisely the kind of solution to the frame problem we seek. The *Universal frame axiom* makes so-called frame axioms unnecessary. Thus, we can add as many effect axioms as we like and rely upon axioms (2.5) and (2.6) to implement inertia.

The description of the robot scenario in section 2.2 would be the following:

$$\begin{aligned} \text{Holds}(\text{broken}(x), \text{do}(\text{drop}(r, x), s)) &\leftarrow \text{Holds}(\text{holding}(r, x), s), \text{fragile}(x) \\ \neg \text{Holds}(\text{broken}(x), \text{do}(\text{repair}(r, x), s)) &\leftarrow \text{Holds}(\text{hasglue}(r), s), \text{Holds}(\text{broken}(x), s) \end{aligned}$$

$$\begin{aligned} \text{Affects}(\text{drop}(r, x), \text{broken}(x), s) &\leftarrow \text{Holds}(\text{holding}(r, x), s), \text{fragile}(x) \\ \text{Affects}(\text{repair}(r, x), \text{broken}(x), s) &\leftarrow \text{Holds}(\text{hasglue}(r), s), \text{Holds}(\text{broken}(x), s) \end{aligned}$$

Lifschitz [46] and Baral [11, 10] extend this LP formalization under the Answer Set semantics [44], to cope with indirect effects, exceptions, undefinedness and concurrent actions. Furthermore, several extensions have been developed based on the Situation Calculus to cope with exogenous actions, durative actions and continuous change [104, 86]. We will deal with some of these extensions in the next chapters.

Some formalizations of the Situation Calculus include a binary predicate *Poss*, apart from *do*, *holds* and *affects*, such that for any action *a* and any situation *s*, *Poss(a, s)* is true if *a* is possible (executable) in *s*. For instance, in the blocks world, we would write:

$$\text{Poss}(\text{move}(x, y), s) \supset \text{holds}(\text{clear}(x), s), \text{holds}(\text{clear}(y), s)$$

to represent that it is possible to move block *x* onto *y* provided both *x* and *y* are clear. Thus, *holds/3* clauses represent the *conditional effects* whereas *Poss* represents the *qualifications* to the action and acts as a domain constraint forbidding some next states. If an action produces different effects in different situations, clauses for *Poss/2* represent the union of the applicability conditions no matter what the effects are.

Other formalisms use predicates that are somewhat similar to *affects/3*. A dialect of the Situation Calculus due to F. Lin [67] uses a predicate *Caused* to capture simultaneously both the concept of causality and the truth value of the fluent, that constitutes a solution to both the Frame and the Ramification problems. Temporal Action Logic [50] uses a predicate *Occluded* to solve the Frame Problem, that expresses that a fluent has changed from false to true. Pertinence Action Language [101] uses a predicate *Pertinent* that like *affects/3* is not related to the truth value, but it represents a more general concept than simply change of value.

Chapter 3

Inductive Logic Programming

Inductive Logic Programming [93] (ILP) is a research area formed at the intersection of Machine Learning and Logic Programming. Unlike classical inductive learning [17], where only propositional learning systems are used, ILP uses Logic Programming as the representational mechanism for hypotheses and observations. By doing so, ILP can overcome the two main limitations of classical machine learning techniques:

- the use of a limited knowledge representation formalism (essentially a propositional logic)
- and the difficulties in using background knowledge in the learning process

The first limitation is important because many domains of expertise can be only expressed in a FOL, or a variant of first order logic. An ILP method can be used for propositional domains, however, the utility of ILP is most viewed when dealing with relational domains. As a counterpart, the task of ILP becomes significantly more complex as the space of solutions is extremely large. For this reason, additional mechanisms are added to the classic induction methods that restrict the space of solutions following the user preferences.

ILP can use so-called *Background Knowledge* (BK), consisting basically of predicate definitions to be used during learning [94, 106]. The fact that both facts, rules and the background are in the form of Logic Programs is an appealing feature of ILP systems, where for instance, the background can include domain-independent definitions, e.g., type information, patterns to indicate what kind of rules will be learned, and so on. This kind of information is not obvious how could be added to a propositional learning system.

ILP systems have been applied successfully in a number of real-world domains. Presently successful applications areas for ILP systems include the learning of structure-activity rules for drug design, finite-element mesh analysis design rules, and so on. In this chapter we give an overview of ILP methods focusing specially on those features that are relevant for learning action models.

3.1 General definition of ILP

ILP systems develop predicate descriptions from examples and background knowledge (B). The examples $E = E^+ \cup E^-$, background knowledge B and final descriptions are all described as logic programs. E usually consists of ground unit clauses of a single target predicate. E can be separated into E^+ , ground unit definite clauses and E^- , ground unit headless Horn clauses.

The following example, dealing with natural numbers, might be represented using the following clauses:

$$\begin{aligned} E^+ &= \{ \text{even}(0), \text{even}(2), \dots \\ E^- &= \{ \leftarrow \text{even}(1), \leftarrow \text{even}(3), \dots \\ B &= \begin{cases} \text{odd}(1), \text{odd}(3), \dots \\ \text{succ}(0, 1), \text{succ}(1, 2), \text{succ}(2, 3) \dots \\ \text{zero}(0) \end{cases} \end{aligned}$$

As in propositional learning, the objective is to find a most general rule or set of rules that explain the positive examples and no negative examples. This means that Prolog after consulting program B will answer:

```
?- even(4).
no
?- even(3).
no
```

Note that, the answer is due to the application of negation-as-failure, because actually Prolog does not know anything about $\text{even}/2$. After learning a correct hypothesis, Prolog will answer:

```
?- even(4).
yes
?- even(3).
no
```

Formally we have:

Definition 3.1 (General problem specification of ILP) The general problem specification of ILP is, given background knowledge BK and examples E , find the simplest consistent hypothesis H such that:

$$\begin{aligned} \text{(Prior Necessity)} & \quad BK \not\models E^+ \\ \text{(Posterior Sufficiency)} & \quad BK \cup H \models E^+ \\ \text{(Prior Satisfiability)} & \quad BK \cup E^- \not\models \perp \\ \text{(Posterior Satisfiability)} & \quad BK \cup H \cup E^- \not\models \perp \end{aligned}$$

□

Prior Necessity states that the positive examples must not be derivable from only the background, otherwise, learning is not necessary. Prior Satisfiability states that negative examples do not cause contradiction with background, given that the background consists of predicate definitions whose validity is previously assumed and cannot be retracted. Posterior Sufficiency and Satisfiability state that the solution must be a hypothesis that is complete and consistent respectively.

The result of the learning process in the even/odd example for the predicate $\text{even}/2$ is a theory in the form:

$$\text{even}(A) \leftarrow \text{zero}(A) \tag{3.1}$$

$$\text{even}(A) \leftarrow \text{succ}(B, A), \text{odd}(B) \tag{3.2}$$

A crucial feature of most advanced ILP algorithms is the possibility of including arbitrary Prolog programs as background knowledge, and not only ground clauses as in the first methods [106, 94], e.g., by calling a Prolog interpreter to derive ground atoms from intensionally coded specifications of background predicates. Some methods like Progol [89], integrate the induction method into a Prolog interpreter. In the previous example, an intensional definition for the predicate `succ/2` can be provided:

$$\text{succ}(A, B) \leftarrow B \text{ is } A + 1.$$

3.2 Testing the generality of hypotheses

A central issue in the definition of any ILP method is the method for testing the coverage of a hypothesis, i.e., testing whether a hypothesis entails an example e . This can be done extensionally or intensionally.

Definition 3.2 (Extensional coverage) A hypothesis H under background B *extensionally* covers an example e ($H \succcurlyeq e$), iff there exists a clause $c \in H$ and a substitution θ such that

$$\begin{aligned} \text{head}(c) \theta &= e \\ \text{body}(c) \theta &\subseteq M(B) \end{aligned}$$

where $M(B)$ is the least Herbrand model of the Background Knowledge B . □

In the even/odd example, we have that the learned theory covers extensionally `even(2)` if there is an instantiation of the rules 3.1 or 3.2, for instance:

$$\text{even}(2) \leftarrow \text{succ}(1, 2), \text{odd}(1)$$

Extensional coverage uses θ -subsumption which is computationally more efficient than resolution but that it is known to suffer from several drawbacks:

- It has been proved that θ -subsumption is sound, e.g., $c \succcurlyeq d$ implies $c \rightarrow d$. The converse, however, does not hold for certain types of self-recursive clauses. For instance, the following clause:

$$\text{ancestor}(X, Y) \leftarrow \text{ancestor}(X, Z), \text{ancestor}(Z, Y)$$

does not subsume the clause:

$$\text{ancestor}(a, d) \leftarrow \text{ancestor}(a, b), \text{ancestor}(b, c), \text{ancestor}(c, d)$$

because there is no substitution θ that makes $\text{body}(c) \theta \subseteq M(B)$. However, in fact, the first clause is more general than the second one. Extensional coverage would need the fact `ancestor(b, d)`. This means, that extensional coverage is very dependent on the completeness of the training set, i.e., it needs that all intermediate facts to prove an example are provided.

- Given this extensional coverage test, we have to remove from the search space the clause $p(X) \leftarrow p(X)$ because it is always consistent and complete.

The problems come from the fact that clauses are learned extensionally but then the whole program is interpreted intensionally (i.e., it is run on a Prolog interpreter). Adopting an intensional evaluation of clauses, most of the problems of extensionality are automatically overcome, since a logic program is learned in the same way as it will be used.

Definition 3.3 (Intensional coverage) A hypothesis H under background B *intensionally* covers an example e iff there exists a clause $c \in H$ such that

$$B \cup \{c\} \models e$$

□

From a practical point of view, logical entailment of a fact e by a theory $T = B \cup \{c\}$ can be verified using a Prolog interpreter with knowledge base T and the query $?- e$. With recursive clauses a hypothesis may have to be executed several times to prove an example. For instance, in the example above, the clause for *ancestor/2* is executed once for *ancestor(b, d)* and once for *ancestor(a, d)*.

3.3 Methods in ILP

The theory of ILP is based on proof theory and model theory for the first order predicate calculus, where inductive hypothesis formation is characterized by techniques including [93]:

- inverse resolution
- relative least general generalization
- inverse entailment

First methods [92] approached the problem from the direction of resolution proof-theory, where inference rules based on inverse resolution are able to invert one deductive inference step. Resolution is a sound proof procedure and can be inverted to form a useful inductive inference procedure. Resolution is diagrammatically shown with a large v .

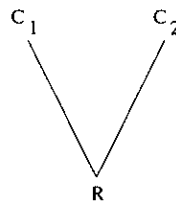


Figure 3.1: Resolution

There are two operators that carry out inverse resolution, *absorption* and *identification*, which are called V -operators. Each operator builds one of the two parent clauses given the other parent clause and the resolvent. More precisely, absorption constructs C_2 from C_1 and R , while identification constructs C_1 from C_2 and R . Let us consider the following example given in [82]:

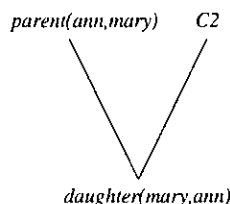


Figure 3.2: Inverse Resolution

We could learn the clause $C_2 = \text{daughter}(X, Y) \leftarrow \text{parent}(Y, X)$ which would be the most useful clause. Other solutions include:

$$\begin{aligned}
 \text{daughter}(\text{mary}, \text{ann}) &\leftarrow \text{parent}(\text{ann}, \text{mary}) \\
 \text{daughter}(X, \text{ann}) &\leftarrow \text{parent}(\text{ann}, X) \\
 \text{daughter}(\text{mary}, Y) &\leftarrow \text{parent}(Y, \text{mary}) \\
 \text{daughter}(\text{mary}, \text{ann}) &\leftarrow \text{parent}(Y, \text{mary}) \\
 \text{daughter}(\text{mary}, \text{ann}) &\leftarrow \text{parent}(\text{ann}, X) \\
 \text{daughter}(\text{mary}, \text{ann}) &\leftarrow \text{parent}(Y, X)
 \end{aligned}$$

In general there exists a vast range of possible answers of which only a few will be of any value, which introduces many sources of indeterminacy.

The *Golem* system [94] aims at overcoming the search explosion of its predecessors, due to the high non-determinism of the inverse resolution rules. Golem uses Plotkin's *relative least general generalization* [105] (*rlgg*). In the *rlgg* operator we are given background knowledge B and two examples e_1 and e_2 , and we want to find a clause C such that both $B \wedge C \models e_1$ and $B \wedge C \models e_2$ are true, i.e., a generalization of the clauses $e_1 \leftarrow B$ and $e_2 \leftarrow B$. To compute the *rlgg* of the two examples with respect to B , Golem takes the *lgg* of these clauses. For instance, for the even/odd domain, Golem will build from $\text{even}(8)$ and $\text{even}(12)$ the following clause:

$$\text{even}(a) \leftarrow \text{succ}(a, b), \text{succ}(c, a), \text{odd}(c), \text{succ}(d, c), \dots$$

In practice, this process is repeated for several examples until a more general hypothesis is obtained. The *rlgg* operator eliminates all indeterminacy of the search, though it suffers from several problems:

- It requires the background to be given extensionally.
- The two examples contribute to the generation of one clause, hence, they must be part of the same clause in the target concept. If they correspond to different clauses, then, in general, the clause produced by *rlgg* will be useless. To address this problem, Golem consider a large sample of pairs and the best of these is selected and generalized.
- In general this *rlgg* will be large. Some biases must be applied to the clause to reduce it.

The Golem system together with Foil, were the first systems to be considered efficient. In both, background must be provided as ground facts. The Foil system [106] is an extended version of the classical C4.5 [107] to deal with relational data, such that, the learning process remains the same but the specialization process is modified to learn Horn clauses. The process of specialization starts from the most general clause and adds literals to the clause –based on an heuristic measure– until a solution is found (Fig. 3.3).

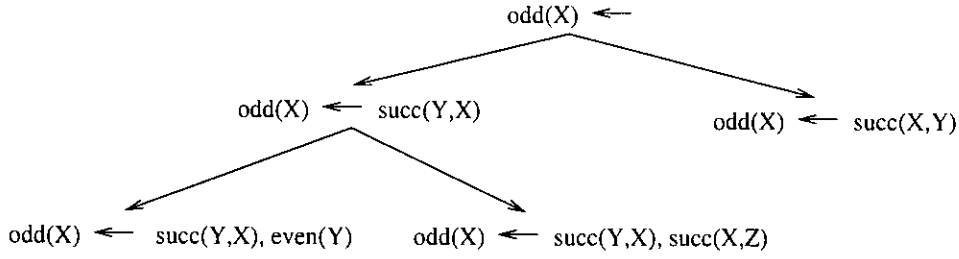


Figure 3.3: Foil's search space for the even/odd domain

Current methods consider a model-theory approach. In particular, Inverse Entailment (IE) is a generalization and enhancement of previous approaches, implemented in the Progol system [89]. Given background B , an example E and a hypothesis H satisfying $B \wedge H \models E$, we can invert the entailment relation such that:

$$B \wedge H \models E \Leftrightarrow B \models (H \rightarrow E) \quad (3.3)$$

$$\Leftrightarrow B \models (\neg E \rightarrow \neg H) \quad (3.4)$$

$$\Leftrightarrow B \wedge \neg E \models \neg H \quad (3.5)$$

Steps 3.3 and 3.5 are derived through the deduction theorem, whereas step 3.4 requires the contrapositive form of a clause. Put \perp as the conjunction of ground literals that are true in every model of $B \wedge \neg E$. Then we can write:

$$\neg \perp \models \neg H \quad (3.6)$$

$$H \models \perp \quad (3.7)$$

Such \perp is effective for reducing the hypotheses space, where a possible hypothesis H is constructed as a clause that subsumes \perp , thus, H is derived, in some sense, deductively. For instance, Progol [89] builds the following clause from $even(2)$ and the background:

$$\perp = even(a) \leftarrow succ(b, a), odd(b), succ(a, c), odd(c), \dots$$

which is subsumed by the clause 3.2 for $even/2$. In practice, Progol performs an A^* search in the space of hypotheses delimited by the most general clause and the bottom clause, until a consistent hypothesis is found that maximizes a compression measure.

Yamamoto [134] showed that IE is complete for relative subsumption but incomplete for entailment. The example Yamamoto used –where functions have been removed– is the following:

$$B = \left\{ \begin{array}{l} even(0) \leftarrow \\ even(y) \leftarrow s(x, y), odd(x) \end{array} \right.$$

$$E = odd(3) \leftarrow s(2, 3), s(1, 2), s(0, 1)$$

$$\perp = odd(z) \leftarrow even(0), s(y, z), s(x, y), s(0, x)$$

The correct hypothesis $H = odd(u) \leftarrow s(v, u), even(v)$ does not subsume \perp relative to B –thus, H cannot be derived by IE under B – and however $B \wedge H \models E$. In this case, the incompleteness of the background seems to be the cause, as we do not know $odd(1), odd(2), even(1), even(2)$.

In general, IE is incomplete with recursive clauses, where it needs to use the hypothesis more than once to prove the example. Yamamoto [134] and Furukawa [42] gave different conditions that guarantee the completeness of IE. Muggleton showed that enlarging the bottom set leads to completeness of IE [90], where the bottom clause is enlarged with $\{a \mid a \in HB(B \wedge \bar{E}) - M(B \wedge \bar{E})\}$ –where HB is the Herbrand base and M is the least Herbrand model– resulting in the following \perp -clause:

$$odd(z); odd(x); \dots; even(x); \dots \leftarrow even(0), s(y, z), s(x, y), s(0, x)$$

where $;$ represents disjunction, from which the intended clause can be derived. We can see that in order to prove $odd(z)$ the intended clause must be executed twice, once for $odd(x)$ and once for $odd(z)$.

In general, a notion of abduction is needed to cope with the problems caused by the incompleteness of the background knowledge [56]. More recently, a technique called Theory Completion using Inverse Entailment (TCIE)[91] based on IE developed by S. Muggleton uses a notion of abduction. Let us consider the following example of [89].

$$\begin{aligned} E^+ &= hasbeak(tweety) \leftarrow \\ E^- &= \leftarrow hasbeak(tweety) \\ B &= \begin{cases} hasbeak(X) \leftarrow bird(X) \\ bird(X) \leftarrow vulture(X) \end{cases} \\ \perp(B, E) &= \{ hasbeak(tweety), bird(tweety), vulture(tweety) \leftarrow \\ H_1 &= bird(tweety) \leftarrow \\ H_2 &= bird(X) \leftarrow \\ H_3 &= vulture(tweety) \leftarrow \\ H_4 &= vulture(X) \leftarrow \end{aligned}$$

In this case, H_1, H_2, H_3, H_4 are potential hypotheses that allow to prove E^+ and however, the predicate in E^+ does not coincide with the head of the hypotheses¹. TCIE is given multiple goal predicates and it uses an standard covering algorithm where each example is generalized using a multi-predicate search over all the predicates to find the hypothesis that covers the given example with maximal information compression. TCIE is capable of recovering accuracy to a substantial degree, even when large sections of the background are missing.

3.4 Declarative bias

The space of solutions when learning in first order predicate calculus has to be reduced by using some kind of bias. Most learning systems learn function-free Horn clauses and definite programs. Even so, the space of solutions is still intractable. In this section we review some biases used for current systems.

First of all, clauses containing functions, must be flattened before learning can be done. Flattening [116] is a technique that removes constants and functions from the arguments of clauses by introducing a $n + 1$ -ary predicate for each n -ary function ($n \geq 0$). For instance, the fact:

$$odd(succ(succ(succ(0))))$$

¹IE follows the Observation Predicate Learning (OPL) assumption in which both the examples and hypotheses define the same predicate.

can be flattened into:

$$\text{odd}(z) \leftarrow \text{succ}(y, z), \text{succ}(x, y), \text{succ}(0, x)$$

Many different methods exist for declarative bias, each relying on an special formalism. In these methods, the user can previously establish the form of the clauses to be searched, by selecting a predicate for the head and predicates that can appear in the body, and so on, adapting the search to the domain at hand. Language bias imposes certain syntactic restrictions on the form of clauses allowed in hypotheses. Recent results show that reducing the size of the target language often makes ILP learning more tractable. The main restrictions are on the introduction of existentially quantified variables in the bodies of clauses. Most important domain-independent biases are the following:

Definition 3.4 (Constrained clause) A clause is constrained if all variables in the body also appear in the head. \square

For instance, the following clause:

$$\text{notfree}(A) \leftarrow \text{on}(B, A)$$

stating that a block is not free when any other block is on it, would be non-constrained according to the definition.

Definition 3.5 (Range-restricted clause) A clause is non-generative or range-restricted if all variables in the head also appear in the body. \square

For instance: this classical definition used in many Prolog books is not range-restricted, as variable C is not referred to in the head.

$$\text{grandparent}(A, B) \leftarrow \text{parent}(A, C), \text{parent}(C, B).$$

Definition 3.6 (Determinate clause) A clause is determinate iff each of its literals is determinate; a literal is determinate if each of its variables that does not appear in the preceding literals has only one possible binding given the bindings of its variables that appear in preceding literals. \square

For instance, the relation *linked_to* in an undirected graph makes that the relation *connected* cannot be represented as a set of determinate Horn clauses, because several bindings are possible for C in *linked_to*(A, C).

$$\begin{aligned} \text{connected}(A, B) &\leftarrow \text{linked_to}(A, B) \\ \text{connected}(A, B) &\leftarrow \text{linked_to}(A, C), \text{connected}(C, B) \end{aligned}$$

This restriction is useful in ILP as it simplifies and speeds up the resolution process, however, it reduces the concepts that can be represented.

Definition 3.7 (Depth of a variable) The depth $d(v)$ of a variable v in a clause C is defined as follows:

$$d(v) = \begin{cases} 0 & \text{if } v \text{ is in the head of } C \\ (\min_{u \in U_v} d(u)) + 1 & \text{otherwise} \end{cases}$$

where U_v are the variables in atoms in the body of C that contains v . \square

In the following clause, variable z has depth 0, y has depth 1 and x has depth 2.

$$\text{odd}(z) \leftarrow \text{succ}(y, z), \text{succ}(x, y), \text{succ}(0, x)$$

Consider asking Prolog what lists 1 is a member of. Clearly there are an infinite number of answers, and Prolog cannot find all of them. The *member/2* predicate was not really written with those sorts of questions in mind, i.e., the first argument was meant to be the variable, not the second [18]. This sort of difficulty is prevented by the use of so-called *mode declarations* that specify input and output arguments of predicates.

Definition 3.8 (Input/Output variables) Input variables of a literal L_i ($i \leq n$) in the ordered Horn clause $A \leftarrow L_1, \dots, L_n$ are those appearing in L_i that also appear in the clause $A \leftarrow L_1, \dots, L_{i-1}$. All other variables are called output variables. \square

In mode declarations, + types are used where there is an input argument of a predicate, and - types are used for an output argument. For instance, in the *odd/even* domain, new variables are introduced through predicate *succ/2*.

$$\{\text{even}(+nat), \text{zero}(+nat), \text{succ}(-nat, +nat), \text{odd}(+nat)\}$$

Mode declarations in Prolog, combine input/output arguments for every predicate with limits on the number of alternative solutions (*recall*) for instantiating an atom. For a predicate such as *succ/2* the recall would be 1 since a number has at most one predecessor, whereas for a predicate such as *square_root/2* the recall would be 2 since a number has at most two square roots. An alternative approach to language restrictions related to the idea of mode and type declarations, is the use of *templates* that describe the form hypotheses must take. This approach is sometimes referred to as rule-models.

3.5 Recursive programs

Learning recursive programs is an important issue in ILP.

Definition 3.9 (Recursive clause) A clause is recursive if the predicate symbol in its head appears in any of the literals in its body. \square

A recursive program contains at least a base case (non-recursive) and one or more recursive clauses. For instance, the definition of *member/2* is of the form:

$$\text{member}(A, [A|B]) \tag{3.8}$$

$$\text{member}(A, [B|C]) \leftarrow \text{member}(A, C) \tag{3.9}$$

Unlike non-recursive programs, to check if a hypothesis covers an example e , the proof procedure actually needs other examples in its recursive execution apart from the background. For instance, when learning the predicate *member/2*, the proof of $\text{member}(3, [4, 1, 3])$ needs the fact $\text{member}(3, [1, 3])$ and $\text{member}(3, [3])$.

The common approach in learning recursive clauses is to include all the positive examples into the background knowledge. This allows to determine coverage (extensionally) by using these facts to unify with the recursive literals in the body of the clause. In this case, the recursive clause can be learned before the base case. However, this introduces several problems, that are

dealt in a variety of ways. Consider the following example given in [82], with a recursive clause $add(A, B, C) \leftarrow add(B, A, C)$ and the following three instances.

$$\begin{aligned} &add(1, 2, 3) \\ &add(1, 1, 2) \\ &add(2, 1, 3) \end{aligned}$$

If we estimate the coverage by adding the positive examples to the background, the clause will cover extensionally all the examples. However, to cover $add(1, 2, 3)$ we need $add(2, 1, 3)$ and viceversa, and to cover $add(1, 1, 2)$ we need again $add(1, 1, 2)$. For a more real coverage, we can keep track of the examples used to prove every example so that cycles are avoided [82]. In this case, the clause can only cover one of $add(1, 2, 3)$ or $add(2, 1, 3)$ but not both. By doing so, the clause covers just one example.

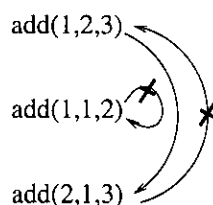


Figure 3.4: Computing coverage of recursive clauses

A further question concerns whether all *intermediate recursive calls* are included in the background, otherwise the extensional proof will fail. For instance, $member(3, [4, 1, 3])$ needs the fact $member(3, [1, 3])$. Thus, extensional evaluation will fail to find the solution when some important examples are missing.

With *intensional evaluation*, logical entailment is used to proof every positive example. In a Prolog interpreter, examples covered are determined by using a SLD resolution proof. A base case that depends strictly on the background must be learned before, otherwise, the recursive clauses will be shown to cover no examples, -e.g., the clause for $add/3$ - due to the non-finite recursion. If no control is added to avoid non-terminating programs, termination must be granted in some way, e.g., by setting a bound on the depth of the prover, such that false is returned for every query that overpasses the limit. Intensional evaluation makes learning less sensible to the incompleteness of the training set. For instance, with respect to the $member/2$ example, the proof of $member(3, [4, 1, 3])$ successes from the base case $member(3, [3])$ even without all intermediate facts.

One of the first systems, Foil, uses more sophisticated methods to assure recursive soundness. For instance, a recursive literal is added to the current hypothesis only if the program will terminate normally. This option requires an analysis of the constants in the domain.

$$3 \xrightarrow{succ} 2 \xrightarrow{succ} 1 \xrightarrow{succ} 0$$

$$[4, 1, 3] \xrightarrow{cons} [1, 3] \xrightarrow{cons} [3]$$

The program for $member/2$ is granted to terminate because every call to each other is done with a reduced argument, given that $cons/2$ imposes an ordering on the constants, so that no goal will be called twice². Something similar happens for the predicate $succ/2$.

²Predicate $cons/2$ is used to represent the list constructor $[Head|Tail]$

In general, recursion is a quite complex part of ILP and many approaches exist in the literature to manage it [20]. The interest of learning recursive clauses in the ILP literature is mostly illustrated with programs like *quicksort* (and other programs working with lists) where most popular algorithms have many difficulties. However, these algorithms have not developed special procedures for this task. It seems that most practical applications of ILP do not require complex recursive theories.

3.6 Multiple predicate learning

Definition 3.1 corresponds to so-called *single-predicate learning* (*spl*) where a single predicate is learned. In general, multiple predicates have to be learned, for instance, in the *even/odd* domain, we have considered that *odd/2* is provided in the background for learning *even/2*, however we could consider a learning task where a definition for both *even/2* and *odd/2* has to be learned.

Definition 3.10 (Multiple predicate learning) In a multiple predicate learning (MPL) problem, training data E contains examples for m predicates p_i such that E^+ and E^- are divided into m subsets $E_{p_i}^+$ and $E_{p_i}^-$. \square

Single predicate learning checks only that individual clauses are consistent (locally), not that the hypothesis as a whole is consistent with the negative examples (globally). That is, when learning a clause c for a predicate p_i , the evaluation is local to c instead of testing the coverage globally using $B \cup H \cup \{c\} \models e$, where H is the hypothesis built so far including clauses for every p_j with $j \neq i$. Global properties of hypotheses and clauses are defined in the context of the entire example set E and the hypothesis H whereas local properties are defined using E_{p_i} and H_{p_i} .

In general, learning multiple predicate definitions is equivalent to learning of recursive programs and they have similar problems. Classical approaches decompose a *mpl* problem into multiple single-predicate learning (*spl*) tasks using standard ILP methods. But, in general, it is not possible to solve a *mpl* problem by putting together the definitions learned for each predicate p_i obtained by a *spl* system [108]. By learning completely one predicate after another, the success of a *mpl* task depends on the order of the *spl* subtasks, because each *spl* task uses the clauses previously learned for other predicates as background.

Extensional evaluation avoids these problems because the examples in the training set are used as background instead of the clauses previously learned. However, the learned theory can be:

- Intensionally complete but extensionally incomplete.
- Extensionally complete but intensionally incomplete.
- Extensionally consistent but intensionally consistent.

Firstly, if we try to learn successively a definition for each p_i with a *spl* process, extensional evaluation means that when learning p_i , each predicate p_j , with $j \neq i$, has to be considered as a background predicate, i.e., p_j must have been determined extensionally by means of all its examples. Other clauses for p_j (i.e., those learned previously) are not used to try a derivation for p_i . Thus, extensional testing is very sensitive to the incompleteness in the background data, since each predicate to be learned forms part of the background for the other predicates, and the

available data for the target predicates may be incomplete (similarly to the example with the predicate `member/2`). Thus, the learned theory can be intensionally complete but extensionally incomplete.

Secondly, care must be taken to avoid non-terminating programs that are extensionally complete. For instance, the following program:

$$\begin{aligned} C_1 \quad \text{even}(A) &\leftarrow \text{zero}(A) \\ C_2 \quad \text{odd}(A) &\leftarrow \text{succ}(A, B), \text{even}(B) \\ C_3 \quad \text{even}(A) &\leftarrow \text{succ}(B, A), \text{odd}(B) \end{aligned}$$

does not terminate and however, every clause taken individually is locally complete and consistent. Martin and Vrain [75] present a way to deal with the drawbacks due to recursive or mutually recursive definitions that lead to infinite loops. They use extensional evaluation but distinguish between examples extensionally covered by a clause, and the examples included in the semantics of the program, and produce a theory that is complete and consistent. Situations where a theory is extensionally complete and however there are examples not proved, are dealt with by learning additional clauses –non-recursive– that extensionally cover them. In the theory above, new clauses are learned:

$$\begin{aligned} C_3 \quad \text{odd}(A) &\leftarrow \text{succ}(B, A), \text{zero}(B) \\ C_4 \quad \text{odd}(A) &\leftarrow \text{succ}(B, A), \text{succ}(C, B), \text{odd}(C) \end{aligned}$$

Thirdly, the problem of theories that are extensionally consistent but intensionally inconsistent, is intimately related to the incompleteness of the training set, i.e., when there are important examples that are missing in the training set. If the non-derivation of a negative example requires proving with negation as failure $\text{not } p(y)$, where p is a target predicate, any learned hypothesis together with the theory might entail facts for the target predicate that are not in the evidence, for instance $p(y)$, so that the whole theory may become globally inconsistent. Let us consider a very simple MPL task that consists of natural numbers, but the training set is limited to $\{1, 2, 3, 4, 5\}$ for the positive and negative examples.

$$\begin{array}{ll} + \quad p(1), p(3), q(1), q(3) & C_1 : p(x) \leftarrow q(x) \\ - \quad p(2), p(4), q(5) & C_2 : q(x) \leftarrow r(x) \\ B \quad r(1), r(2), r(3) & \end{array}$$

Induction is done with the order $\{p, q\}$. Learned rules C_1 and C_2 are valid taken individually, however, the whole theory entails $p(2)$, thus it is not a valid solution for the MPL task. The training set is incomplete wrt. to $q(2)$ –i.e., $q(2)$ is not in E^+ or E^- – and rule C_1 is consistent because $q(2)$ cannot be proved. The rule for q completes some examples, thus possibly affecting any previous rule that includes q as a condition. In this case, C_1 is inconsistent because $q(2)$ is intensionally covered.

When intensional evaluation is adopted, most of the above problems are not present, however, learning to succeed must be done in a particular ordering. A theory is built incrementally, starting from an empty theory (T_0), and adding a new clause at each step, thus we get a sequence of theories:

$$T_n \succ \dots \succ T_1 \succ T_0$$

such that $T_{i+1} = T_i \cup \{c\}$ for some clause c , where all theories are consistent, every T_{i+1} covers more examples than T_i and T_n covers the whole set of examples. The background used for the first *spl* task is the initial background, however this is subsequently enlarged with the rules learned. As a consequence, a *base case* (T_1) must be found that is constructed only from the initial background (e.g., *zero/1* or *succ/2*), i.e., no references are possible to *even* or *odd*. By doing so, learned programs are guaranteed to be globally complete. The underlying idea is that an example cannot be proved based on another example that has not been proved yet, to avoid cyclical dependences. However, it is necessary to discover the right order at which predicates should be learned. If the multiple *spl* tasks are ordered incorrectly, learning is still possible but the quality of the results can become drastically low. Once that ordering has been determined, the problem can be reduced to multiple *spl* tasks. Kakas et al. [55] adopts an hybrid of extensional and intensional evaluation, where examples are tested using the Prolog proof procedure, but examples for other predicates are also used as part of the background, to minimize ordering effects.

For mutually recursive theories, like in the *odd/even* example, there is no such ordering (Fig. 3.5), because the learning of one predicate must be interleaved with the learning of the other ones. A way to perform such interleaving is to start a multiple search –similarly to [91]– where multiple clauses are searched in parallel one for each predicate that has examples that remain to be covered [33, 108], such that the selection will be based on the compression measure achieved by each individual search. Thus, the ordering of the *spl* tasks is also learned.

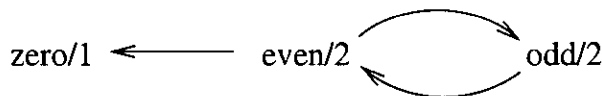


Figure 3.5: Dependency graph for *even/odd*

In this case, we need to learn in the following ordering (*even; odd; even*) where the *spl* task for *even/1* must be considered multiple times, resulting in the following theory:

$$\begin{aligned}
 C_1 : \text{even}(A) &\leftarrow \text{zero}(A) \\
 C_2 : \text{odd}(A) &\leftarrow \text{succ}(B, A), \text{even}(B) \\
 C_3 : \text{even}(A) &\leftarrow \text{succ}(B, A), \text{odd}(B)
 \end{aligned}$$

However, adding a locally consistent clause to the theory can make any previous clause globally inconsistent³. In the following example given in [33], C_2 becomes inconsistent when the second rule for p (C_3) is added to the theory, because $p(c)$ is now proved and hence $q(d)$.

$$\begin{array}{ll}
 + \ p(a), p(c), p(e), q(b) & C_1 : p(x) \leftarrow f(x) \\
 - \ q(d) & C_2 : q(y) \leftarrow p(z), s(z, y) \\
 B \ f(a), s(a, b), s(b, c), s(c, d), s(d, e) & C_3 : p(u) \leftarrow q(v), s(v, u)
 \end{array}$$

Avoiding global inconsistency requires an expensive retesting of examples, because the consistency of a clause must be checked always globally, i.e., using the negative examples for all the target predicates. The system MPL uses a *backtracking mechanism*, where already generated

³Note that the case of global inconsistency we showed with the Prolog algorithm, cannot appear now, given that the clause for q will be *always* learned before the clause for p .

clauses are re-evaluated and potentially deleted. However, when the theory includes mutual recursion we need to consider not all orderings (e.g., *odd even* and *even odd*) but all possible sequences (e.g., *odd even odd ...*) where each *spl* task may have to be considered multiple times, until all examples are covered. A possible solution is to *enlarge* the set of negative examples with those assumptions made for assuring the consistency of a rule [55], e.g., $q(2)$ in the small Progol domain. The basic idea is similar to the handling of negation in [16] with respect to negative examples: if the non-derivation of a negative example requires *not* proving $q(2)$, this is equivalent to require that $T \not\models q(2)$, i.e., $q(2)$ is a new negative example generated on-the-fly. This allows to check global consistency by testing only the assumptions generated from previous clauses and not all the negative examples. However, learning is sensitive to the ordering. To reduce this dependency, a *backtracking mechanism* is also used in [55] where the rules that caused the inconsistency are removed, i.e., those that generated the assumptions that have been violated.

In [98] a theory is revised given a set of positive and negative examples such that the theory is complete and consistent, based on unfolding as an specialization tool. This approach is also based on changing the theory to make it consistent. On the other hand, special techniques like the *layering method* [32] removes the inconsistency while keeping all positive examples covered. The insight is to avoid to re-open the question on the validity of clauses added in previous steps, and accommodating the previously acquired theory with the currently generated hypothesis. In the example above, by adding a new layer –with a new predicate p' – the new theory is consistent because the conflicting condition of C_2 has been renamed to p' , hence it is not affected by C_3 . An additional rule C_4 is added to complete p from p' .

$$\begin{aligned} C_1: p'(x) &\leftarrow f(x) \\ C_2: q(y) &\leftarrow p(z), s(z, y) \\ C_3: p(u) &\leftarrow q(v), s(v, u) \\ C_4: p(w) &\leftarrow p'(w) \end{aligned}$$

In general, the problem of learning multiple predicates is very costly with respect to *spl* problems. The interest of learning multiple predicates in the ILP literature is mostly illustrated with programs like family relations, where most popular algorithms have many difficulties. Similarly to the case of recursion, it seems that most practical applications of ILP do not require management of multiple predicates.

3.7 Learning Logic programs with negation as failure

Most ILP systems learn definite programs, where only positive atoms are allowed in the body of hypotheses. Negation-as-failure is introduced in AI to deal with lack of information and it introduces non-monotonicity into knowledge representation. Indeed, conclusions might not be solid because the rules leading to them may be defeasible [16]. Let us consider the following data:

$$\begin{aligned} + & p(1), p(3) \\ - & p(2), p(4) \\ B & q(1), q(3) \end{aligned}$$

In this case, we see that the candidate rule $p(X) \leftarrow q(X)$ is consistent because $q(2)$ and $q(4)$ cannot be derived from B . Note that the consistency of the rule is *defeasible* if some rules are learned for q .

It would be also possible to allow the use of negation in the body of learned rules, thus, it is indeed possible to learn logic programs with negation as failure. Let us consider the following data:

$$\begin{array}{l} + \quad p(1), p(3) \\ - \quad p(2), p(4) \\ B \quad q(2), q(4) \end{array}$$

In this case, the completeness of the candidate rule $p(a) \leftarrow \text{not } q(a)$ is defeasible, if additional rules are learned for q , i.e., some positive examples previously covered could remain uncovered after a rule for q is learned. In general, the problem is that inference is no longer monotonic and it can be the case that an example is inferred by a set of clauses but not by a superset of those clauses.

In [16], the assumptions made for the coverage of any learned clause are considered as additional positive or negative examples to avoid the problem. The basic idea is: if the derivation of a positive example requires proving with negation as failure *not P*, this is equivalent to require that $T \not\models P$, i.e., P is a negative example. Similarly if the non-derivation of a negative example requires proving with negation as failure *not P*, this is equivalent to require that $T \models P$, i.e., P is a positive example.

More recently, C. Sakama [118, 119] has studied the properties of inverse resolution and inverse entailment when learning normal logic programs under the stable model semantics.

3.8 Learning Extended Logic Programs

Most previous work on ILP consider definite Horn programs to be learned in a two-valued setting. In such a setting, what is not entailed by the learned theory is considered false, on the basis of the Closed World Assumption (CWA). The application of this CWA in the training set is not appropriate in many cases as by CWA all facts other than the positive examples are assumed to be non-instances of the target concept. By doing so, the role of negative examples is not clear because it is as if we supply a complete classification of all objects [25]. This CWA is applied in the Foil system. Even when an explicit set of negative examples is presented, a rule is learned for the positive value and any other object for which the rule does not apply, it is assumed to be a negative example. Thus, the learned theory returns an answer for the whole universe of objects (Fig. 3.6 a).

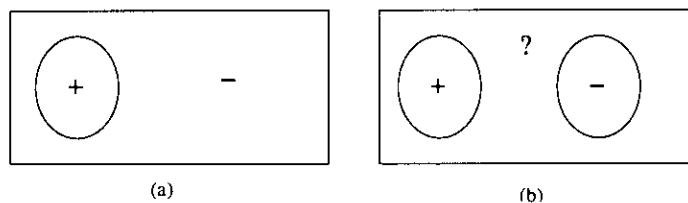


Figure 3.6: Three-valued learning

In the real world, we may not know whether some objects are positive or negative. Such undefinedness cannot be represented with normal logic programs. In a three-valued setting, a definition is learned for both the target concept and its opposite, considering positive and negative examples as instances of two disjoint classes. Explicit negation is used to represent the opposite concept, while default negation is used to handle exceptions to general rules (chapter 7).

Any other object not included in either positive or negative examples is considered *undefined* until the learned theory says that it must be or not be in the concept, so that the learned rules *do not cover the whole universe of objects* (Fig. 3.6 b). By doing so, the learned theory will then classify instances in three ways:

- Instances covered by the positive definition are positive.
- Instances covered by the negative definition are negative.
- Instances not covered by any definition are unknown.

This also introduces the possibility that positive and negative rules overlap in the observed part (Fig. 3.7 b) (that can be avoided by using exceptions) or even in the unobserved part of the universe (Fig. 3.7 a).

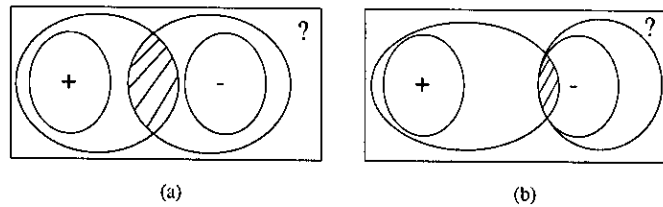


Figure 3.7: Possible overlaps in a three-valued setting

Extended Logic Programs allow to deal separately with classical and default negation, undefinedness in a predicate, contradictions and exceptions [63]. There exist several semantics for ELP, the most popular being the stable models semantics [44] and the well-founded semantics [43]. Several authors have proposed to learn Extended Logic programs. In [54, 63] a study of ILP under the answer sets semantics (system LELP) and well-founded with classical negation (system LIVE) respectively, is presented. They show the interest of these semantics and illustrate the approaches with classical domains used in the literature of non-monotonic logic programming. They both consider the possibility of inducing rules with exceptions.

Part II

Learning Action Theories



Chapter 4

Learning non-monotonic Action Theories

Formalisms for reasoning about actions and change are intended to precisely capture the effects that an *action* produces given the current description of the world and the *preconditions* that specify when the action can be executed. Most of these formalisms encode constraints and preconditions that can be verified given a state and determine if an action can be executed and the state that results from the execution of the action. That is, they allow to do *temporal projections* to predict the state of the world after the execution of a sequence of actions. As to learning, the problem is somewhat the contrary. The learner is provided with observed time traces of property values from an existing dynamic system –starting from an initial world state. Then, it must infer how properties of the domain are affected by the execution of actions, or otherwise are subject to the general law of inertia, thus enabling the derivation of system values that hold at future times based upon events that have happened at previous times.

In the next sections, we formally define the problem of learning to predict the effects of actions using Machine Learning methods that are oriented to classification problems and address the range of phenomena that take part in it.

4.1 Learning action models

The domain to learn is a system that changes its state when acted upon according to some set of unknown rules or functions. The learner initially knows nothing of the contexts in which actions produce changes in the environment, nor what those changes are likely to be. Contexts specify features of the world state that must be present for actions to apply, and effects specify how features of the context change in response to an action. We also require that the learned theory describes what changes in response to an action, not what stays the same.

An agent in such an environment learns to predict the effects of his actions through observation, where inputs correspond to the actions executed and outputs correspond to the perceptual information available. We assume that actions are separated from their consequences, i.e., an agent generates commands to the environment that have a nominal correspondence to real actions, but the corresponding action is not always taken. Thus, an agent can execute its actions in any environment regardless of which consequence the action might cause. For instance, a contraction of one's arm muscle can be executed regardless of whether the arm is free to move [124].

The problem of learning action models is a non-trivial task, taking into account the range of

phenomena to accommodate, among others [111]:

- Uncertainty

In some cases, the observer cannot determine the value of a fluent, e.g., because the perception ability of the agent is limited, or its value has to be sensed explicitly through a sensing action.

- Unreliable sensors.

In real scenarios, there can be noise in both the observations and actions.

- Non-Deterministic.

An environment is deterministic if the next state of the environment is completely determined by the current state and the actions selected by the agents.

- Exogenous and natural events.

In most cases, it is assumed that changes in the values of fluents can only be caused by execution of actions –i.e., the environment is only manipulated by the agent– otherwise the state of the system is assumed to be stable. If there are action sources other than the agent itself, an environment may seem active to an observer (*autonomous change*). However, in many cases, autonomous changes are quite common, for instance, if the domain has properties that are non-amenable to manipulation (weather conditions), a second agent is acting in the domain, and so on. In this case, the learner must distinguish the effects caused by his actions from those caused by other sources.

- Concurrent actions.

When actions are executed concurrently, an effect may depend on a particular *combination* of actions, an action may qualify another action's effects, effects can be cancelled and so on.

- Complex actions.

Most actions (e.g., picking up a block, going from one location to another) take time. Similarly, effect propagations usually incorporate very small delays. For all practical purposes these delays can be abstracted away and the effects assumed to be simultaneous and instantaneous. Abstracting small delays is often convenient because the resulting model is simpler or there is no actual knowledge on the delays for providing an accurate model. In some cases, however, the delays must be explicitly incorporated into the learning process.

- Discrete and continuous time.

In discrete scenarios, the agent's experience is divided into episodes that consist of the agent perceiving and then acting, and the environment is assumed to be static between episodes. Scenarios with continuous change involve continuously varying parameters.

- Environments partially known to an agent.

In most cases, it is assumed that the agent's sensors give it enough information to tell exactly which state it is in (i.e. the world is accessible). Then it can calculate exactly which state it will be in after any sequence of actions. In other cases, the environment is only partially known by the agent. An *accessible* environment is convenient because the

agent need not maintain any internal state to keep track of the world. This corresponds to the concept of episodic and non-episodic scenarios [117]. In *episodic* scenarios the quality of an action can depend just on the episode itself, because subsequent episodes do not depend on what actions occurred in previous episodes.

- Probabilistic effects.

In non-deterministic environments or with much noise, effects of actions can be assigned a probability [12], so that the learned theory tells when changes in the environment are associated with particular actions more or less often than we would expect by random chance.

In the remaining part of this chapter, we consider a basic framework where transitions are assumed to be instantaneous, deterministic and occur only when some agent executes a single action. In part III, we study some extensions that allow to deal with more complex phenomena.

4.2 A basic definition

The problem of learning to predict the effects of actions can be posed as a general *classification problem* where general rules are learned that classify objects into one of among several possible concepts. When learning the effects of an action a , instances are given for each *situation* where the action is executed and the concepts to be learned are: the action produced (resp. did not produce) the effect e in the resulting situation. Positive examples are observations of the shape (s, a, e) , where a is an action, e an effect and s the situation where a was executed, whereas negative examples are negations of observations, representing that action a did not cause the effect e . The separation of the evidence into causes and effects contains information about causality, thus the agent's experience is divided into episodes that consists of the agent perceiving and then acting. This separation is common to all action formalisms, however it may take a different form in each formalism.

We therefore consider a learning problem where we want to learn a theory H from a (possibly empty) initial theory T and from a set of positive and negative examples. The next definition follows the usual definition of induction on logic formalisms [93].

Definition 4.1 (Learning the effects of actions)

Given:

- A set E^+ of positive examples in the form (s, a, e) .
- A set E^- of negative examples in the form (s, a, e) .
- A (possibly empty) theory T .

Find:

- a theory H such that:

$$\text{(Prior Necessity)} \quad T \cup \{a\} \not\models e \quad \text{for } (s, a, e) \in E^+ \quad (4.1)$$

$$\text{(Prior Satisfiability)} \quad T \cup \{a\} \not\models \perp \quad (4.2)$$

$$\text{(Posterior Sufficiency)} \quad T \cup H \cup \{a\} \models e \quad \text{for } (s, a, e) \in E^+ \quad (4.3)$$

$$\text{(Posterior Satisfiability)} \quad T \cup H \cup \{a\} \not\models e \quad \text{for } (s, a, e) \in E^- \quad (4.4)$$

□

Prior Necessity states that the effect e must not be derivable from only the previous knowledge T . Prior Satisfiability states that action a can be added to T without causing contradiction. Posterior Sufficiency and Satisfiability state that the solution must be a hypothesis that is complete and consistent respectively, i.e., H derives e when a is executed but only for the positive examples. This definition is a general framework independent of any particular Action Language. In the next section, we provide a more detailed definition where action theories are represented as logic programs.

4.3 Learning Action Theories as Logic Programs

The use of Logic Programming makes it feasible to study the integration of Inductive Logic Programming (ILP) with logic-based formalisms for reasoning about actions and change [71, 73, 74]. The result is that an action theory is learned in the same way as it will be used, thus there is no a different representation for learning and another different for reasoning or planning. On the other hand, ILP provides a high expressivity with respect to propositional methods and the possibility of including arbitrary logic programs as background knowledge which is essential for managing, for instance, the Frame problem.

The form of the programs to be learned is that of the Situation Calculus. Situation Calculus is particularly well suited for the task of learning the effects of actions for several reasons. First of all, Logic programming can be used by importing the ontology of Situation Calculus, without need for a complex formalization. On the other hand, it has been proved sufficient for modeling a wide range of domains, and several extensions exist that deal with concurrent actions, continuous change, and so on. Furthermore, the form at which actions, effects and situations are represented in the Situation Calculus allows that multiple narratives starting from different initial situations, can be used for learning in an homogeneous and natural way. Let us consider the following domain.

Example 2 (Switches) *A simple circuit that includes a lamp and two switches Sw_1 and Sw_2 , is controlled by two actions $toggle(sw_1)$ and $toggle(sw_2)$.* □

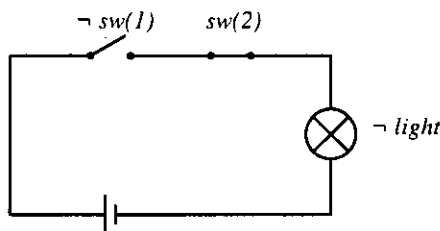


Figure 4.1: A simple circuit

This example concerns the representation of knowledge about the objects in a circuit, and how such knowledge is acquired. The circuit is a system that changes its state when acted upon according to some set of unknown rules or functions. The problem is to identify the effects of actions, describing how they change the state of the circuit when they are performed. Machine learning methods can help to construct the description of the circuit from the actions and the

effects produced by these, where measurements accumulated during simulations are the input to learning. We used a simple *wander* program that collects data about the actions while exploring the environment. With each exploratory step, the program records the action that was taken and the fluents that changed. We assume that the environment is only manipulated by one agent and there is no noise in the observations and actions.

Observations taken from the circuit must be converted into a LP form before learning. Examples are observations of the shape $holds(f_i, do(a_j, s))$ and $\neg holds(f_i, do(a_j, s'))$ that represent the value of a fluent f_i after executing an action a_j . Observations are given for every sequence of actions (or narrative) executed. Situations are represented through a sequence of actions starting from the initial situation and the effects are represented as *holds/2* facts of the form:

$$\begin{aligned} & holds(closed(sw_1), do(toggle(sw_1), s_0)) \\ & holds(active(light), do(toggle(sw_1), s_0)) \\ & \neg holds(closed(sw_2), do(toggle(sw_2), do(toggle(sw_1), s_0))) \\ & \neg holds(active(light), do(toggle(sw_2), do(toggle(sw_1), s_0))) \\ & holds(closed(sw_2), do(toggle(sw_2), do(toggle(sw_2), do(toggle(sw_1), s_0)))) \\ & holds(active(light), do(toggle(sw_2), do(toggle(sw_2), do(toggle(sw_1), s_0)))) \end{aligned}$$

Observations must include *holds/2* facts for the initial situation:

$$\begin{aligned} & \neg holds(closed(sw_1), s_0) \\ & holds(closed(sw_2), s_0) \\ & \neg holds(active(light), s_0) \end{aligned}$$

which represent the initial conditions of the domain. The constant s_0 makes explicit the starting situation at which narratives start.

In the Situation Calculus, *narratives* correspond to all linear sequences of actions starting from an initial situation (Fig. 4.2).

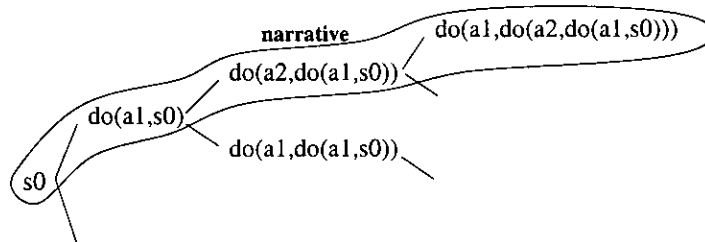


Figure 4.2: Narratives in the Situation Calculus

As a consequence, multiple narratives of events can be represented in a single model, hence examples form a tree rooted at the initial situation s_0 (Fig. 4.3).

This represents an advantage for Situation Calculus with respect to the Event Calculus [88] and in general to any narrative-based formalism. In the latter, a problem arises if we present several sequences of examples in the same session to the inductor, given that situations (represented as natural numbers) are common to all examples. Thus, if we present two or more examples with the same situation term, the inductor will consider the facts as belonging to the same situation. As a consequence, narrative formalisms need to represent every possible sequence of actions as

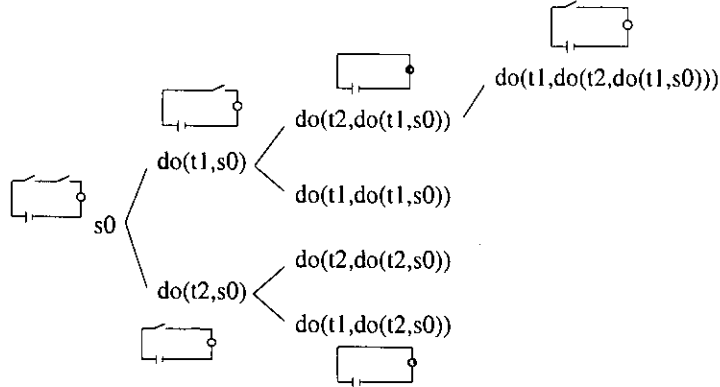


Figure 4.3: Tree of situations

independent models to make them jointly consistent. Furthermore, situations common to several narratives are to be stored multiple times.

$happens(toggle(sw_1), 1).$	$holds(closed(sw_1), 1).$
$happens(toggle(sw_2), 2).$	$holds(active(light), 1).$
$happens(toggle(sw_2), 3).$	$\neg holds(closed(sw_2), 2).$
	$\neg holds(active(light), 2).$
	$holds(closed(sw_2), 3).$
	$holds(active(light), 3).$

However, we still depend on a common initial situation, so that all narratives must begin in the same state (same values for all fluents). To avoid this, we allow different initial situations through a set of constants s_0^i . This is important, above all, in domains where some states are not reachable (or hard to reach) from some initial situations, or where actions are irreversible.

ILP algorithms deal with clauses with function symbols as though they were function-free by using *flattening* [116]. In this case, flattening introduces for the function symbol do of arity 2 a new predicate do_p of arity 3, where the first 2 arguments are the same as for the function, i.e., the previous situation and the action, and the last argument is the result of the function, i.e., the resulting situation. Thus:

$$Holds(f, do(a_n, do(\dots, do(a_1, s_0)) \dots))$$

is internally represented as:

$$Holds(f, s_n) \leftarrow do_p(a_n, s_{n-1}, s_n), \dots, do_p(a_1, s_0, s_1)$$

Multiple narratives can be represented in a unique model because the constants introduced for $do/2$ terms are generated from both the action and the previous situation, so that if $do_p(a, s, s')$ is true then $do_p(a', s, s')$ is false for every $a' \neq a$ and s' is the successor of s and not of any other s'' .

4.3.1 The Frame Problem

The use of logic-based formalisms raises specific challenges for action theories, of which the most famous is the *Frame Problem*. LP uses negation-as-failure (NAF) as a means of overcoming the

frame problem through the *Universal inertia axiom*. Inertia is tackled in the following way: a fluent may only change if a cause for the change can be derived from the theory.

On the other hand, observations used for learning are of the shape $holds(f_i, do(a_j, s))$ and $\neg holds(f_i, do(a_j, s))$, that represent the value of a fluent f_i after executing an action a_j . Thus, the Frame problem is *also present in the observations*, namely, if a fluent does not change after executing an action, its truth value must be explicitly asserted in the input data. As a consequence, non-affected values of fluents must be part of the information given to the learning algorithm, thus having to deal with large datasets from which only a small fraction corresponds to effects of actions. For instance, the observation:

$$holds(closed(sw_2), do(toggle(sw_1), s_0))$$

corresponds to the value of $closed(sw_2)$ after executing the action $toggle(sw_1)$, but it is not an effect of the action.

Furthermore, if these inertia values are part of the examples for a fluent f , we should learn a clause (for completeness) for each pair (a, f) , where action a does not change fluent f . These clause are called frame axioms:

$$Holds(f, do(a, s)) \leftarrow Holds(f, s), \pi^+ \quad (4.5)$$

$$\neg Holds(f, do(a, s)) \leftarrow \neg Holds(f, s), \pi^- \quad (4.6)$$

where π^+ and π^- represent all those conditions under which a does not affect f . For instance, some of the frame axioms for the negative value of $light$ in the basic circuit are:

$$\neg Holds(active(light), do(toggle(sw_1), s)) \leftarrow \neg Holds(active(light), s), \\ \neg Holds(closed(sw_2), s)$$

$$\neg Holds(active(light), do(toggle(sw_2), s)) \leftarrow \neg Holds(active(light), s), \\ \neg Holds(closed(sw_1), s)$$

$$\neg Holds(active(light), do(a, s)) \leftarrow \neg Holds(active(light), s), \\ a \neq toggle(sw_1), \\ a \neq toggle(sw_2)$$

expressing that the light remains off when one of the switches is toggled and the other is open, or when no switch is toggled. The frame axioms correspond to the negation of the conditions in the effect axioms. A system without the inertia principle would have an intractable, or even impossible, representation. Let us consider this example: assume a robot can sense whether it is in the corridor. If the inertia values must be covered by the learned rules, these must include an effect axiom for all actions (turning and navigational) that do *not* lead the robot out of the corridor, such that the robot is still on the corridor after the actions.

Furthermore, as the apprentice makes no difference between caused values and inertia values, it might find a consistent clause that covers both types of examples. These inertia examples might affect the learning process by altering the frequencies of literals in the training set specially because inertia values are usually much more numerous. For instance, let us consider an extreme situation in the simple circuit described in the previous section. If there is an additional fluent (not relevant for prediction tasks) that correlates exactly, in the given training set, with the positive value of $light$ (caused or not) when $toggle(sw_1)$ is executed, the apprentice will prefer a single rule that covers all examples, rather than the intended effect axiom and the frame axioms.

To avoid this situation, inertia values should be “separated” before learning and used as background knowledge. This is achieved with the inclusion of axioms 2.5 and 2.6 in the background, so that examples need only be explicitly given for those situations *where a fluent changes*, e.g., $holds(on(a, table), do(move(a, table), s_1))$, whereas the inertia axiom propagates non-affected truth values from one situation to the next one, completing every situation, e.g., $holds(on(b, c), do(move(a, table), s_1))$. Thus, we see how the use of the inertia axiom is a crucial feature that allows an explicit and clear distinction between the effects of actions and the inertia observations. An important consequence is that induction works only over *caused values*, whereas for the inertia values, the inertia axiom is returned as part of the learned theory.

To enable such distinction in the observations, we need to re-express the training set as ground facts for the predicate $affects/3$. To this end, positive $affects/3$ atoms are generated for every positive example and added to the background. Without these $affects/3$ atoms in the background, the inertia axiom produces inconsistency with the corresponding $holds/2$ atoms in the training set.

$$\begin{aligned} & affects(toggle(sw_1), closed(sw_1), s_0) \\ & affects(toggle(sw_1), active(light), s_0) \\ & affects(toggle(sw_2), closed(sw_2), do(toggle(sw_1), s_0)) \\ & affects(toggle(sw_2), active(light), do(toggle(sw_1), s_0)) \\ & affects(toggle(sw_2), closed(sw_2), do(toggle(sw_2), do(toggle(sw_1), s_0))) \\ & affects(toggle(sw_2), active(light), do(toggle(sw_2), do(toggle(sw_1), s_0))) \end{aligned}$$

As a consequence, a fluent cannot be caused to hold the same value it had in the previous situation, i.e., it is impossible to have *two consecutive positive (resp. negative) examples for a fluent*. For instance, an agent will not observe that a door is caused to be closed if an agent *try* the action of closing it when it was already closed. Thus, the learned theory will describe what changes in response to an action, not what stays the same.

The Situation Calculus provides a more compact and natural description of observations with respect to previous approaches. In the latter, each example in, e.g., the Blocks world, describes the properties of each block before and after the operation, together with the subject of the operation (i.e., the block to be moved) and its destination (Fig. 4.4), where the effects of the action are the difference between the pre- and post-execution states. Thus, situations are considered as independent entities not related to a particular narrative. By doing so, positive examples must be extracted from narratives so that the information provided by the narrative, i.e., the sequence of actions, is lost. A consequence is that they need the explicit representation of unchanged properties from one particular situation to another.

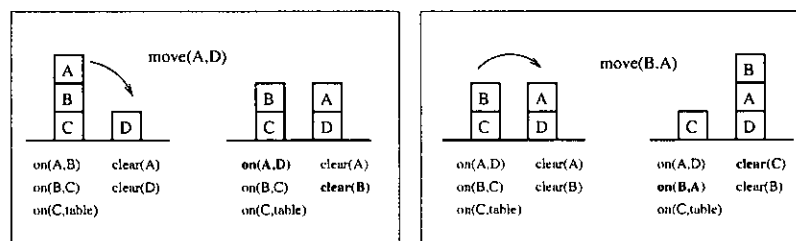


Figure 4.4: Datasets corresponding to Fig. 1.1

Note that situation $do(move(a, d), s_0)$ is the post-execution state of $move(a, d)$ but also the pre-execution state of $move(b, a)$. In the Situation Calculus positive examples are the individual effects of actions, for instance, the examples corresponding to Fig. 4.4 are, among others, the following:

$$\begin{aligned} & \text{holds}(on(a, d), do(move(a, d), s_0)) \\ & \text{holds}(clear(b), do(move(a, d), s_0)) \\ & \text{holds}(on(b, a), do(move(b, a), do(move(a, d), s_0))) \\ & \text{holds}(clear(c), do(move(b, a), do(move(a, d), s_0))) \\ & \dots \end{aligned}$$

where the narrative is explicitly represented in the examples through the situation constants, so that learning can be done directly from the narratives, which allows inference on the narratives to, for instance, implement inertia or complete missing values.

4.3.2 Formal definition

We follow the approach to learning Extended Logic Programs of [63, 54]. In the usual definition of learning Extended Logic Programs in static domains, conventional ILP methods are applied to learn a definition for both a target concept and its opposite, considering positive and negative examples as instances of two disjoint classes. Thus, a positive definition is learned from E^+ and E^- and a negative definition from E^- and E^+ . When learning the negative value, the roles of the example sets (E^+, E^-) are switched, so that E^- becomes the set of positive examples and E^+ becomes the set of negative examples. However, when learning action models, examples correspond to *transitions* rather than to fluent values at particular situations. Thus, positive examples correspond to the effects of actions whereas negative examples represent unsuccessful executions of actions, i.e., when learning the positive value of a fluent f_i , positive examples correspond to transitions of type b whereas negative examples correspond to transitions of type d , but also a to avoid that the positive and negative definitions overlap. Similarly for the negative value.

	Transition	
a)	true	→ false
b)	false	→ true
c)	true	→ true
d)	false	→ false

Table 4.1: Possible observations for learning

Every positive example consists of one *holds/2* atom that represents the truth value of a fluent, and one *affects/3* atom that represents “change of value”, so that negative examples contradict the *holds/2* atom, the *affects/3* atom, or both. In the ILP literature, E^- is reserved for the negative examples, however when learning Extended Logic Programs, it is used for the positive examples corresponding to the negative value of a fluent.

We therefore consider the following learning problem, where E contains examples for m fluents such that E is divided into m subsets E_{f_i} , where the learning procedure will be called twice for each fluent, once for the positive concept $E_{f_i}^+$ and once for the negative concept $E_{f_i}^-$.

Definition 4.2 (Learning Action Theories in the Situation Calculus)

Given

- A *domain description* consisting of two nonempty sets: a set \mathcal{F} of fluent names, and a set \mathcal{A} of action names.
- A set \mathcal{N} of narratives \mathcal{N}_k each starting at a situation s_0^k .
- A set $E^+ = \cup_{i=1}^n E_{f_i}^+$ of examples (ground facts) $holds(f_i, do(a_j, s))$, representing observations where a fluent $f_i \in \mathcal{F}$ became true after executing an action $a_j \in \mathcal{A}$ at a situation s .
- A set $E^- = \cup_{i=1}^m E_{f_i}^-$ of examples (ground facts) $holds(f_i, do(a_j, s))$ representing observations where a fluent $f_i \in \mathcal{F}$ became false after executing an action $a_j \in \mathcal{A}$ at a situation s .
- A set $E^{affects} = \cup_{i=1}^{n+m} E_{f_i}^{affects}$ of ground facts $affects(a_j, f_i, s)$ for every $e \in (E^+ \cup E^-)$ representing observations where an action $a_j \in \mathcal{A}$ *did* affect a fluent $f_i \in \mathcal{F}$ at a situation s .
- A set $E^{inertia} = \cup_{i=1}^{n'} E_{f_i}^{inertia}$ of ground facts $affects(a_j, f_i, s)$ representing observations where an action $a_j \in \mathcal{A}$ *did not* affect a fluent $f_i \in \mathcal{F}$ at a situation s .
- Background knowledge (BK), including $holds/2$ ground facts for all fluents at the initial situations s_0^k and the universal inertia axiom (eq. 2.5,2.6).

Find a Situation Calculus program $H^+ = \cup_{i=1}^{n_1} H_{f_i}^+$, and $H^- = \cup_{i=1}^{n_2} H_{f_i}^-$ composed of axioms in the form (2.2) and (2.3) respectively, such that:

$$(\forall e^+ \in E^+) \quad BK \cup H^+ \cup H^- \cup H^{affects} \models e^+ \quad (4.7)$$

$$(\forall e^- \in E^-) \quad BK \cup H^+ \cup H^- \cup H^{affects} \models \neg e^- \quad (4.8)$$

and respectively

$$(\forall e^- \in E^-) \quad BK \cup H^+ \cup H^- \cup H^{affects} \not\models e^- \quad (4.9)$$

$$(\forall e^+ \in E^+) \quad BK \cup H^+ \cup H^- \cup H^{affects} \not\models \neg e^+ \quad (4.10)$$

and

$$(\forall a \in E^{affects}) \quad BK \cup H^+ \cup H^- \cup H^{affects} \models a \quad (4.11)$$

$$(\forall a \in E^{inertia}) \quad BK \cup H^+ \cup H^- \cup H^{affects} \not\models a \quad (4.12)$$

where $H^{affects}$ are the corresponding *affects/3* axioms in the form (2.4), with the same body as the effect axioms in H^+ and H^- . \square

In order to satisfy the completeness requirement, the learned rules for the positive concept will entail all the examples for the positive value (eq. 4.7) and the corresponding *affects/3* literals, while the rule for the negative concept will entail all the (explicitly negated) examples (eq. 4.8) and the corresponding *affects/3* literals (eq. 4.11). The consistency requirement is satisfied when the learned rules both for the positive and negative concept do not entail examples of the complementary concept (eq. 4.9 and 4.10) or the inertia facts (eq. 4.12). This definition implicitly assumes that the training sets E^+ and E^- are disjoint. However, the definition does not rule out trivial solutions like $H = E^+$, neither capture the requirement that the learned hypothesis correctly predicts unseen examples. It should therefore be seen as a general framework [38], which needs to be further instantiated for the kinds of tasks addressed in practice.

When learning a fluent f_i , *holds/2* facts at situations $s \neq s_0$ where the fluent is affected by any action, become part of E^+ , whereas the background (B) includes the value of f_i at the initial situations s_0^i together with the *affects/3* atoms corresponding to every positive example that

disable inertia in the situations where f_i changed, hence $B \not\models E^+$ (because the corresponding *holds/2* literal is not in B) as required by the Prior Necessity condition in the ILP definition, but also $B \not\models \neg E^+$ (because inertia is disabled). That is, a positive example is never entailed by the inertia axiom. Note that if the *affects/3* atoms are not part of B , every situation included in the training set is equivalent to the initial situation (through the inertia axiom) before any clauses are learned.

With respect to negative examples, it is assumed that an action is not successful in the situations where a fluent did not change, so that, negative examples are added for every situation (included in the training set) where a fluent f is not affected by an action, thus effectively making a *CWA over causality*, but restricted to the narratives in the training set. This avoids that the learned theory infers a defined value for an inertia value (even with the same truth value), so that no new *affects/3* atoms are produced for the narratives in the training set, i.e., the extension for the predicate *affects/3* does not vary. Otherwise, the learned theory might produce a caused value for a fluent and a situation for which inertia was assumed initially, and produce non-monotonic effects during learning (section 4.3.4).

The result of learning consists of so-called *effect axioms* or *action laws* in the form:

$$\text{Holds}(f, \text{do}(a, s)) \leftarrow \pi^+ \quad (4.13)$$

$$\neg \text{Holds}(f, \text{do}(a, s)) \leftarrow \pi^- \quad (4.14)$$

stating that, in any situation, if the precondition π^+ (resp. π^-) holds, then the effect will hold (resp. not hold) in the resulting situation after executing an action a (f refers to each fluent used to describe properties of a domain). Effect axioms represent the executability conditions of actions and how those actions affect the value of the fluents when executed. When an action causes multiple effects, these are treated as direct effects of the action. The management of so-called indirect effects will be dealt in chapter 6. Effect axioms introduce a bias for the clauses to be learned, where only the previous situation can appear in the body of a clause. Furthermore, generalization over fluents is not considered, but only over their arguments.

4.3.3 Undefinedness in the observations

In real world, complete information is impossible to achieve and it is necessary to reason on the basis of the available information. The *incompleteness* of the observations is considered with respect to a narrative. A narrative is complete if all fluents have a defined value in any situation s' between the initial situation s_0 and the last situation s of the narrative, i.e., $s_0 \leq s' \leq s$. In some cases, a fluent is not defined in the initial situation, whereas in others a fluent can become undefined after executing an action. For instance, let us consider an scenario where an agent executes the action of pushing an object which causes the state of a small lamp to be occluded. Similarly, it could be that the device being sensed has an internal non-defined state. In others, it might become non-observable through an exogenous action (chapter 8).

However, due to the inertia axiom, the extension for any fluent is complete provided it has a defined value in s_0 , and thus, situations are completed where the fluent has an unknown value. In these cases, to allow *undefinedness* in the training set, we need to explicitly assert that a fluent is *affected* to disable inertia, i.e., a fluent becomes undefined after performing an action. When these extra *affects/3* atoms are included for a fluent f at a situation s , the inertia axiom is disabled, so that an undefined value is produced for f at s , because the corresponding literal *holds(f, s)* or *-holds(f, s)* is missing. Note that a defined value for f at that situation may be produced after some rules have been learned, thus producing non-monotonic effects during

learning. It would be possible to assume inertia for these cases, i.e., assuming that the last known value is also the current one, thus making a kind of *CWA over inertia*. However, we still need to allow that the learned theory produces a defined value for f at that situation that might be different from the one assumed.

In this three-valued setting, it would be even possible to learn a definition for such “caused” undefinedness (case e of table 4.2) in the value of a fluent, i.e., to learn that as an effect of executing an action a particular fluent becomes not observable. For this task, the learned theory should produce an undefined value for the fluent after executing the action under some preconditions. Actually, the theory should not produce any value at all, however it should infer that the fluent is affected to disable inertia, i.e., only the rule for *affects/3* is to be learned.

On the other hand, it would be possible that a fluent is undefined and it becomes defined later, e.g., the object is removed and the lamp is visible again, or a sensing action is executed. Thus, the set of possible *observations* for learning are displayed in table 4.2, where cases a and b represent positive examples respectively, and cases c and d represent inertia values.

	Transition	
$a)$	true	→ false
$b)$	false	→ true
$c)$	true	→ true
$d)$	false	→ false
$e)$	true/false	→ unknown
$f)$	unknown	→ true/false

Table 4.2: Possible observations in a three-valued setting

4.3.4 Testing the generality of hypotheses

A central issue in the definition of any ILP method is the method for testing the coverage of a hypothesis, i.e., testing whether an hypothesis entails an example. Action theories in the Situation Calculus are *recursive* logic programs, where an ordering is imposed in the situation constants so that termination is granted.

$$s_0 \prec do(a, s_0) \prec do(a', do(a, s_0)) \prec \dots$$

In fact, as generalization over fluents is not considered, programs in the Situation Calculus can be seen as recursive programs for a single predicate *holds/2*, or as programs with multiple predicates (fluents) whose definitions can be possibly interdependent. Indeed, some implementations of the Situation Calculus do not use a predicate *holds/2* explicitly, so that two notations are possible:

$$\text{holds}(\text{clear}(a), s_0) \text{ or } \text{clear}(a, s_0)$$

Thus, the problem of learning action theories constitutes in general a multiple-predicate learning (MPL) problem, where the consistency and completeness of hypotheses must be fulfilled by the set of clauses learned for every fluent taken as a whole theory in the context of the entire example set E , and not at a clause level for a particular fluent f_i and using E_{f_i} . Unlike most previous applications of ILP that do not require complex recursive theories, recursion is an important issue when learning action theories, where most fluents are so-called *dependent fluents* which are assumed to rely on the properties of the world around it, unlike independent

or *primitive* fluents. In fact, the complexity of the learning task is to find the *dependency graph* for the fluents.

With recursive programs, to check if the current hypothesis covers an example E , a proof procedure actually needs other examples, for instance, when a fluent depends on its own value in the previous situation. Let us consider the Blocks world.

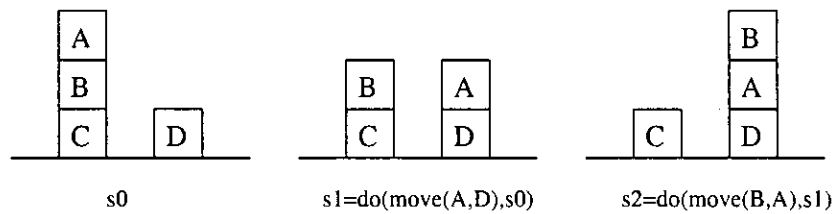


Figure 4.5: A narrative in the blocks world

Given the narrative in Fig. 4.5, to prove

$$E = \text{holds}(\text{clear}(c), \text{do}(\text{move}(b, a), \text{do}(\text{move}(a, d), s_0)))$$

we need the value for $\text{clear}(b)$ at the previous situation. In this case, fluent $\text{clear}/1$ is part of the background for learning $\text{clear}/1$. The common approach in learning such recursive programs is to include all the positive examples into the background B , so that an effect axiom covers extensionally the example, i.e., there exists a ground instance of the axiom $e \leftarrow l_1, \dots, l_n$ where each l_i belongs to $E \cup B$.

A drawback is that extensional evaluation cannot deal with certain kinds of *incompleteness* in the training set. If there are gaps in the narrative of Fig. 4.5, e.g., the fluent $\text{clear}/1$ has a defined value at s_0 and somewhat $\text{clear}(b)$ is not defined at situation $\text{do}(\text{move}(a, d), s_0)$, then extensional evaluation is not enough, i.e., a correct hypothesis might entail the example but does not subsume it. This corresponds to the case of a theory that is intensionally complete but extensionally incomplete. For instance, the following effect axiom:

$$\text{holds}(\text{clear}(A), \text{do}(\text{move}(B, C), S)) \leftarrow \text{holds}(\text{clear}(B), S), \dots \quad (4.15)$$

entails E but does not subsume it.

Adopting an intensional evaluation, a clause is evaluated by performing a derivation of each example from a program composed by the clause, the background and the clauses previously learned instead of the atoms in the training set. Thus, intensional evaluation for Situation Calculus programs corresponds to a *temporal projection* problem. By doing so, most of the problems of extensionality are automatically overcome. For instance, intensional evaluation can cope with the incompleteness shown above, because a learned hypothesis may complete the set of instances about a fluent. In this case, the clause above needs to be executed several times to prove E , one for $\text{clear}(c)$ at $\text{do}(\text{move}(b, a), \text{do}(\text{move}(a, d), s_0))$ and one for $\text{clear}(b)$ at $\text{do}(\text{move}(a, d), s_0)$, where the initial situation s_0 corresponds to the *base case* of the recursion.

$$\text{clear}(a)_{s_0} \xleftarrow{\text{needs}} \text{clear}(b)_{s_1} \xleftarrow{\text{needs}} \text{clear}(c)_{s_2}$$

Thus, the proof of E succeeds from the base case even without all intermediate facts. For this to succeed, narratives need to be explicitly represented in the training set, as it is the case in the Situation Calculus. A recursive theory is built incrementally, starting from a *base case* (the

initial situation), i.e., examples are to be proved following the order they have in the narrative. In general, however, intensional evaluation cannot deal with every possible incompleteness of the training set, for instance, it needs that the fluent to be learned is defined in the initial situation.

In the general case of recursion, if the fluent to be learned f_i , depends on the value of other fluents f_j with $j \neq i$, extensional evaluation uses examples for these fluents as additional background knowledge. As a consequence, if the specification of the narratives is partial, the background knowledge for one fluent includes the incomplete data for the other fluents to be learned. In this case, however, intensional evaluation requires to *find an ordering* to learn the multiple fluents sequentially, i.e., the *dependency graph*. Ideally, in the circuit of example (2), fluents should be learned following the relationships $\{sw_1, sw_2, light\}$ or $\{sw_2, sw_1, light\}$, where *light* depends on the state of the switches and not viceversa (Fig. 4.6).

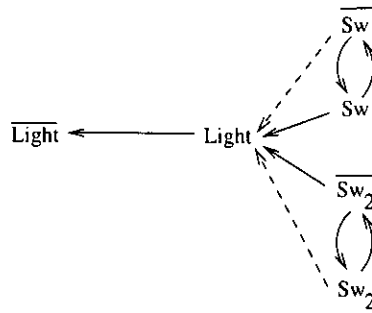


Figure 4.6: Dependency graph of the simple circuit

In practice, these relationships are not known a priori. Furthermore, when learning mutually recursive fluents, the learning of clauses for the different fluents must be interleaved. In particular, Situation Calculus theories are recursive theories where fluents may refer to each other in their definitions, hence intensional evaluation requires interleaving the learning of the different fluents. For instance, the positive and the negative value of sw_1 are mutually dependent when the action $toggle(sw_1)$ is executed, hence the first rule to be learned will be for the positive value (resp. negative) if the switch is open (resp. closed) in the initial situation.

$$\begin{aligned} Holds(closed(sw_1), do(toggle(sw_1), s)) &\leftarrow \neg Holds(closed(sw_1), s) \\ \neg Holds(closed(sw_1), do(toggle(sw_1), s)) &\leftarrow Holds(closed(sw_1), s) \end{aligned}$$

In practice, as we saw in section 3.6, a *multiple search* has to be started for every fluent that has examples that remain to be covered, so that the learning of fluents is interleaved. Unfortunately, as new clauses are learned successively, adding a locally consistent clause for a fluent may make previous clauses for other fluents inconsistent. This happens when the non-derivation of a negative example (for a clause) requires proving with negation as failure *not E*, where *E* will be covered later by another clause. In this case, some of methods commented in chapter 3 must be applied to recover consistency [33, 55, 108].

On the other hand, whereas the hypotheses are always positive programs, the use of default negation for the *affects/3* predicate in the inertia axiom, makes that the derivation of a positive example *E* might require proving with negation as failure *not A* where *A* is the *affects/3* atom corresponding to another example, and thus, it can be the case that *E* is inferred by a set of clauses but not by a superset of those clauses [16]. For instance, if *A* is covered later by some rule, the whole theory will not cover *E*. However, the addition of the *affects/3* atoms to the background and the generation of negative examples for the inertia observations avoids such

non-monotonic effects. By doing so, the extension for the predicate *holds/2* of $B \wedge H$ is always a superset with respect to B , that is, when $B \models E^+$, we have that $B \wedge H \models E^+$, whereas the extension for the predicate *affects/3* does not vary, i.e., the learned theory does not produce new *affects/3* literals for the narratives in the training set. As a consequence, the derivation of a positive example never requires proving with negation as failure *not E*, however, as we have seen, it is still possible for the non-derivation of a negative example¹.

In general, only intensional evaluation can guarantee global properties of the learned theory. However, the main problem of intensional evaluation is the computational cost with respect to extensional evaluation, hence most current ILP algorithms still adopt the latter or hybrid approaches [55]. Unlike this, extensional evaluation is not order-dependent, but it may produce theories that are extensionally consistent but intensionally inconsistent, and intensionally complete but extensionally incomplete. However, these cases arise only when some important positive examples are missing. Furthermore, extensional evaluation needs to explicitly disregard *non-terminating* programs, however, a Situation Calculus program containing only effect axioms is guaranteed to have a finite recursion due to the shape of the effect axioms. Cycles are only possible when so-called indirect effects are considered².

The prototype described in chapter 5 currently adopts extensional evaluation with a special mechanism for dealing with non-terminating programs (see chapter 6). When the narratives provided in the training set are completely specified—all fluents have a defined value in any situation s' between the initial situation s_0 and the last situation s of the narrative—the prototype learns a theory that is globally consistent and extensionally complete³. This is so because any hypothesis together with the background cannot entail facts that are not in the evidence. As a consequence, every learning task does not need to take into account learned information and keeps always the same background knowledge for learning new definitions. Unlike this, when the narratives are partially specified, e.g., a fluent becomes unknown in a situation, there is no such guarantee, and additional procedures should be added.

4.3.5 A three-valued setting for SC programs

When classifying a yet unseen object, an agent needs to distinguish between allowed actions, forbidden actions, and actions with an unknown outcome, and therefore it needs to learn in a three-valued setting. The use of a third value introduces more situations to be managed with respect to a two-valued setting.

Following the notation of Lamma et al. [63], the theory that is learned will contain rules (for

¹Note that this could also happen when only positive programs are allowed.

²The problem of non-terminating programs due to cyclic theories will be dealt with in chapter 6.

³Together with the addition of the *affects/3* atoms to the background and the generation of negative examples for the inertia observations.

every fluent f) of the following form:

$$\begin{aligned}
[\neg] Holds(f, s_0) &\leftarrow \\
Holds(f, do(a, s)) &\leftarrow \pi^+ \\
\neg Holds(f, do(a, s)) &\leftarrow \pi^- \\
Affects(a, f, s) &\leftarrow \pi^+ \\
Affects(a, f, s) &\leftarrow \pi^- \\
Holds(f, do(a, s)) &\leftarrow Holds(f, s) \wedge \text{not } Affects(a, f, s) \\
\neg Holds(f, do(a, s)) &\leftarrow \neg Holds(f, s) \wedge \text{not } Affects(a, f, s)
\end{aligned}$$

where π^+ (resp. π^-) is the definition learned for the positive (resp. negative) value. These definitions are also used for *affects/3* clauses.

The possibility of dealing with *contradictory theories*, makes that the requirement of consistency with respect to the training set can be enlarged to require that the program be non-contradictory also for unseen atoms [63], i.e., $B \cup H \not\models L \wedge \neg L$ for every atom L of the target predicate. In order to handle possible contradiction in [63], contradictory learned rules are defused (case 3 in table 4.3) by making the learned definition for a positive concept p depend on the default negation of the negative concept. The undefined classification is obtained by making opposite rules mutually defeasible, or “nondeterministic”. In case of contradiction $\neg\pi^+ \wedge \pi^-$ is true— this will introduce mutual circularity, and hence undefinedness in the WFSX semantics. As pointed in [63] identifying contradictions on unseen literals is useful to detect spaces to be further explored.

Note that π^+ and π^- can display as well the undefined truth value, when the value of a fluent could not be determined. If one of π^+ or π^- is undefined and the other is true, then the rules make both *holds(f, s)* and \neg *holds(f, s)* undefined. However, as pointed in [63] this is counter-intuitive, a defined value should prevail over an undefined one. For this task, Lamma et al. use additional clauses where a predicate *Undefined* is explicitly used.

1	π^+ and π^- are defined	π^+ true	true
2	π^+ and π^- are defined	π^- true	false
3	π^+ and π^- are defined	$(\pi^+ \wedge \pi^-)$ true	undefined
4	π^+ and π^- are defined	$(\pi^+ \vee \pi^-)$ false	inertia
5	π^+ is defined	π^+ true	true
6	π^+ is defined	π^+ false	undefined
7	π^+ and π^- are undefined	—	undefined

Table 4.3: Possible situations in a three-valued setting

In the stable model semantics when both π^+ and π^- are true, the theory becomes inconsistent, i.e., it has no models. To avoid them we just need to disable all rules that prove *holds(f, s)* and \neg *holds(f, s)*.

$$\begin{aligned}
holds(f, s) &\leftarrow \pi^+, \text{not } \pi^- \\
\neg holds(f, s) &\leftarrow \pi^-, \text{not } \pi^+
\end{aligned}$$

If one of π^+ or π^- is unknown and the other is true, the defined value prevails over the undefined one, because the default negation of an unknown value is true (case 5 in table 4.3). When both π^+ and π^- are false, inertia is applied (case 4). Furthermore, if one of π^+ or π^- is undefined and the other is false, then the rules above enable inertia, since NAF is used. However, we may want to disable the inertia rule in certain cases. For instance, if we do not know whether the switch sw_1 is currently closed, then we do not want to conclude by inertia that the value of *light* will remain the same after closing sw_2 . In this case we need to disable the inertia rule not only when the preconditions for the change in the value of a fluent are known to hold, but whenever there is no evidence that they do not hold. This requires the modification of rules for the predicate *affects*/ β in the final theory [46].

$$\text{Affects}(a, f, s) \leftarrow \text{not } \neg\pi \quad (4.16)$$

where π is of the form $\text{Holds}(P_1, s), \dots, \text{Holds}(P_n, s)$ and $\text{not } \neg\pi$ is of the form $\text{not } \neg\text{Holds}(P_1, s), \dots, \text{not } \neg\text{Holds}(P_n, s)$. Thus, the final theory is in the form:

$$\begin{aligned} \neg\text{Holds}(f, s_0) &\leftarrow \\ \text{Holds}(f, \text{do}(a, s)) &\leftarrow \pi^+, \text{not } \text{Undefined}(f, a, s) \\ \neg\text{Holds}(f, \text{do}(a, s)) &\leftarrow \pi^-, \text{not } \text{Undefined}(f, a, s) \\ \text{Affects}(a, f, s) &\leftarrow \text{not } \neg\pi^+ \\ \text{Affects}(a, f, s) &\leftarrow \text{not } \neg\pi^- \\ \text{Holds}(f, \text{do}(a, s)) &\leftarrow \text{Holds}(f, s) \wedge \text{not } \text{Affects}(a, f, s) \\ \neg\text{Holds}(f, \text{do}(a, s)) &\leftarrow \neg\text{Holds}(f, s) \wedge \text{not } \text{Affects}(a, f, s) \\ \text{Undefined}(f, a, s) &\leftarrow \pi^+, \pi^- \end{aligned}$$

The predicate *affects*/ β is applied whenever the negation of π^+ or π^- cannot be proved. The predicate *Undefined*/ β avoids contradiction when both π^+ and π^- are true in a situation. The form of the theory differs partially from that given in [9, 46]. In [46], contradiction is not explicitly managed. With respect to [9], the predicate *Undefined*/ β includes as argument the fluent and depends on the preconditions of the action and not on *Affects*, thus a defined value for one of π^+ or π^- will prevail over an undefined one.

4.4 Conclusions

In this chapter, we formally defined the problem of learning to predict the effects of actions using Machine Learning methods that are oriented to classification problems. Unlike previous approaches, we use non-monotonic Action Languages to represent and learn the effects of actions. This allows us, for instance, to avoid the explicit representation of unchanged properties from one particular situation to another, and provides with a natural and homogeneous representation. On the other hand, most implementations of Action Theories use logic programming directly to formalize some aspects of reasoning about actions or provide translations to logic programs. This allows that Inductive Logic Programming methods can be applied effectively. In particular we used a Logic Programming implementation of the Situation Calculus, which combines expressivity and simplicity. Furthermore, the form at which actions, effects and situations are represented in the Situation Calculus allows that multiple narratives starting from different initial situations, can be used for learning in an homogeneous and natural way. In the next chapter we describe a prototype implementation based on the framework presented in this chapter.

Chapter 5

Implementation: *LRAC*

We have developed a prototype *LRAC* (Learning to Reason about Actions and Change) [72] implemented in the YAP Prolog system [24] consisting of a top-down ILP algorithm. The aim of the implementation is not producing a final system but to provide with a prototype that allows to explore the application of different techniques and strategies. A specific implementation is needed for the management of particular features of action formalisms that must be integrated in the learning process, rather than using directly any ILP algorithm. This also makes the prototype independent of the ILP method used. Additionally, it allows to apply a semantics for Extended Logic Programs instead of the normal Prolog proof procedure and introduce necessary features to deal with multiple predicate learning tasks, which is an issue usually ignored in most popular methods.

5.1 Learning Action Theories with ILP methods

We follow an approach to learning action theories in the Situation Calculus that consists in applying conventional ILP techniques to learn a definition for both E^+ and E^- . The implemented ILP algorithm is mostly based on Progol [95] –that implements Inverting Entailment (IE)– focused to and augmented with particular features needed for learning action theories. Progol is the most representative system in the ILP field. Progol has been applied successfully in several real problems and the results have been published in relevant journals in those fields. Another particular feature that makes it attractive is the use of a most-specific clause to delimit the search space. Progol learns the most general definition for a target predicate unlike Golem that learns the most specific generalization, however, it would be possible to integrate both methods in the prototype, so that the user can choose the level of generalization for each domain and even differently for the positive and negative values [63].

Inverting Entailment (IE) computes the so-called bottom clause or \perp -clause (most specific generalization) from a positive example (*seed*). The \perp -clause is computed from the least Herbrand model of $B \wedge \neg E^+$, where E^+ is the seed and B is the background knowledge. Theoretically, the bottom clause includes all the background knowledge, however in practice, only the background relevant to the seed is considered by applying some biases.

In particular, the Situation Calculus introduces a bias for such clause, where only the previous situation can appear in the body of a clause and no references are possible to the predicate *affects/3*. This is the form of so-called *effect axioms*. Since each example E corresponds to a particular situation $do(a_i, s_i)$, where $do(a_i, s_i)$ is the constant used in describing E , background

knowledge which does not pertain to s_i is irrelevant and need not be considered in the bottom clause. For instance, in the example of the Blocks world shown in section 4.3.4, the bottom clause generated from the seed

$$\text{holds}(\text{clear}(c), \text{do}(\text{move}(b, a), \text{do}(\text{move}(a, d), s_0)))$$

includes the values of fluents $on/2$ and $clear/1$ at situation $\text{do}(\text{move}(a, d), s_0)$ and other static predicates that are specific of the domain. Generalization is applied for every fluent to be used as condition in the body and for the fluent (and the action) in the head, following a syntactic bias. Similarly, a bias can be applied over the \perp -clause at a clause level and not at a literal level for a particular fluent.

In general, however, Inverting Entailment is incomplete with recursive clauses when some important training examples are missing. This incompleteness is also present when learning action theories. For instance, when

$$\text{holds}(\text{clear}(b), \text{do}(\text{move}(a, d), s_0))$$

is missing, the bottom clause does not include the intended effect axiom (eq. 4.15), hence IE will not find the solution. These cases would require that the bottom-clause is applied recursively. Muggleton showed that enlarging the bottom clause leads to completeness of IE [90], however this has not been implemented yet. The search in Progol is delimited by the empty clause and the \perp -clause, where a possible hypothesis H is constructed as a clause that subsumes \perp .

We adopt the *answer set* semantics for learning action theories in the form of ELP. We follow the approach to learning ELPs of [63, 54] that consists in applying conventional ILP techniques where the answer set semantics substitutes the standard LP proof procedure to test the coverage of examples. In the answer set semantics, the answer to a ground query A is either yes, no or undefined depending whether the answer set contains A , $\neg A$ or neither [54]. A stable models semantics for extended logic programs (ELP) is given by computing the *answer sets* of a program where classical negation is “implemented” by reifying the truth value and adding some consistency axioms.

```
false :- holds(F,+,S),holds(F,-,S).
```

This makes that when the model includes both an atom and its negation, the program becomes inconsistent. The prototype implemented uses *smodels* [96] that is an efficient implementation of the stable models semantics. The covering algorithm computes the finite *stable model* of the theory to test the coverage of examples. We assume that Background B is a *consistent* logic program i.e., B has a single model, and the same assumption is done for $B \cup H$ where H is a possible hypothesis. Thus, the unique and finite answer set is computed through *smodels*, and the coverage test must check that every example is included in the unique stable model of the program $B \cup H$. Initially, Progol is called with the computed stable model as background knowledge from which the bottom-clause is computed. The background given to Progol is constructed from the unique stable model of $B \cup E^+$.

5.2 Description of *LRAC*

The system *LRAC* is invoked with the command:

```
lrac <domain> [<fluent>]
```

that launches the Yap Prolog, where `<domain>` is the domain to be learned and `<fluent>` is an optional argument that instructs *LRAC* to focus on a particular fluent. Input to *LRAC* is given in three separate files:

1. Background knowledge (`<domain>.bg`)
2. Positive examples (`<domain>.train`)
3. Declarative bias (`<domain>.bias`)

Background knowledge (`.bg`) includes domain-dependent definitions. This must include user-defined type declarations to be used by the inductor to focus on well-formed clauses¹. The syntax adopted is that of the system *smodels* [96] that computes the stable models of a logic program. For instance, for the circuit of example 2 the following definitions are provided:

```

sort(device;switch).

sorts(active(device)).
sorts(closed(switch)).
sorts(toggle(switch)).

switch(sw1;sw2).
device(light).

fluent(closed(W)) :- switch(W).
fluent(active(W)) :- device(W).
action(toggle(W)) :- switch(W).

```

Predicate `sort/1` specifies the list of sorts of a particular domain apart from the domain-independent sorts, namely $\{fluent, value, action, situation\}$, whereas `sorts/1` specify the sorts for every argument of a predicate. Possible values for every sort are given by using the sort name as a functor, and separated by a `;`. The form of fluents and actions (and static predicates) is given through sort information for every argument.

File `.train` includes observations in the form of ground `holds/3` facts. The truth value is reified to allow the derivation of negative value fluents, hence negative information will be represented as `holds(F, -, S)`. `holds/3` facts for the initial situation(s) are then explicitly separated and included in the background together with the inertia axiom.

```

holds(closed(sw1),+,do(toggle(sw1),s0)).
holds(active(light),+,do(toggle(sw1),s0)).
holds(closed(sw2),-,do(toggle(sw2),do(toggle(sw1),s0))).
holds(active(light),-,do(toggle(sw2),do(toggle(sw1),s0))).
holds(closed(sw2),+,do(toggle(sw2),do(toggle(sw2),do(toggle(sw1),s0)))).
holds(active(light),+,do(toggle(sw2),do(toggle(sw2),do(toggle(sw1),s0)))).

```

Positive `affects/3` atoms are internally generated for every positive example and added to the background.

¹Otherwise they could be partially constructed from data [83]


```

holds(F,V,do(A,S)) :- holds(F,V,S),not affects(A,F,S).

affects(toggle(sw1),active(light),do(toggle(sw2),s0)).
...

```

LRAC initially loads the training set as well as background predicates.

1. Read training set `domain.train`.
2. Compute the stable model.
3. Read bias file `domain.bias`.
4. Retrieve all situation terms.
5. Generate negative examples.

Figure 5.1: Loading a domain into *LRAC*

Negative examples in the form `holds(f,v,s)` are added for every situation (included in the training set) where a fluent *f* is not affected by an action. These are generated automatically given the high number of negative examples that are possible.

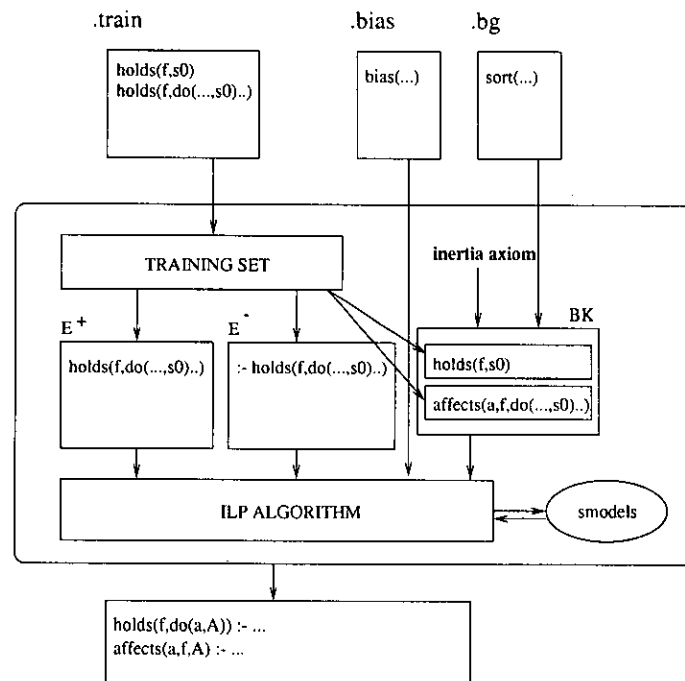


Figure 5.2: Using an ILP algorithm to learn Situation Calculus theories

Before learning, a bias must be specified for each learning task to delimit the language used during the learning process. The built-in `bias/2` predicate establishes which predicate will be generalized, i.e., the predicate allowed to appear in the head of clauses.

$$\text{bias}(\text{Head}, \text{BodyLiterals})$$

When forming rules with heads subsumed by *Head*, the bias disregards literals other than those included in the list *BodyLiterals*. These `bias/2` declarations replace both `mode` declarations of Prolog where declarations for the body literals are assigned to each target predicate

individually following *determination/2* declarations. Additionally, these *determination/2* declarations can only refer to the functor and the arity of the predicates, and thus it is not possible to assign different *modeb* declarations for a single fluent in different *modeh/2* declarations. We found this feature useful when multiple predicates have to be learned such that the bias for body literals can be different for every fluent –i.e., for each argument of predicate *holds/3* and for each action. This will be also useful to independently bias the search for direct and indirect effects with the same fluent allowing for different mode declarations (this will be dealt in chapter 6). Bias declarations for the circuit (2) are:

```
bias(holds(closed(#),#,do(#,+)),
     [holds(closed(#2),#,+),holds(active(#),#,+)]).
bias(holds(active(#),#,do(#,+)),
     [holds(closed(#2),#,+),holds(active(#),#,+)]).
```

Symbols + and - are meant to be replaced by input (resp. output) variables during the construction of the \perp -clause. Unlike this, # is meant to be replaced by a constant, so that for instance, no generalization is applied over the action *toggle/1*. The *recall* for each literal –i.e., number of alternative solutions for instantiating an atom– is assumed to be 1 by default, otherwise it must be explicitly provided for one of arguments of a literal. For instance, with the fluent *closed/1*, two possible instantiations are possible, one for each switch.

Two situations are included in a Situation Calculus formulae, namely, the previous situation and the situation resulting of executing an action. This introduces an implicit *bias* for the clauses to be learned, where only the previous situation can appear in the body of a hypothesized clause. Thus, any literal added to a clause `holds(F,do(a,S)) :- ...` will refer to *S* (not to `do(a,S)`). From the *bias/2* declarations, mode declarations for Progol are built, by using information about sorts.

```
:- modeh(1,holds(active(#device),#value,do(toggle(#switch),+sit)))?
:- modeh(1,holds(closed(#switch),#value,do(toggle(#switch),+sit)))?
:- modeb(2,holds(closed(#switch),#value,+sit))?
:- modeb(1,holds(active(#device),#value,+sit))?
```

After loading a domain, LRAC starts learning for the domain specified and optionally for the fluent specified, where the argument <fluent> can be a ground or non-ground term, which will make the learned clauses more or less specific respectively. For instance,

```
lrac sw closed(_)
```

learns a definition for each possible instantiation of fluent *closed/1*. When `lrac` receives no fluent as argument, it learns rules for all fluents, provided the necessary *bias/2* declarations are provided in the `.bias` file. For each seed *e*, the first bias declaration *bias(h,b)* is found such that *h* subsumes *e* with substitution θ . Examples to be covered (resp. not covered) are filtered according to the second argument of *lrac* (when this is provided). A more accurate filtering is done after getting a *bias/2* statement by using the head of the \perp -clause, that helps to reduce the overhead of the testing phase.

The learning procedure will be called twice, once for the positive concept and once for the negative concept and separately for every action that is shown to cause the fluent to change in any situation. The search in Progol is delimited by the empty clause and the so-called \perp -clause (most specific generalization) constructed from a *seed*, whose size and form are controlled by applying

bias declarations and other special settings. The phase of constructing the bottom-clause can be carried out by Progol independently of the search process. For this reason, we have modified a version of Progol4.4 [95] that simply returns the bottom clause for a given seed –forcing it as the first example in the input file– without starting the search process. Once Progol has computed the most specific clause, this is redirected to a file and read into memory. For instance, for the fluent `active(light)` and the seed:

```
holds(active(light),+,do(t1,do(t2,s0))).
```

Progol returns:

```
holds(active(light),+,do(toggle(sw1),A) :-
    holds(closed(sw2),+,A), holds(closed(sw1),-,A),
    holds(active(light),-,A).
```

The size of the \perp -clause may become significantly large specially when dealing with non-determinate literals. Additional restrictions can be specified apart from the *recall* and the *bias/2* statements by using Progol settings, among others:

set(i,_)	depth of variables in the body
set(c,_)	the maximum number of literals in the body of a clause

Table 5.1: Additional settings for controlling the size of the bottom clause

The declarative bias language used consists of *mode declarations* which is typical of most popular ILP systems, and it is enough for most applications. However, it is useful to have additional expressivity to further pruning the search space. For this reason, we implemented an additional bias mechanism through built-in predicates `prune/2` and `remove/2` that have their corresponding counterparts in Progol. The predicates take two arguments, the head and the body of a clause, and the definition establishes conditions on them that if matched, will remove the clause from search (*remove/2*) and all its refinements (*prune/2*). These pruning statements are extremely useful for stating which kinds of clauses should not be considered in the search. The main advantage of using the LP syntax is that pruning clauses can be expressed by using arbitrary logic programs. For instance, suppose you wanted to disallow self-recursive clauses. This can be achieved by using the following simple prune rule [95]:

```
prune(Head,Body) :- in(Head,Body).
```

The above *prune/2* statement disregards any clause whose *Head* unifies with an atom in its *Body*. We can also decide on to allow *unbounded variables* in the head or body of a clause, and so on. A built-in predicate *in(Lit, Body)* is used to represent that a literal *Lit* is in the body of the clause. Unlike *prune/2*, *remove/2* statements are useful when the current hypothesis is not well-formed, e.g., it is non-constrained, and however, some of the refinements are of interest. We also implemented a variant of *prune/2* that is applied to the bottom clause (before search) to remove those literals that are redundant or are not of interest. The *prune/2* statements can serve the same purpose, however, *prunebottom/3* statements reduce the size of the bottom clause, and thus the number of hypotheses to be explored.

```
prunebottom(Lit,BottomHead,BottomBody).
```

During search, hypotheses are refined by adding new literals from the bottom-clause and respecting mode declarations. For instance, given a mode declaration for the function `succ(+, -)`, the `-` is used in the output argument, so that when an atom `succ/2` is added to the current hypothesis, the first argument must be already bounded and the second argument is the result of the function. As a consequence, the second argument can introduce new variables in the hypothesis. However, an atom `succ(A, B)` with `B` unbounded, is not of interest without adding another atom that also refers to `B`. If this happens in the bottom clause, the atom should be removed (with a `prunebottom/2` statement), however if it happens in the current hypothesis, this should be specialized without being tested previously (with a `remove/2` statement). We add a special predicate `constrained(H, B, succ(+, +))` to mean that `+` arguments are bounded in the clause `H :- B`.

Progol uses a standard covering algorithm where each example is generalized to find the hypothesis that covers the given example with maximal information compression. Positive examples covered by each rule are removed from E^+ and the process is repeated by taking another example from the remaining examples. Finally, the learned rule is added to the background (Fig. 5.3).

1. If $E = \emptyset$ return B
2. Let e be the first example in E
3. Construct clause \perp for e
4. Construct clause H from \perp
5. Let $B = B \cup H$
6. Let $E' = \{e : e \in E \text{ and } B \models e\}$
7. Let $E = E - E'$
8. Goto 1

Figure 5.3: Progol covering algorithm

The search algorithm of Progol is implemented in the Yap Prolog that lists those clauses which subsume the bottom-clause and which might be part of a final hypothesis, and tests each hypothesis on the positive and negative examples. The algorithm searches in a top-down fashion for the best generalization of \perp using an A^* search to find a subset of literals in the body with the maximal compression –such that the resulting clause is well-formed according to the syntactic bias. The \perp clause is processed in order from left to right (starting with the head), and respecting the input/output arguments of predicates, so that a queue of current candidate clauses is maintained ².

1. initialize queue to contain the clause with an empty body.
2. remove candidate s with highest f_s from the queue.
3. if $n_s = 0$, $f_s > 0$ and $f_s \geq g_r$ for all r that have been visited so far, then return s .
4. unless $n_s = 0$, $g_s \leq 0$ or $c_s \geq c$, refine s .
5. add all refinements to the queue.
6. unless the queue is empty goto 2.

Figure 5.4: Progol search algorithm

²This ordering introduces an additional bias in the formation of candidate hypotheses [4]

The compression measure used in Progol is of the form [95]:

$$\begin{aligned} g_s &= p_s - c_s - h_s \\ f_s &= g_s - n_s \end{aligned}$$

where p_s (resp. n_s) are the positive (resp. negative) examples covered by the clause, c_s is the current length of the clause, and h_s is an optimistic estimate of the number of literals needed to make the clause I/O complete (a value is derived for all output variables of its head).

Pruning of uninteresting candidates is done at step 3, so that some conditions must be fulfilled by clauses to be further specialized, namely, no negatives are covered, the length must be less than the maximum allowed, and a minimal compression is required ($g_s \leq 0$) –i.e., any hypothesis must cover at least as many positives examples as the number of literals in the body. If the queue becomes empty, the seed is returned without generalization. All examples that tried to generalize and however no rule was learned because no enough compression was achieved, are returned together with the learned clauses. The search stops when it is guaranteed that no better clause in the queue, with a length $cs \leq c$ (where c is the maximum number of literals allowed in the body) will be found, optimistically assuming that further specialization of the clauses will remove all negative examples covered while keeping all positive examples covered, i.e., g_s is an upper bound of f_r for all refinements r of s . As in Progol, a limit can be imposed on the number of hypotheses to be searched in complex domains.

The most costly operation in the search process is the *coverage testing* made for every candidate hypothesis. By default, extensional coverage testing is adopted so that positive examples are added to the background. The actual reason is that, in practice, intensional testing makes learning very costly and the execution times increase very significantly with respect to extensional testing. With extensional evaluation, the order at which fluents are learned is not important. In practice, *LRAC* takes a seed randomly from the training set, so that fluents are not necessarily learned completely one after another.

The interpreter for the stable model semantics needs the logic programs to be grounded. With extensional coverage testing, the unique stable model has to be computed just once. Intensional testing corresponds to the entailment relation, i.e., the stable model of $B \cup H$ is to be computed for every H , hence intensional coverage is very costly in most cases. Variables are assumed to range over the object constants used in the program so that clauses are constrained (by using type predicates) to allow for a finite grounding from which, the stable model is computed. Due to the range restriction, every situation included in any narrative must be included a priori as a term, so that the range of the function $do/2$ is restricted to the situation constants corresponding to the narratives in the training set.

```
situation(s0).
situation(do(toggle(sw1),s0)).
situation(do(toggle(sw2),do(toggle(sw1),s0))).
```

Noise is another important issue when dealing with real world data. We are not specially concerned with noise treatment, however there exist many approaches in the literature to noise most based on pruning methods or compression. Progol relies on a compression measure to avoid overfitting the training set, i.e., obtaining very specific rules which constitute very poor generalizations. Noise can be also treated as a kind of non-determinism and dealt with by relaxing the consistency requirement [5] (see section 7.2).

5.2.1 Learning the basic circuit with *LRAC*

We used a simple *wander* program that collects data about its actions while exploring the environment.

```
wander <domain> <narratives> <length>
```

Argument <domain> is the name of a file that contains the domain description needed for the simulation, from which *wander* will generate <narratives> narratives of length up to <length>. With each exploratory step, the program records the action that was taken, and the fluents that changed. We assume that the environment is only manipulated by one agent and there is no noise or uncertainty in the observations and actions.

LRAC initially loads the domain data.

```
$ lrac sw
[ YAP version Yap-4.3.18 ]
L R A C Version 18/01/2001
File sw
Actions: toggle/1
Fluents: closed/1 active/1
Reading training set...(40s) done.
Executing smodels ... done.
Reading bias... done.
```

and then it starts learning taking a positive example, computing the most specific clause and searching for the best clause.

```
[Generalizing holds(closed(sw1),-,do(toggle(sw1),do(toggle(sw1),s0)))].]
Calling progol...done
[Most specific clause is]
holds(closed(sw1),-,do(toggle(sw1),A)) :-
  holds(closed(sw1),+,A),
  holds(closed(sw2),-,A),
  holds(active(light),-,A).
Searching... done.
4 nodes explored
[Result of search is]
holds(closed(sw1),-,do(toggle(sw1),A)) :-
  holds(closed(sw1),+,A).
[18 redundant clauses retracted]
...
```

From 10 narratives of length 4 including 50 positive examples, *LRAC* returned the following clauses in the basic circuit:

```
holds(closed(sw1),+,do(toggle(sw1),A)) :-
  holds(closed(sw1),-,A).
holds(closed(sw1),-,do(toggle(sw1),A)) :-
  holds(closed(sw1),+,A).
holds(closed(sw2),+,do(toggle(sw2),A)) :-
  holds(closed(sw2),-,A).
holds(closed(sw2),-,do(toggle(sw2),A)) :-
  holds(closed(sw2),+,A).
holds(active(light),+,do(toggle(sw1),A)) :-
```

```

holds(closed(sw1),-,A),
holds(closed(sw2),+,A).
holds(active(light),+,do(toggle(sw2),A)) :-
  holds(closed(sw1),-,A),
  holds(closed(sw2),+,A).
holds(active(light),-,do(toggle(sw1),A)) :-
  holds(active(light),+,A).
holds(active(light),-,do(toggle(sw2),A)) :-
  holds(active(light),+,A).

```

According to the rules, action *toggle*(*sw*₁) (resp. *toggle*(*sw*₂)) toggles switch *sw*₁ (resp. *sw*₂), and both switches affect *light* through the actions that modify them. The corresponding *affects*/*3* clauses are generated automatically from the learned effect axioms, since they have the same bodies [121]. This allows to distinguish “caused” values, so that the inertia axiom is disabled and no contradiction is produced.

```

affects(toggle(sw1),closed(sw1),A) :-
  holds(closed(sw1),+,A).
affects(toggle(sw1),closed(sw1),A) :-
  holds(closed(sw1),-,A).
affects(toggle(sw2),closed(sw2),A) :-
  holds(closed(sw2),+,A).
affects(toggle(sw2),closed(sw2),A) :-
  holds(closed(sw2),-,A).
affects(toggle(sw1),active(light),A) :-
  holds(closed(sw1),-,A),
  holds(closed(sw2),+,A).
affects(toggle(sw1),active(light),A) :-
  holds(active(light),+,A).
affects(toggle(sw2),active(light),A) :-
  holds(closed(sw1),-,A),
  holds(closed(sw2),+,A).
affects(toggle(sw2),active(light),A) :-
  holds(active(light),+,A).

```

5.2.2 The Blocks world

Let us consider again the Blocks world (Fig. 5.5).

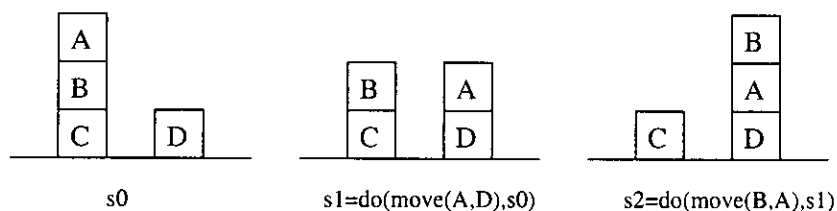


Figure 5.5: Blocks world

Background knowledge includes type definitions and the objects included in the scenario. We include an special predicate *diff*/*2* to represent that two variables do not represent the same object, i.e., $X \neq Y$, where X and Y are variables.

```

sort(block;location).
sorts(on(blok,location)).
sorts(clear(blok)).
sorts(move(blok,location)).
sorts(diff(blok,location)).
sorts(table(location)).

block(a;b;c;d).
location(table).
location(A) :- block(A).
table(table).

fluent(clear(B)) :- block(B).
fluent(on(A,B)) :- block(A),location(B),diff(A,B).
action(move(A,B)) :- block(A),location(B),diff(A,B).

diff(A,B) :- location(A),location(B),A!=B.

```

The sequence of actions of Fig. 5.5 is represented as the following LP facts:

```

holds(clear(a),+,s0).
holds(clear(d),+,s0).
holds(clear(b),-,s0).
holds(clear(c),-,s0).
holds(on(a,b),+,s0).
holds(on(b,c),+,s0).
holds(on(c,table),+,s0).
holds(on(d,table),+,s0).

holds(on(a,b),-,do(move(a,d),s0)).
holds(on(a,d),+,do(move(a,d),s0)).
holds(clear(b),+,do(move(a,d),s0)).
holds(clear(d),-,do(move(a,d),s0)).
holds(on(b,c),-,do(move(b,a),do(move(a,d),s0))).
holds(on(b,a),+,do(move(b,a),do(move(a,d),s0))).
holds(clear(c),+,do(move(b,a),do(move(a,d),s0))).
holds(clear(a),-,do(move(b,a),do(move(a,d),s0))).

```

The use of many-sorted predicates for actions and fluents require a little effort in finding a bias sufficient for learning, further than controlling the situation variable. For instance, when objects other than situations are to be considered, we need to decide to instantiate or not every predicate argument, so that generalization works over them.

```

bias(holds(on(+,#,do(move(+,#,+))),
    [holds(on(+,#,+),holds(clear(+,#,+),diff(+,#,+))]).
bias(holds(clear(+,#,do(move(+,#,+))),
    [holds(on(+,#,+),holds(clear(+,#,+),diff(+,#,+))]).

```

For structured fluents, the generation of negative examples requires to build all the possible instantiations of fluents according to the argument *Head* of the appropriate *bias/2* statement. In the blocks world, from the example `holds(on(a,b),+,do(move(a,b),s1))` and the above bias statements, the following negative examples are generated, representing effects that were not produced by the action *move/2* at any situation:


```

holds(on(b,a),-,do(move(a,b),s1)).
holds(on(a,c),-,do(move(a,b),s1)).
...
holds(on(b,a),-,do(move(a,b),s2)).
...

```

Before starting its search, *LRAC* calls Progol to construct the “most specific clause”. For instance, for the predicate *on/2* the clause constructed was:

```

holds(on(A,B),+,do(move(A,B),C)) :- holds(on(A,A),-,C), holds(on(A,
    B),-,C), holds(on(B,A),-,C), holds(on(B,B),-,C), holds(clear(A),
    t,C), holds(clear(B),+,C), diff(A,table), diff(A,B), diff(B,
    table), diff(B,A).

```

Literals in the form *holds(on(A,A),-,D)* are not of interest, and introduce an overhead for the search process. To avoid these useless literals, we add a *prunebottom/3* statement like the following:

```

prunebottom(holds(on(A,A),-,_),Head,Body).

```

Functional dependences for *on(+,-)* and *on(-,+)* can be also incorporated as well as *symmetry* properties for *diff/2* and *on/2*, and the *transitive* property of *diff/2*.

```

prune(Head,Body) :- in(holds(on(A,B),-,_),Body),in(holds(on(B,A),-,_),Body).
prune(Head,Body) :- in(holds(on(A,_),+,_),Body),in(holds(on(A,_),-,_),Body).
prunebottom(diff(A,B),Head,Body) :- in(diff(B,A),Body).
prune(Head,Body) :- in(diff(A,B),Body),in(diff(B,C),Body),in(diff(A,C),Body).

```

We generated 144 situations including 133 positive examples and 2448 negative examples. From these examples, *LRAC* learned the following effect axioms:

```

holds(on(A,B),+,do(move(A,B),C)) :-
    holds(clear(A),+,C),
    holds(clear(B),+,C).
holds(on(A,B),+,do(move(A,B),C)) :-
    holds(on(A,B),-,C),
    holds(clear(A),+,C),
    table(B).
holds(on(A,B),-,do(move(A,D),C)) :-
    holds(on(A,B),+,C),
    holds(clear(A),+,C),
    holds(clear(D),+,C),
    diff(B,D).
holds(on(A,B),-,do(move(A,D),C)) :-
    holds(on(A,B),+,C),
    holds(clear(A),+,C),
    table(D).
holds(clear(A),+,do(move(C,D),B)) :-
    holds(clear(D),+,B),
    holds(clear(C),+,B),
    holds(on(C,A),+,B),
    diff(D,A).

```

```
holds(clear(A),+,do(move(C,D),B)) :-  
  holds(clear(C),+,B),  
  holds(on(C,A),+,B),  
  table(D).  
holds(clear(A),-,do(move(C,A),B)) :-  
  holds(clear(C),+,B),  
  holds(clear(A),+,B).
```

The rules for *on/2* states that moving a block *A* onto *B* will be possible when both blocks are clear or when the first block is clear and *B* is the table, and that a block *A* will not be over *B* (a block or the table) if we move *A* to a third (different) block *C* when both *A* and the target block *C* are clear. Similarly for *clear/1*. Unbounded variables are displayed as `_` by the Yap Prolog.

A difference with the clauses learned by S. Moyle is the clause below that appears in [88] and that trivially states that moving one block onto another will not make the first to be on a third if the second is already on the third.

```
holds(on(A,B),-,do(move(A,C),D)) :- holds(on(B,C),-,D).
```

This is due to a bad generalization caused by either the size of the training set or a skewed training set. The inclusion of any example where the three blocks involved are not forming a tower allowed *LRAC* to obtain the correct clause which correctly states that a block will not be over another block if we move the first one to a third block when both the first and the third blocks are clear.

5.3 Conclusions

In this chapter we have described a prototype for learning Situation Calculus theories in the form of Extended Logic Programs, that is based on the Inverting Entailment method, but it could use other similar methods. The prototype uses a variant of the algorithm Prolog for computing the bottom-clause, an interpreter for computing the stable models of a logic program, and adopts extensional evaluation for efficiency reasons.

Chapter 6

Learning Indirect Effects

The constraints that we have considered so far were associated with a particular action. In many cases, the effects of an action are not caused directly by the execution of an action but indirectly through other simultaneous changes. Such indirect effects are usually represented as consequences of general laws describing dependences between fluents. Previous approaches to learning action models are restricted to predicting a single outcome or effect of an action. This forces the explicit representation of all the effects of an action as direct effects, producing the so-called *Ramification problem* [58]. The learner should infer how properties of a domain are (directly/indirectly) affected by the execution of actions, or otherwise are subject to the general law of inertia. We study the learnability and compactness of the learned theory in this new framework. For this task we will use the standard benchmark scenarios for the ramification problem used in the literature.

6.1 The Ramification problem

The Ramification problem has been described by Ginsberg and Smith [49]:

For any given action there are essentially an infinite number of possible consequences that depend upon the details of the situation in which the action occurs.

For instance, the constraint that a player may not be in check after his own move, represents an implicit precondition on possible moves. It would be very difficult to express this action precondition explicitly, as a condition on the starting state from which the player makes the move [26]. The increased complexity of axiom effects is a consequence of the fact that they need to anticipate the *ramifications* of the executed action, and these become increasingly numerous and complicated as the complexity of the domain increases.

In many cases, the effects of an action are not caused directly by the execution of the action but indirectly through other changes. Such indirect effects are usually represented as consequences of general laws describing dependences between components of the world description. Formally, we have:

Definition 6.1 (Domain constraint) A domain constraint is a formula

$$\text{Holds}(f, s) \leftarrow \pi \tag{6.1}$$

where the *Holds* literals in π are only of the form $[\neg]\text{Holds}(f', s)$. \square

According to these domain constraints, indirect effects are usually derived from the state of fluents, while direct effects come from the execution of an action. Most solutions that have been proposed for the ramification problem require effect axioms to specify the most significant effects of the action and to rely on domain constraints for specifying additional changes that are due to the action. Thus, effect axioms are used to describe the dynamics of the primitive fluents and state constraints are used to describe the derived fluents in terms of the primitive ones. For instance, according to the following constraint:

$$\text{Holds}(\text{active}(\text{light}), A) \leftarrow \text{Holds}(\text{closed}(sw_1), A), \text{Holds}(\text{closed}(sw_2), A) \quad (6.2)$$

light is a derived fluent, such that it is always on when both switches (primitive fluents) are closed, not matter the actions that were executed.

However, the use of domain constraints for the propagation of effects produces counterintuitive solutions even in most simple cases. According to the above constraint, for instance, *light* is always on when both switches are closed. In classical logic, if *sw*₁ is open and *light* is false and then *sw*₁ is closed, the violation of the above constraint can be repaired by making *light* true or making *sw*₂ false, which does not correspond to the intended behavior. The use of causality for the Ramification problem avoids this situation by introducing directionality in the formula [67, 101], i.e., *light* is the effect and never the cause.

6.2 Indirect effects in the Situation Calculus

The basic definition of the Situation Calculus suffers from the Ramification problem, given that the predicate *affects/3* that is used to overcome the Frame problem, cannot be used to represent indirect effects because the situation argument represents the situation where the action is executed and not the resulting situation, and the action is a required argument. However, the introduction of domain constraints in the Situation Calculus is not sufficient. A fluent that has been initiated/terminated directly through an effect axiom cannot then be terminated/initiated indirectly through a state constraint, unless it is released from inertia beforehand [123], otherwise it would lead to contradiction. To add indirect effects to the Situation Calculus theories we need to consider action formalisms where a notion of causation is explicitly represented about how changes in one state variable may cause changes in another state variable.

We use a dialect of the Situation Calculus [67] with a predicate *caused/3* instead of *affects/3*. F. Lin [67] incorporates causality into a Situation Calculus based formalism through the predicate *caused(f, v, s)* that is true if the fluent *f* is caused (by something unspecified) to have the truth value *v* in the situation *s*. This allows to express *fluent-triggered* causal statements, i.e., a fluent is caused by other fluent, apart from action-triggered ones, which are convenient for representing the indirect effects of actions. We formally define a Logic Programming implementation of the Lin's dialect in the form:

Definition 6.2 (A Situation Calculus Program with indirect effects) A Situation Calculus program is the conjunction of:

- A finite set of general clauses

$$[-]\text{Holds}(f, s_0) \quad (6.3)$$

where *s*₀ denotes the initial situation.

- A finite set of clauses of the form

$$Caused(f, v, do(a, s)) \leftarrow \pi \quad (6.4)$$

where π does not mention the *Caused* predicate and every occurrence of the *Holds* predicate in π is of the form $[\neg]Holds(f', s)$. The description states that, in any situation, if the precondition holds then the effect will hold in the resulting situation. These axioms are called *effect axioms* or *action laws*.

- A finite set of clauses of the form

$$Caused(f, v, s) \leftarrow \pi \quad (6.5)$$

where π does not mention the *Caused* predicate and every occurrence of the *Holds* predicate in π is of the form $[\neg]Holds(f', s)$. The description represents how changes in certain fluents propagate to cause changes in other fluents. These axioms are called *causal laws*.

- The universal frame axiom describes how the world stays the same (as opposed to how it changes)

$$Holds(f, do(a, s)) \leftarrow Holds(f, s) \wedge \text{not } Caused(f, v, do(a, s)) \quad (6.6)$$

$$\neg Holds(f, do(a, s)) \leftarrow \neg Holds(f, s) \wedge \text{not } Caused(f, v, do(a, s)) \quad (6.7)$$

- Some clauses that propagate caused values to *Holds*.

$$Holds(f, do(a, s)) \leftarrow Caused(f, \text{true}, do(a, s)) \quad (6.8)$$

$$\neg Holds(f, do(a, s)) \leftarrow Caused(f, \text{false}, do(a, s)) \quad (6.9)$$

□

Examples are now in the form of *caused/3* atoms, so that each example represents now the truth value and the causality that were before represented separately by *holds/3* and *affects/3*. A difference with respect to the *affects/3* predicate is that the latter refers to the situation where the action is executed, whereas *caused/3* refers to the resulting situation. Thus, *caused/3* can be used to represent both direct and indirect effects by using an argument situation in the form *S* or *do(a, S)*. Note that the reification of the truth value in the predicate *Caused* is needed to distinguish “caused not to hold” from “not caused to hold”. Using axioms in the form 6.5 in addition to effect axioms rather than trying to give all the effects of actions directly is essential for the modularity of a representation [26]. Additionally, the use of causality allows a clear distinction between caused values and inertia values, so that we distinguish between the claim that *a* is true and the stronger claim that there is a cause for it to be true.

6.3 Indirect effects = effects propagation

Indirect effects actually represent a *propagation of changes* with respect to domain constraints. Thus, we have that:

- i) *Only those fluents that change simultaneously with another fluent can indirectly affect to each other.*

As a consequence:

- ii) *When a single fluent changes after executing an action, then at least an effect axiom must be included for that fluent and that action (e.g., $\text{toggle}(sw_1)$ with sw_1 and $\text{toggle}(sw_3)$ with sw_3).*

Obviously, when a single effect is produced, this is directly caused by the action. A further consequence is:

- iii) *Causal rules can only cover those examples where the action executed produced several simultaneous effects.*

Without additional considerations, causal rules might cover examples where a single effect is produced, because *caused* is used only in the head of the rule. These single effects must not be considered when computing the coverage of causal rules, which might produce an overestimated coverage. This restriction can be incorporated into the algorithm, by allowing only direct effects for those seeds where a single fluent changes. Just a simple restriction can avoid undesired results. For instance, when sw_1 is opened and sw_2 was open, the following clause will not be considered because sw_1 is the only fluent to change, so that it can never be an indirect effect of *light*. However, the clause is still possible provided *light* was on before opening sw_1 .

$$\text{Caused}(\text{closed}(sw_1), \text{false}, A) \leftarrow \neg \text{Holds}(\text{active}(\text{light}), A).$$

On the other hand, indirect effects represent a *propagation of changes*. As a consequence, the previous conditions are not strong enough, because the body of a causal rule might include only fluents that did not change. This might produce invalid theories, for instance, if there is a fluent that never changes with *relay* but that correlates exactly with the positive value of *relay* in the training examples, the algorithm will prefer it rather than $sw_1 \wedge sw_3$, and however there is no propagation of effects. Thus, we can safely establish that:

- iv) *In any causal rule, at least one of the fluents in the body must have changed simultaneously with the fluent in the head.*

This can be also incorporated in the algorithm, such that, when a causal rule is found that is a solution, this is removed if it does not fit the requirement. Note that this pruning cannot be done during the intermediate stages of the specialization, given that some conditions that will appear in the final clause will be missing.

However, given a particular set of observations, we have also that:

- v) *A fluent A can depend on a fluent B even if they were never observed to change simultaneously.*

This could happen if, e.g., *light* is always changed through sw_1 in the examples, however, *light* still depends on sw_2 . Thus, if we consider strictly condition *iv*, the solution might be removed from the search space before starting. In an ideal case where we have all possible sets of fluents that may change simultaneously in any situation, we could theoretically discard all those pairs of fluents that are never observed to change simultaneously, e.g., sw_1 and sw_3 do not affect each other. On the other hand, it is not difficult to run into scenarios where two fluents never change simultaneously and however one depends on the other. Let us consider two gear wheels that can be turning or static. The wheels can be pushed independently of each other, such that, when they are *coupled*, both wheels are in the same state. If we consider that the wheels cannot

be (de)coupled when any of them is turning, then a change in the fluent *coupled*, when both wheels are static, will not have any visible effect on the state of the wheels –i.e., there will be no group of changes $\{turn(wheel_1), turn(wheel_2), coupled\}$ – and however, the fluent *coupled* still affects the behavior of the wheels, and it is needed to predict indirectly the state of the wheels.

6.4 Learning Action Theories with indirect effects

The previous chapter forces the explicit representation of all the effects of an action as direct effects, producing the Ramification problem. Furthermore, the ramification problem produces additional effects during learning. The need to anticipate the *ramifications* of the executed action, causes that the learner has to produce a high number of clauses based possibly on little evidence. As a consequence, the learned hypotheses may be unnecessarily complex and thereby also less reliable and accurate. Unlike this, causal laws can make programs sensibly shorter, which have a positive influence on their learnability, as the difficulty of learning a given logic program is very much related to its length.

The learner must now infer how properties of a domain are (directly/indirectly) affected by the execution of actions, or otherwise are subject to the general law of inertia. According to the previous section, we adapt the definition 4.2.

Definition 6.3 (Learning SC programs with indirect effects)

Given:

- A *domain description* consisting of two nonempty sets: a set \mathcal{F} of fluent names, and a set \mathcal{A} of action names.
- A set \mathcal{N} of narratives \mathcal{N}_k each starting at a situation s_0^k .
- A set $E^+ = \cup_{i=1}^n E_{f_i}^+$ of examples (ground facts) *caused*($f_i, \text{true}, do(a_j, s)$), representing observations where a fluent $f_i \in \mathcal{F}$ changed from false to true after executing an action $a_j \in \mathcal{A}$ at situation s .
- A set $E^- = \cup_{i=1}^m E_{f_i}^-$ of examples (ground facts) *caused*($f_i, \text{false}, do(a_j, s)$) representing observations where a fluent $f_i \in \mathcal{F}$ changed from true to false after executing an action $a_j \in \mathcal{A}$ at situation s .
- A set $E^{inertia} = \cup_{i=1}^{n'} E_{f_i}^{inertia}$ of ground facts *caused*($f_i, v, do(a_j, s)$) representing observations where a fluent $f_i \in \mathcal{F}$ did *not* change after executing an action $a_j \in \mathcal{A}$ at situation s .
- Background knowledge (BK), including *holds/3* ground facts for all fluents at the initial situations s_0^k , the universal inertia axiom (eq. 6.6 and 6.7) and axioms (eq. 6.8 and 6.9).

Find a Situation Calculus program $H^+ = \cup_{i=1}^{n_1} H_{f_i}^+$, and respectively $H^- = \cup_{i=1}^{n_2} H_{f_i}^-$ composed of axioms in the form (6.4 and 6.5), such that:

$$(\forall e^+ \in E^+) \quad BK \cup H^+ \cup H^- \models e^+ \quad (6.10)$$

$$(\forall e^- \in E^-) \quad BK \cup H^+ \cup H^- \models e^- \quad (6.11)$$

and respectively

$$(\forall e^- \in E^-) \quad BK \cup H^+ \cup H^- \not\models e^- \quad (6.12)$$

$$(\forall e^+ \in E^+) \quad BK \cup H^+ \cup H^- \not\models e^+ \quad (6.13)$$

and

$$(\forall e \in E^{inertia}) \quad BK \cup H^+ \cup H^- \not\models e \quad (6.14)$$

□

Observations for the circuit of example 2 can be represented similarly as:

```
holds(closed(sw1),-,s0).
holds(closed(sw2),-,s0).
holds(closed(light),-,s0).

caused(closed(sw1),+,do(toggle(sw1),s0)).
caused(closed(sw2),+,do(toggle(sw2),do(toggle(sw1),s0))).
caused(active(light),+,do(toggle(sw2),do(toggle(sw1),s0))).
...
```

The form of the clauses to be learned must be also changed, because with the normal bias we cannot refer to other effects in the current situation.

```
bias(caused(active(#),#,+),[holds(closed(#),#,+),holds(active(#),#,+)]).
bias(caused(closed(#),#,+),[holds(closed(#),#,+),holds(active(#),#,+)]).
```

Care must be taken to ensure that the addition of indirect effects to a Situation Calculus program does not cause a *mutual recursion* [108] (this will be dealt with in section 6.6). The *prune/2* statement below avoids obvious non-terminating clauses.

```
prune(Head,Body) :-
    Head=caused(F,V,S),
    in(holds(F,V,S),Body).
```

This pruning clause simplifies bias declarations, as we can express a single *bias/2* declaration for all the switches rather than enumerating the valid instantiations of *closed(#)* for each *bias/2* statement of each switch.

We repeated the learning process where *LRAC* tried to learn every fluent as an indirect effect of other fluents and found:

```
caused(active(light),+,A) :- holds(closed(sw1),+,A), holds(closed(sw2),+,A).
caused(active(light),-,A) :- holds(closed(sw1),-,A).
caused(active(light),-,A) :- holds(closed(sw2),-,A).
```

In this case, we obtained a complete solution where *light* is an *indirect effect* of the switches and these affect *light* not directly but through actions that modify them (*toggle(sw₁)*, *toggle(sw₂)*). The first clause states that the lamp is controlled by two switches, i.e., the lamp is active whenever both *sw₁* and *sw₂* hold simultaneously and it subsumes the corresponding effect axioms. Notice that the first learned causal rule together with the axioms (6.8) and (6.9) entail the corresponding domain constraint, i.e. they can be used to derive the indirect effects of actions. However, forcing indirect effects in other fluents may produce odd generalizations or even no compression at all. For instance, clauses found for both *sw₁* and *sw₂* are not complete and do not even subsume any effect axiom.

```
caused(closed(sw1),+,A) :-
    holds(active(light),+,A).
caused(closed(sw1),-,A) :-
    holds(closed(sw2),+,A),
    holds(active(light),-,A).
caused(closed(sw2),+,A) :-
```

```

holds(active(light),+,A).
caused(closed(sw2),-,A) :-
  holds(closed(sw1),+,A),
  holds(active(light),-,A).

/* examples not generalized */
caused(sw1,+,do(toggle(sw1),s0)).
caused(sw2,+,do(toggle(sw2),s0)).
...

```

This is the expected result because the current state of the switches cannot be inferred just from the resulting situation but from the previous one. The natural choice is that a switch does not depend directly on any other switch. The non-generalized examples are returned together with the learned clauses. As a result, only *light* can be modeled naturally (and totally) as an indirect effect. In this case, there are also two effect axioms for the negative value of *light* containing a single condition, thus, learning *light* as a direct effect or as an indirect effect does not affect the size of the theory.

In general, we have that actions can have multiple direct and indirect effects, and that fluents can be a direct (resp. indirect) effect of multiple actions and possibly of the same action under different conditions. To actually find the right set of axioms, we could learn direct and indirect effects separately, take all learned clauses, and search all subsets of clauses that are complete with, e.g., a preference for smaller theories. As an additional condition, every clause must cover an example not covered by the rest of clauses in the subset. Unfortunately, this constitutes a *covering problem* where the worst-case complexity is $O(c^n)$ where c is the number of clauses and n is the largest number of clauses allowed in the solution. Furthermore, a bad selection of the first clauses can cause a “snowballing” effect over subsequent clauses in the cover.

A simpler possibility we have considered is to allow the inductor to determine at each step whether a fluent should be learned as a direct or an indirect effect, thus learning possibly a mix of axioms. In this way, the decision of using direct or indirect effects is done heuristically. For this task, we need to perform a *multiple search*, such that two bottom clauses are built (Fig. 6.1).

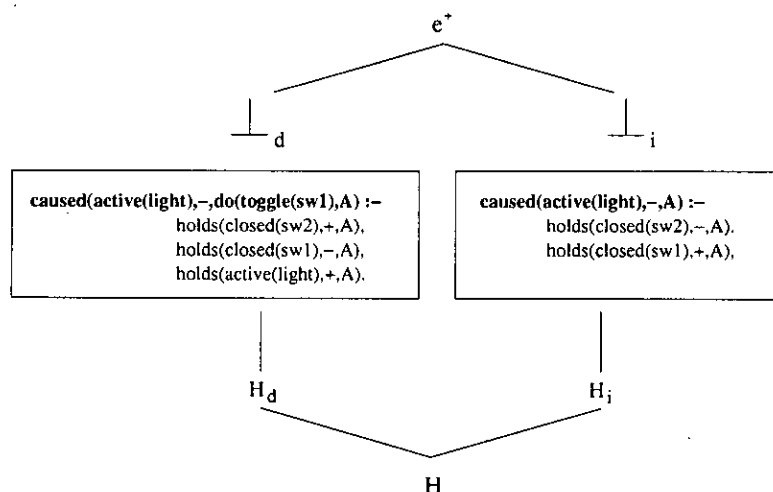


Figure 6.1: Building a bottom clause for direct and indirect effects

LRAC will take the best of clauses returned by each search according to the Progol heuristic

measure. For this task, multiple *bias/2* statements must be provided so that the inductor will take the appropriate one for each search. For instance:

```
bias(caused(closed(#),#,do(#,+)),[holds(closed(#),#,+),holds(active(#),#,+)]).
bias(caused(closed(#),#,+),[holds(closed(#),#,+),holds(active(#),#,+)]).
bias(caused(active(#),#,do(#,+)),[holds(closed(#),#,+),holds(active(#),#,+)]).
bias(caused(active(#),#,+),[holds(closed(#),#,+),holds(active(#),#,+)]).
```

Under these conditions *LRAC* returned the following clauses:

```
caused(closed(sw1),+,do(toggle(sw1),A)) :-
    holds(closed(sw1),-,A).
caused(closed(sw1),-,do(toggle(sw1),A)) :-
    holds(closed(sw1),+,A).
caused(closed(sw2),+,do(toggle(sw2),A)) :-
    holds(closed(sw2),-,A).
caused(closed(sw2),-,do(toggle(sw2),A)) :-
    holds(closed(sw2),+,A).
caused(active(light),+,A) :-
    holds(closed(sw1),+,A),
    holds(closed(sw2),+,A).
caused(active(light),-,A) :-
    holds(closed(sw1),-,A).
caused(active(light),-,A) :-
    holds(closed(sw2),-,A).
```

that correspond to the intended description of the domain. The impact on learnability is not quite noticeable (just one clause less than with direct effects) given the small size of the domain.

6.4.1 A more complex circuit

The increased complexity of axiom effects is a consequence of the fact that they need to anticipate the *ramifications* of the executed action, and these become increasingly numerous and complicated as the complexity of the domain increases. Suppose a third switch is introduced, named *sw₃*, plus a *relay*.

Example 3 (Thielscher's circuit (a)) A simple circuit that includes a lamp, a relay, and three switches *sw₁*, *sw₂* and *sw₃*, together with some actions in the form *toggle(sw_i)*. □

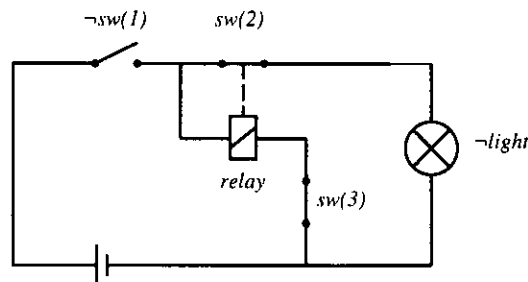


Figure 6.2: An extended electric circuit

We generated a training set consisting of 173 situations with 270 positive examples and 1460 negative examples. By considering only direct effects, *LRAC* obtained the following clauses:

```

caused(closed(sw1),-,do(toggle(sw1),A)) :-
    holds(closed(sw1),+,A).
caused(closed(sw1),+,do(toggle(sw1),A)) :-
    holds(closed(sw1),-,A).
caused(closed(sw2),-,do(toggle(sw2),A)) :-
    holds(closed(sw2),+,A).
caused(closed(sw2),-,do(toggle(sw3),A)) :-
    holds(active(light),+,A).
caused(closed(sw2),-,do(toggle(sw1),A)) :-
    holds(closed(sw2),+,A),
    holds(closed(sw3),+,A).
caused(closed(sw2),+,do(toggle(sw2),A)) :-
    holds(closed(sw2),-,A),
    holds(active(relay),-,A).
caused(closed(sw3),-,do(toggle(sw3),A)) :-
    holds(closed(sw3),+,A).
caused(closed(sw3),+,do(toggle(sw3),A)) :-
    holds(closed(sw3),-,A).
caused(active(light),-,do(toggle(sw1),A)) :-
    holds(active(light),+,A).
caused(active(light),-,do(toggle(sw2),A)) :-
    holds(active(light),+,A).
caused(active(light),-,do(toggle(sw3),A)) :-
    holds(active(light),+,A).
caused(active(light),+,do(toggle(sw1),A)) :-
    holds(closed(sw1),-,A),
    holds(closed(sw2),+,A).
caused(active(light),+,do(toggle(sw2),A)) :-
    holds(closed(sw1),+,A),
    holds(closed(sw2),-,A).
caused(active(relay),+,do(toggle(sw1),A)) :-
    holds(closed(sw3),+,A),
    holds(active(relay),-,A).
caused(active(relay),+,do(toggle(sw3),A)) :-
    holds(closed(sw1),+,A),
    holds(closed(sw3),-,A).
caused(active(relay),-,do(toggle(sw1),A)) :-
    holds(closed(sw1),+,A).
caused(active(relay),-,do(toggle(sw3),A)) :-
    holds(closed(sw3),+,A).

```

According to the rules, action *toggle*(*sw*₁) (resp. *toggle*(*sw*₃)) toggles switch *sw*₁ (resp. *sw*₃). For the rest of fluents, the system learned 5 clauses for *light*, 4 for *relay* and 4 for *sw*₂ –where all actions *toggle*(*sw*_{*i*}) affect them. The dependency graph is shown in Fig. 6.3.

In this case, forcing all effects as indirect, *LRAC* learned:

```

caused(active(light),+,A) :-
    holds(closed(sw1),+,A),
    holds(closed(sw2),+,A).
caused(closed(light),-,A) :-

```

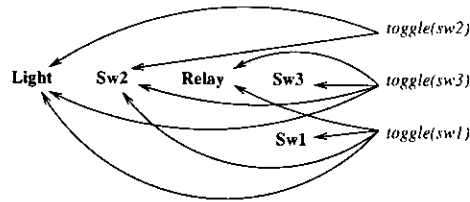


Figure 6.3: Direct effects in the Thielscher's circuit

```

holds(closed(sw1),-,A).
caused(closed(light),-,A) :-
  holds(closed(sw2),-,A).
caused(closed(relay),+,A) :-
  holds(closed(sw1),+,A),
  holds(closed(sw3),+,A).
caused(closed(relay),-,A) :-
  holds(closed(sw1),-,A).
caused(closed(relay),-,A) :-
  holds(closed(sw3),-,A).

```

For *light* and *relay* we obtained a complete solution. The *relay* is controlled by switches sw_1 and sw_3 , i.e., the relay is active whenever both sw_1 and sw_3 hold simultaneously. Similarly for *light* with sw_1 and sw_2 . As in the previous circuit, no complete solution was found for sw_1 and sw_3 . In some cases they produced odd generalizations, in others no compression was achieved or even no consistent clause could be found. Fluent sw_2 is more problematic because no complete set of clauses were found, by considering strictly indirect effects, and however some clauses individually were meaningful and subsumed some of the effect axioms.

When we allowed *LRAC* to determine whether an effect is a direct effect or an indirect effect, we obtained the following theory:

```

caused(closed(sw1),-,do(toggle(sw1),A)) :-
  holds(closed(sw1),+,A).
caused(closed(sw1),+,do(toggle(sw1),A)) :-
  holds(closed(sw1),-,A).
caused(closed(sw2),+,do(toggle(sw2),A)) :-
  holds(closed(sw2),-,A),
  holds(active(relay),-,A).
caused(closed(sw2),-,A) :-
  holds(active(relay),+,A).
caused(closed(sw2),-,do(toggle(sw2),A)) :-
  holds(closed(sw2),+,A).
caused(closed(sw3),-,do(toggle(sw3),A)) :-
  holds(closed(sw3),+,A).
caused(closed(sw3),+,do(toggle(sw3),A)) :-
  holds(closed(sw3),-,A).
caused(active(light),+,A) :-
  holds(closed(sw1),+,A),
  holds(closed(sw2),+,A).
caused(closed(light),-,A) :-
  holds(closed(sw1),-,A).
caused(closed(light),-,A) :-

```

```

holds(closed(sw2),-,A).
caused(closed(relay),+,A):-
  holds(closed(sw1),+,A),
  holds(closed(sw3),+,A).
caused(closed(relay),-,A):-
  holds(closed(sw1),-,A).
caused(closed(relay),-,A):-
  holds(closed(sw3),-,A).

```

According to the rules, *light* and *relay* are still considered indirect effects whereas *sw₁* and *sw₃* are direct effects. However, *sw₂* is in some cases a direct effect and in other cases an indirect effect. The difference in the size of the theory learned with respect to the theory with only direct effects is significant (table 6.1).

Fluent	direct	direct/indirect
<i>sw₂</i>	4	3
<i>relay</i>	4	3
<i>light</i>	5	3

Table 6.1: Number of clauses for direct and indirect effects in example 3

Note that the causal rule for the negative value of *sw₂* does not contradict the effect axiom—in fact, the condition of the causal rule for the negative value appears negated in the effect axiom for the positive value¹. The new dependency graph is showed in Fig. 6.4.

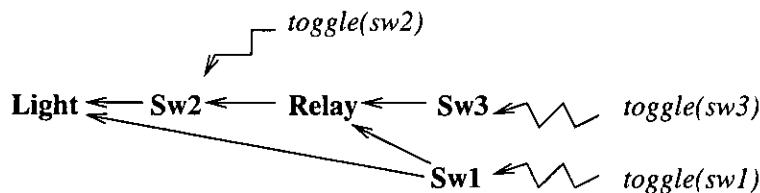


Figure 6.4: Direct and indirect effects in the Thielscher's circuit

During learning, the inductor must prefer direct over indirect effects (and viceversa). In general, when multiple effects co-occur, it is not clear which fluents are connected to actions and which are fluent-triggered. The problem is even harder if actions can be executed concurrently (section 8.1), because fluents that change simultaneously can be direct effects of different actions. *LRAC* uses by default the compression measure of Progol which biases the learner to prefer smaller theories. The insight is that indirect effects can make programs sensibly shorter, given that a causal rule may subsume the direct effects of several actions.

In some cases, one causal rule subsumes completely the effect axioms (e.g., *light*) whereas in other cases only partially (e.g., *sw₂*). With regard to the positive value of *relay* (resp. *light*), a causal rule is enough whereas two effect axioms are needed (resp. 3 for *light*). For the negative value, *relay* needs two clauses in any case. The negative value of *sw₂* is in some cases a direct effect of *toggle(sw₂)* and in other cases an indirect effect of *toggle(sw₁)* and *toggle(sw₃)*. For the latter case, indirect effects will be preferred as just one clause is required ($\overline{sw_2} \leftarrow relay$).

¹However, contradiction would be possible if multiple actions are executed concurrently (section 8.1).

Additionally, *LRAC* will prefer the latter to $\overline{sw_2} \leftarrow sw_1, sw_3$, because although they cover the same number of examples the former has length 1. In these cases, the same coverage can be achieved with much less causal rules, thus, if the causal rule is in the search space, *LRAC* will always prefer it. Unlike this, the relation $light \rightarrow sw_1$ was not learned instead of the direct effect, given that, although the relation is true, it just covers a fraction of the examples for sw_1 whereas the direct effect covers all of them. And similarly for $sw_1 \leftarrow relay$, and so on.

In practice, *LRAC* uses implicitly the empirical frequencies observed in the training set (an approximation to the conditional distribution $Pr(F|F')$ where F denotes a change in a fluent). In this case, the training set might include only certain changes thus biasing the above process. For instance, if whenever the *relay* becomes active, the switch sw_2 is opened, then it would be possible to learn the relation $relay \leftarrow \overline{sw_2}$. If the above case appears when the *relay* is made active through $toggle(sw_3)$ –thus, *light* becomes also false– then it would be possible to learn the relation $\overline{sw_2} \leftarrow light$, or even $light \leftarrow relay$. Similarly, if whenever the switch sw_1 is closed, the switch sw_2 was closed and consequently the *light* becomes active, then it would be possible to learn the relation $sw_1 \leftarrow light$. In this case, sw_1 is considered an indirect effect of $toggle(sw_1)$.

In general, identifying indirect effects in a set of observations is intrinsically complex. If the propagation of effects has observable delays, these could help by using the sequence of changes, given that most definitions about causality require a precedence of causes over effects. For instance, sw_1 will never change after *light*. Despite this, two effects can be direct effects of an action and still have different delays, hence, the observed effect propagation may not correspond to the causality relations. However, to be able to observe small delays requires in many cases advanced sensory capabilities. In practice, compression-based measures seem an adequate estimator and correspond to the initial intuition that causal rules make theories sensibly shorter. The experiments carried out, although not exhaustive, seem to confirm it.

6.4.2 Causality-based approaches

From definition 4.2, “caused” is associated to a change of value in a fluent. However, action formalisms provide different meanings for “caused” other than simply “change of value” which introduces a particular feature of causal rules wrt. effect axioms in learning indirect effects, that is clear in example (3). The clause:

```
caused(closed(sw2),-,A) :- holds(active(relay),+,A).
```

asserts that sw_2 is caused to be open whenever the *relay* is active. But let us consider a situation where sw_2 is open and is toggled. The *relay* avoids the switch to close (and thus sw_2 persists), however the clause infers that the negative value of sw_2 is again caused (inertia does not apply). As a consequence, a number of “useless” (even if correct) instances of *caused/3* can be derived for any situation where *relay* is true but not caused. Useless here just means that it does not give rise to any changes, just maintains a reason for sw_2 being false: Note that this problem does not actually appear for the positive value of *light* in example (2), because there are no actions not affecting *light*, i.e., the positive value does never persist for *light*, however, it does when other actions are included that does not affect *light*.

This introduces some non-desired situations for reasoning [101]. It would be possible to include a *caused/3* atom in the body that acts as a trigger of the causal rule, i.e., a causal rule needs one of the conditions in the body to be caused, solving the “extra” caused atoms generated. When the body includes multiple literals, we just need one of them to be caused.

Unfortunately, this results in as many causal rules as literals in the body, when implemented in Logic Programming.

```
caused(active(light),+,A) :- caused(closed(sw1),+,A), holds(closed(sw2),+,A).
caused(active(light),+,A) :- holds(closed(sw1),+,A), caused(closed(sw2),+,A).
```

Furthermore, for the negative value of *light*, this approach still generates additional *caused/3* atoms, because *light* does not change its value after opening *sw₁* when *sw₂* was open.

```
caused(active(light),-,A) :- caused(closed(sw1),-,A).
```

We could add a condition to express that light has actually changed, e.g., `holds(sw2,+,A)`, thus, *caused* actually represents that a fluent changed its value. However, this cannot be generally applied. A counterexample is provided by

```
caused(closed(sw2),-,A) :- caused(active(relay),+,A)
```

where no condition –given the form of the causal rules– can be added to express that *sw₂* was previously true. We would need to refer to the value of *sw₂* at the previous situation. In general, we need that the body of the clause, taken as a formula, changes to infer that the fluent in the head is caused [26].

In most action formalisms, these extra *caused* atoms are labeled with the meaning “there is a reason other than inertia”, however, this semantics cannot be adopted during learning, because it obviously contradicts the observations given that negative examples for `caused(closed(sw2),-,S)` are given for those situations where *sw₂* is closed or it does not change². A simple solution is to allow these some extra observations to be caused by adding an internal control, so that, causal rules are not tested on those negative examples provided the latter are negative only for the causality and there is no inconsistency on the truth value. Thus, given an example `caused(f,v,do(a,s))`, only those negative examples where both `holds(f,v',s)` and `holds(f,v',do(a,s))`, are considered (*v'* is the inverse of *v*). The learned theory still produces extra *caused/3* atoms, however these do not affect the learning process.

6.4.3 The Blocks world (contd.)

Recall the *blocks world* domain. We repeated the learning process by allowing both causal rules and effect axioms and *LRAC* obtained:

```
caused(clear(A),+,do(move(C,D),B)) :-
  holds(clear(D),+,B),
  holds(clear(C),+,B),
  holds(on(C,A),+,B),
  diff(D,A).
caused(clear(A),+,do(move(C,D),B)) :-
  holds(clear(C),+,B),
  holds(on(C,A),+,B),
  table(D).
caused(clear(B),-,A) :-
  holds(on(_,B),+,A).
```

²Furthermore, these “extra” atoms may affect learning provided we needed to use *caused/3* in the body of a clause (see section 8.2).


```

caused(on(A,B),+,do(move(A,B),C)) :-
    holds(clear(A),+,C),
    holds(clear(B),+,C).
caused(on(A,B),+,do(move(A,B),C)) :-
    holds(clear(A),+,C),
    table(B).
caused(on(A,B),-,C) :-
    holds(on(A,D),+,C),
    diff(B,D).
    
```

The negative value of *clear*/1 and the negative value of *on*/2 can be indirectly predicted through the positive value of *on*/2 (Fig. 6.5). The use of indirect effects produces a shorter theory for the negative value of *on*/2, even with a single action *move*/2. Two effect axioms would be needed to deal separately with the cases where a block is moved onto other block or onto the table. The domain constraint represents that *on*/2 is a *functional fluent*, where the location is at most one for each block, so that when a block is on location *L*, then it is not on every other location different from *L*. However, the domain constraint for the negative value of *clear*/1 does not produce a clear benefit mainly because there are no other actions that can affect it apart from *move*/2. However, as the domain constraint is shorter, the system is biased to prefer it.

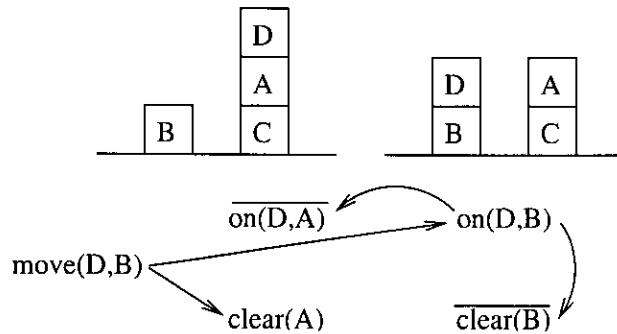


Figure 6.5: Indirect effects in the Blocks world

Actually the positive value of *clear*/2 is also an indirect effect, because a block is clear if no block is above it. However, it is learned as a direct effect of the *move*/2 action because variables in the body but not in the head are existentially quantified.

6.4.4 The Logistics domain

Let us consider the *Logistics* domain which has been considered a benchmark domain in the planning literature.

Example 4 (Logistics domain) *In this domain we have two types of vehicles: trucks and airplanes. The problems in this domain typically start off with a collection of objects at various locations in various cities, and the goal is to redistribute these objects to their new locations. The basic actions are loading and unloading packages from vehicles, driving trucks and flying airplanes.*

□

The domain contains the following actions and fluents:

```

action.lot(P,T) :- package(P),truck(T).
action.unlot(P,T) :- package(P),truck(T).
action.loa(P,A) :- package(P),airplane(A).
action.unloa(P,A) :- package(P),airplane(A).
action.drive(T,LF,LT) :- truck(T),location(LF),location(LT).
action.fly(A,LF,LT) :- airplane(A),airport(LF),airport(LT).

fluent.at(P,L) :- physobj(P),location(L).
fluent.in(P,V) :- package(P),vehicle(V).

```

Background includes domain predicates *truck/1*, *airplane/1*, *location/1*, *airport/1*, *city/1*, *physobj/1* and *package/1*, that indicate the type of objects (where some objects may belong to more than one type), actions to load and unload packages from trucks (*lot/2*, *unlot/2*), and from planes (*loa/2*, *unloa/2*), drive trucks (*drive/3*) and fly planes (*fly/3*), and fluents *at/2* to indicate the location of packages and vehicles, and *in/2* to indicate that some objects are in some vehicles. We used an scenario with three cities and three locations by city (including an airport), and with a single plane and a single truck by each city. Trucks and planes are subtypes of *vehicle* whereas vehicles and packages are physical objects.

```

location(bos_po;pgh_po;la_po;bos_central;pgh_central;la_central).
location(A) :- airport(A).
airport(bos_airport;pgh_airport;la_airport).
city(pgh;bos;la).
package(pkg1;pkg2;pkg3;pkg4).
truck(pgh_truck;bos_truck;la_truck).
airplane(apn1;apn2).

physobj(A) :- package(A).
physobj(A) :- vehicle(A).
vehicle(A) :- truck(A).
vehicle(A) :- airplane(A).

```

Additionally, the following static predicates are used: *incity(loc,city)* to indicate in which city a package, location or vehicle resides, and *diff/2*.

```

incity(pgh_po,pgh).
incity(bos_po,bos).
incity(la_po,la).
incity(pgh_airport,pgh).
incity(bos_airport,bos).
incity(la_airport,la).
incity(pgh_central,pgh).
incity(bos_central,bos).
incity(la_central,la).

```

The training set consisted of 155 situations from which *LRAC* returned:

```

caused(at(A,B),+,do(unlot(A,D),C)) :-
    holds(at(A,B),-,C),
    holds(in(A,D),+,C),
    holds(at(D,B),+,C).
caused(at(A,B),+,do(unloa(A,D),C)) :-

```

```

holds(at(A,B),-,C),
holds(in(A,D),+,C),
holds(at(D,B),+,C).
caused(at(B,_),-,A) :-
  holds(in(B,_),+,A).
caused(at(A,B),+,do(drive(A,D,B),C)) :-
  holds(at(A,B),-,C),
  holds(at(A,D),+,C).
caused(at(B,C),-,A) :-
  holds(at(B,D),+,A),
  diff(D,C).
caused(at(A,B),+,do(fly(A,D,B),C)) :-
  holds(at(A,B),-,C),
  holds(at(A,D),+,C).
caused(in(B,_),-,A) :-
  holds(at(B,_),+,A).
caused(in(A,B),+,do(put(A,B),C)) :-
  holds(in(A,B),-,C),
  holds(at(A,D),+,C),
  holds(at(B,D),+,C).

```

According to the rules, a package can be loaded into a vehicle when both are at the same location, and it can be unloaded from the vehicle where it is in. Trucks can be used to transport goods within a city, and airplanes can be used to transport goods between two cities. The causal laws express that a package cannot be at two locations at the same time or at a location and into a vehicle, independently of the actions that were executed. Note that the following causal rule is not learned:

```

caused(in(B,C),-,A) :-
  holds(in(B,D),+,A),
  diff(D,C).

```

because a package cannot be moved directly from one vehicle to another one, i.e., $\neg in(B,C)$ and $in(B,D)$ never change simultaneously.

The Logistics domain is considered a benchmark for planning methods. In languages used for planning, like PDDL, that has its origins in STRIPS, the effects of actions are still represented as direct effects, for instance:

```

(:action LOAD-AIRPLANE
 :parameters
  (?obj
   ?airplane
   ?loc)
 :precondition
  (and (OBJ ?obj) (AIRPLANE ?airplane) (LOCATION ?loc)
   (at ?obj ?loc) (at ?airplane ?loc))
 :effect
  (and (not (at ?obj ?loc)) (in ?obj ?airplane)))

```

whereas the domain constraints learned correspond to so-called *invariants* in the planning literature, that are used to assist the planner in exploiting the structure that is inherent in the domain, and that often reduce the search [40, 114].

```
FORALL y1. FORALL z1. at(x,y1) AND at(x,z1) => y1=z1
NOT (Exists y1:location. at(x,y1) AND Exists y1:vehicle. in(x,y1))
```

6.5 Relation to integrity constraints

Integrity constraints in Logic Programming are clauses with the special atom *false* in the head, representing conditions that render the theory inconsistent if they become provable. A. Kakas et al. [28] introduced the use of such integrity constraints as a means of specialization in learning, which has a number of advantages, particularly when the given learning problem is incompletely specified. The idea consists in using constraints to specialize overgeneral rules. For instance, in the circuit 3, we can find a similar definition for the positive value of sw_2 , where an overgeneral effect axiom is specialized by an integrity constraint.

```
caused(closed(sw2),+,do(toggle(sw2),A)) :-
    holds(closed(sw2),-,A).

:- holds(closed(sw2),+,A), holds(active(relay),+,A).
```

In this case, the *relay* acts as a domain constraint forbidding some next states. The effect axiom and the constraint are executed in two stages, such that the constraint prunes the resulting situation when this does not satisfy the constraint. If the above integrity constraint is considered simply as another clause in the theory, this will have no effect on the effect axiom which will remain overgeneral.

An advantage over the classical specialization method of adding literals, is that integrity constraints can sometimes provide implicitly the required specialization of the effect axioms without the need of explicit negative training data. For instance, the above constraint can exist independently of any negative examples, i.e., sw_2 can never be closed when the relay is active.

The integrity constraints are not simply some extra clauses of the theory learned on some additional concepts. Although the integrity constraints could be used to provide a partial definition for these predicates, in some cases, their main purpose is to specialize the effect axioms. Thus, in general, these integrity constraints cannot substitute the causal rules, because the former serve only to prune models but not to infer the value of a fluent. For instance, the constraint above does not infer that sw_2 will be opened when the relay is active. As a counterpart, the effect axioms and the constraints cannot be learned separately, and the choice of the integrity constraints is not independent from the rules of the theory and part of the difficulty is to find the “relevant” constraints that would compensate correctly for the rules of the theory.

The use of explicit constraints provides sometimes more expressivity, because the use of negation allows universal quantification in the body of clauses. For instance, an action’s precondition where all the blocks must be on the table, cannot be expressed with normal logic programs, however it can be expressed with the following constraint:

```
:- ..., holds(on(B,C),+,A),diff(C,table).
```

Constraints can be also used to express complex qualifications, e.g., constraints about the resulting situation. For instance, the constraint that a player may not be in check after his own move, represents an implicit precondition on possible moves, and it would be very difficult to express this action precondition explicitly as a condition on the starting state from which the player makes the move [26]. Unlike this, in the following theory:

```

caused(in(P,Row,Col),+,do(move(P,Row,Col),A)) :-
    holds(valid(move(P,Row,Col)),+,A).
:- holds(incheck,+,do(move(P,Row,Col),A)).

```

the constraint specializes the overgeneral effect axiom, so that only valid and legal moves are allowed. The fluent *incheck* does not depend on how the check is produced, which provides with a much simpler definition. Furthermore, the constraint represents a general rule in chess games that exists independently of any negative examples.

It would be possible to use a mixed axiom where a constraint and an effect axiom are integrated in a single axiom (*inter-constraint*). For instance, in the example 3 we have:

```

caused(closed(sw2),+,do(toggle(sw2),A)) :-
    holds(closed(sw2),-,A),
    holds(active(relay),-,do(toggle(sw2),A)).

```

such that the value of *relay* refers to the resulting situation. However, note that this kind of axioms only work when the constraint is not affected by the action, i.e., the relay does not need the result of the effect axiom (it is not affected by *toggle(sw2)*), otherwise the effect axiom includes a mutual recursion. For instance, in the chess game, the check depends on the position after moving the piece, and the executability of the movement depends on the check. However, it can be interesting when actions can be executed concurrently (section 8.1).

```

caused(in(P,Row,Col),+,do(move(P,Row,Col),A)) :-
    holds(valid(move(P,Row,Col)),+,A),
    holds(incheck,-,do(move(P,Row,Col),A)).

```

In this case, so-called inter-constraints are not usable, whereas with the use of explicit constraints, the overgeneral effect axiom is executed first, and only then the constraint is applied.

6.6 Cycles

A potentially problematic form of dependency that could specially arise in the representation of physical systems is *cyclic dependences* between effects. Cyclic definitions can appear even in the description of perfectly normal, well-behaved physical systems, thus, a complete learning method must be able to learn theories containing cycles.

In the Situation Calculus, action theories are in general recursive theories where direct effects are the base cases. Intuitively, there will be no recursion if the fluents are *stratified* and the heads of clauses only ever mention fluents at the lowest level of the stratification. We can formally express the properties of well-formed action theories such that the domain progresses after the execution of actions.

- i) each effect should be caused, either by a primitive action or by another effect.
- ii) there cannot exist self-supported effects.
- iii) effect causation is well-ordered, i.e., there should not be an infinite chain of effects in which each effect is caused by the next effect in the chain.

However, care must be taken to ensure that the addition of domain constraints to a Situation Calculus program does not cause a *non-finite recursion*, where fluents may use one another in their definitions. For instance, the clauses below produce a non-terminating recursion.

```

caused(active(light),+,A) :-
    holds(closed(sw1),+,A),
    holds(closed(sw2),+,A).
caused(closed(sw1),+,A) :-
    holds(active(light),+,A).

```

Both clauses can be locally consistent and complete considered separately but their union is globally useless.

In systems adopting *extensional* evaluation, after each new rule is added to the theory, the whole theory must be evaluated globally to check if the theory is valid. On the other hand, if intensional evaluation is adopted, the clauses above are not possible, because no cyclical dependences are possibly produced, although termination must be granted in the prover in some way. In systems that use Prolog, a bound is set on the number of resolution steps of the prover, whereas in the stable models semantics, there are no such termination problems. For mutually recursive theories, the learning of one fluent must be interleaved with the learning of the other ones [33, 108], otherwise the theory will not be found (similarly to the *even/odd* domain). Furthermore, a *base case* is to be found before the recursive one, i.e., some direct effects must be learned first.

The cyclical dependence in the circuit shown before does not correspond to a real behavior of the system, because the switches modify the state of the bulb, but not viceversa. There are cases however, where a cyclical definition can be theoretically correct. Let us consider the following example including a valid positive cycle that can cause a non-finite recursion.

Example 5 (Gear wheels) Consider two connected gear wheels with actions *push* and *stop* for each wheel that make a wheel start (resp. stop) turning. \square



Figure 6.6: Gear wheels

Forcing all effects as direct effects, *LRAC* returned:

```

caused(turn(wheel1),+,do(push(wheel1),A)).
caused(turn(wheel1),+,do(push(wheel2),A)).
caused(turn(wheel1),-,do(stop(wheel1),A)).
caused(turn(wheel1),-,do(stop(wheel2),A)).
caused(turn(wheel2),+,do(push(wheel1),A)).
caused(turn(wheel2),+,do(push(wheel2),A)).
caused(turn(wheel2),-,do(stop(wheel2),A)).
caused(turn(wheel2),-,do(stop(wheel1),A)).

```

In this example, every wheel has an action associated, such that any force causing a wheel to start or stop turning propagates to the rest of wheels (and viceversa). Thus, all wheels are considered direct effects of all actions that affect any of them. When we allow indirect effects and extensional evaluation is adopted, *LRAC* easily learned the two counterparts of a *double implication*, where every wheel is both an effect and a cause.

```

caused(turn(wheel1),+,A) :-
    holds(turn(wheel2),+,A).
caused(turn(wheel1),-,A) :-
    holds(turn(wheel2),-,A).
caused(turn(wheel2),+,A) :-
    holds(turn(wheel1),+,A).
caused(turn(wheel2),-,A) :-
    holds(turn(wheel1),-,A).

```

The learned theory has 4 causal rules whereas it requires 8 effect axioms, hence, *LRAC* will always prefer indirect effects. With extensional evaluation, the previous theory is complete and consistent because the positive examples are always added to the model, however, the definition is useless because there is no connection to actions, i.e., it cannot be used to derive conclusions. Progol detects such incompleteness (step 6 of Fig. 5.3) however the cycle avoids further rules to be learned. To avoid theories that are extensionally complete but intensionally incomplete, then some of the methods considered in section 4.3.4 must be applied to make it complete, e.g, some of the clauses must be explicitly removed to avoid the cycle. However, we have used a different approach. In the stable model semantics, given the following theory T :

$$\begin{aligned}
 p &\leftarrow q \\
 q &\leftarrow p
 \end{aligned}$$

with a positive cycle whose unique stable model is the empty model $\{\}$, we can find a theory T' by adding some rules to T that somewhat “complete” T , for instance, adding the facts p or q or some rules for them but based on a third-party atom, e.g., $p \leftarrow r$. In the wheels example we just have to prove the *necessary* facts, without removing the cycle, to make the theory complete. For this reason, we add some *effect axioms* that extensionally cover the examples that were not proved by the theory (Fig.6.7).

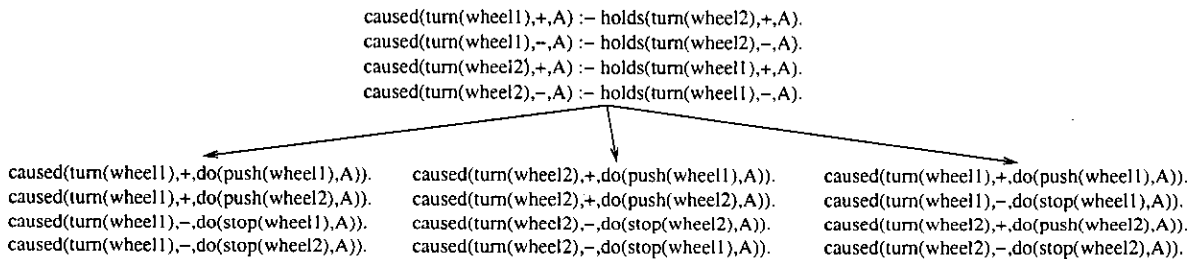


Figure 6.7: Adding direct effects to the gear wheels example

There is no a general rule as to what particular effect axioms should be added. Intuitively we should consider $turn(wheel_i)$ as a direct effect of $push(wheel_i)$ and $stop(wheel_i)$ and an indirect effect of $push(wheel_j)$ and $stop(wheel_j)$, with $j \neq i$. However, this nominal correspondence between actions and fluents cannot be assumed in general. In the worst case (the left-most side and the center of Fig. 6.7) the whole theory is re-learned containing only direct effects for one of the wheels, whereas the causal rules propagate the changes to the other wheel. Thus, the causal rules for one of the wheels become redundant. Only the set of clauses on the right-most side of the figure represents the intended theory, that is:

```

caused(turn(wheel1),+,do(push(wheel1),A)).

```

```

caused(turn(wheel1),-,do(stop(wheel1),A)).
caused(turn(wheel1),+,A):-
    holds(turn(wheel2),+,A).
caused(turn(wheel1),-,A):-
    holds(turn(wheel2),-,A).

caused(turn(wheel2),+,do(push(wheel2),A)).
caused(turn(wheel2),+,A):-
    holds(turn(wheel1),+,A).
caused(turn(wheel2),-,do(stop(wheel2),A)).
caused(turn(wheel2),-,A):-
    holds(turn(wheel1),-,A).

```

In the intended theory, $turn(wheel_1)$ is a direct effect of $push(wheel_1)$ and an indirect effect of $push(wheel_2)$ –and similarly for $turn(wheel_2)$ (Fig. 6.8).

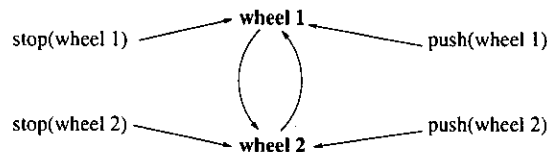


Figure 6.8: Dependences in the wheels example (a)

Thus, the causal rules are not subsumed, so that the examples not covered by the effect axiom, are covered by the causal rule. This method completes the theory, however it does not always produce the intended theory. For instance, the theory is completed only with effect axioms, however, if there is any valid causal rule that completes the theory without causing cycles, this will not be found. However, extending the search to other domain constraints introduces an additional cost because new cycles might appear.

Note that, in the normal Prolog semantics we need to impose a syntactical restriction (no cycles are allowed) that guarantees that the effects are unique and well-defined. In that case, to avoid cycles in the final theory, $turn(wheel_1)$ must be considered always a direct effect and only $turn(wheel_2)$ can be considered as an indirect effect (or viceversa). Unfortunately, this makes the learned theory suffer from the ramification problem, for those fluents where the cycle is broken (Fig. 6.9).

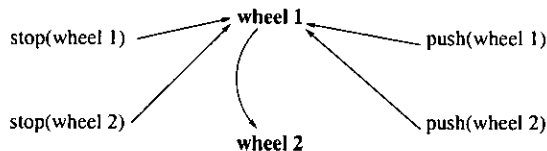


Figure 6.9: Dependences in the wheels example (b)

With intensional evaluation the multiple *spl* tasks must be interleaved, so that the first clause must correspond to a direct effect for one of the wheels. The final theory will be globally complete, however, there is still no guarantee that it corresponds to the intended one.

In practice, whether the inductor will prefer to consider one of the fluents always as a direct effect or it will learn the right theory, is based on a compression measure and depends on the particular domain. In Fig. 6.10, the first clause to be learned should be the one labeled (1), but

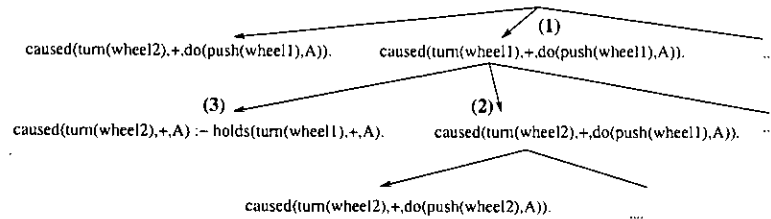


Figure 6.10: Search space using intensional coverage testing

it will be learned if it covers more examples than the one on its left –which converts an indirect effect into a direct effect³. Unfortunately, they cover the same number of examples and *LRAC* will prefer one of them without a clear criterion. The next clause to be learned is for *wheel₂*. Unfortunately, *LRAC* will prefer the effect axiom labeled (2) in Fig. 6.10 to the causal rule labeled (3) because, although both clauses cover the same number of examples, the compression measure punishes larger clauses⁴. Causal rules will be preferred when they produce a shorter theory, however, in this case, effect axioms produce the same compression and *LRAC* is biased to prefer the latter.

6.6.1 Yet another circuit

We have seen that it is possible to learn theories with cycles provided we use a semantics that allow them, however, we have no guarantee that the final theory corresponds to the intended one. In many cases, the cycles found do not correspond to actual cycles and they have to be broken. The difficulty of breaking a cycle is the same as when learning indirect effects, i.e., determine the causality in the fluents. Let us consider another circuit given in [128].

Example 6 (Thielscher’s circuit (b)) *An electric circuit that consists of a number of binary switches, two relays, three resistors, and a couple of light bulbs:*

$$\{s_1, s'_1, s_2, s'_2, s_3, s'_3, li, li_1, li_2, li_3, r_1, r_2, r_3, re_1, re_2\}$$

The various states the circuit may exhibit will be described using the unary fluent names closed and active. The first ranges over all switches, and the scope of the second are both bulbs and relays. There is only a type of action in this domain, changing the position of switches. □

LRAC returned the following theory:

```
caused(closed(s1),-,do(toggle(s1),A)) :-
    holds(closed(s1),+,A).
caused(closed(s1),+,do(toggle(s1),A)) :-
    holds(closed(s1),-,A).

caused(closed(s2),-,do(toggle(s2),A)) :-
    holds(closed(s2),+,A),
    holds(active(re1),-,A).
```

³The use of a seed in the algorithm may produce that the first clause to be learned corresponds to the action *start(wheel₂)* and the fluent *turn(wheel₂)*.

⁴The action in the effect axiom does not affect the length of the axiom.

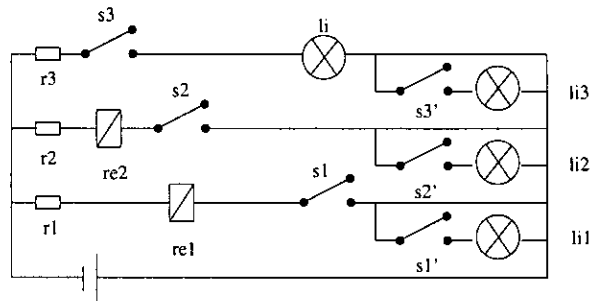


Figure 6.11: Another electric circuit

```

caused(closed(s2),+,do(toggle(s2),A)) :-
    holds(closed(s2),-,A).
caused(closed(s2),+,do(toggle(s1),A)) :-
    holds(closed(s2),-,A),
    holds(closed(s1),-,A).

caused(closed(s3),-,do(toggle(s3),A)) :-
    holds(closed(s3),+,A),
    holds(active(re2),-,A).
caused(closed(s3),+,do(toggle(s3),A)) :-
    holds(closed(s3),-,A).
caused(closed(s3),+,do(toggle(s1),A)) :-
    holds(closed(s3),-,A),
    holds(closed(s1),-,A).
caused(closed(s3),+,do(toggle(s2),A)) :-
    holds(closed(s3),-,A),
    holds(closed(s2),-,A).

caused(closed(s1p),-,do(toggle(s1p),A)) :-
    holds(closed(s1p),+,A).
caused(closed(s1p),+,do(toggle(s1p),A)) :-
    holds(closed(s1p),-,A).

caused(closed(s2p),-,do(toggle(s2p),A)) :-
    holds(closed(s2p),+,A).
caused(closed(s2p),+,do(toggle(s2p),A)) :-
    holds(closed(s2p),-,A).

caused(closed(s3p),-,do(toggle(s3p),A)) :-
    holds(closed(s3p),+,A).
caused(closed(s3p),+,do(toggle(s3p),A)) :-
    holds(closed(s3p),-,A).

caused(active(li),-,A) :-
    holds(closed(s3),-,A).
caused(active(li),+,A) :-
    holds(closed(s3),+,A).

caused(active(li1),-,do(toggle(s1),A)) :-
    holds(active(li1),+,A).

```

```

caused(active(li1),-,do(toggle(s1p),A)) :-
    holds(active(li1),+,A).
caused(active(li1),+,A) :-
    holds(closed(s1),+,A),
    holds(closed(s1p),+,A).

caused(active(li2),-,A) :-
    holds(closed(s2),-,A).
caused(active(li2),-,do(toggle(s2p),A)) :-
    holds(active(li2),+,A).
caused(active(li2),+,A) :-
    holds(closed(s2),+,A),
    holds(closed(s2p),+,A).

caused(active(li3),-,A) :-
    holds(closed(s3),-,A).
caused(active(li3),-,do(toggle(s2p),A)) :-
    holds(active(li3),+,A).
caused(active(li3),+,A) :-
    holds(closed(s3p),+,A),
    holds(closed(s3),+,A).

caused(active(re1),-,do(toggle(s1),A)) :-
    holds(closed(s1),+,A).
caused(active(re1),+,do(toggle(s1),A)) :-
    holds(closed(s1),-,A).

caused(active(re2),-,A) :-
    holds(closed(s2),-,A).
caused(active(re2),+,A) :-
    holds(closed(s2),+,A).

```

According to the circuit, every switch is a direct effect of the corresponding *toggle* action, whereas the bulbs and the relays depend on the state of the switches⁵. However, some causal rules are missing. In some cases, the corresponding effect axioms achieved the same compression and *LRAC* is biased to prefer direct effects. For instance, re_1 is an indirect effect of a single action $toggle(s_1)$, i.e., it depends on s_1 , and however *LRAC* preferred direct effects. With respect to re_2 , *LRAC* preferred a causal law because it is an indirect effect of $toggle(s_1)$ and $toggle(s_2)$ and it produces a shorter theory.

In other cases, the existence of cycles avoided *LRAC* to find the intended theory. For instance, the relays re_1 and re_2 in case of activation attract the switches located above (s_2 and s_3 respectively). However, with extensional evaluation, the following cycles were detected: $\{s_3, li\}$ and $\{s_2, re_2\}$, both for the positive and negative values. In these cases, the intended relation is in the form $re_2 \leftarrow s_2$ (resp. $li \leftarrow s_3$), however there are no examples in the form $\{s_2, \neg re_2\}$ or $\{\neg s_2, re_2\}$. The cycle $\{s_1, re_1\}$ does not appear because there was no gain in using indirect effects.

The cycles were broken by including some direct effects for one of the fluents, however, the corresponding effect axioms cover the same number of examples, thus *LRAC* chose one of them randomly. Actually, both re_2 and s_2 (resp. s_3 and li) need two effect axioms. As a result,

⁵The three resistors, are needed to keep low the current flow through the respective sub-circuit.

the causal rules for one of the fluents became redundant. However, even other causal laws were possible *LRAC* does not complete the theory with other indirect effects but uses only effect axioms for completeness. For instance, correct relations like $s_2 \leftarrow re_1$ are not learned, and however, it also breaks the cycle $\{s_2, re_2\}$. Similarly, the relation $s_3 \leftarrow re_2$ breaks the cycle $\{s_3, li\}$, but *LRAC* could not find it. Thus, other causal rules should be considered to break some cycles. This is an interesting issue to be studied in the future, however new cycles might appear and care must be taken not to consider the same theory twice.

The effect on the size of the theory learned is not evident for s_2 , because it needs also an effect axiom for $toggle(s_2)$ apart from the causal rule, however, it is more evident for s_3 , because it is an indirect effect of both $toggle(s_1)$ and $toggle(s_2)$. Note that intensional evaluation does not guarantee either that the intended theory will be found. For instance, when $toggle(s_1)$ is executed, s_1 and re_1 change at the same time, hence the effect axiom might be assigned to any of them indistinctly.

In this example, the difference in the size of the theory learned when only direct effects are allowed is notable. The impact is bigger for s_3 and li_3 that are in the highest level in the dependency graph (Fig. 6.12).

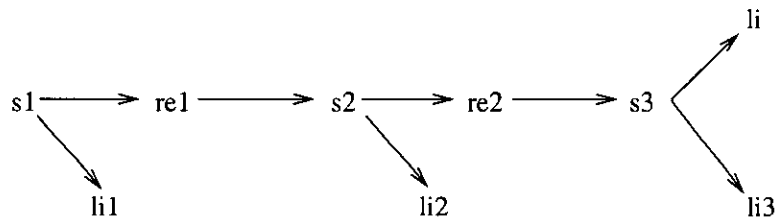


Figure 6.12: Propagation of effects in example 6

6.6.2 Recursive indirect effects

The use of indirect effects includes the possibility of *self-recursive* definitions for objects other than situations. Let us consider the following domain.

Example 7 (A modified blocks scenario) *Let us consider an scenario of the Blocks world that models the action of a robot that can move blocks onto other blocks or onto the ground as well as push blocks to different rooms.* □

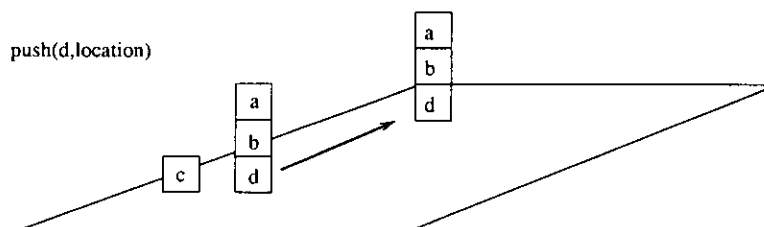


Figure 6.13: A modified Blocks world

The scenario includes the following definitions:

```

block(a;b;c;d;...).
room(r1;r2;r3).
location(ground).
location(A) :- block(A).
fluent(clear(B)) :- block(B).
fluent(on(A,B)) :- block(A),location(B).
fluent(at(A,B)) :- block(A),room(B).
action(move(A,B)) :- block(A),location(B).
action(push(A,B)) :- block(A),room(B).

```

From 25 narratives of length 4 including 96 examples for the fluent *at/2*, *LRAC* returned:

```

holds(at(A,B),+,do(push(A,B),C)) :-
    holds(at(A,B),-,C),
    holds(on(A,D),+,C),
    ground(D).
holds(at(B,C),+,A) :-
    holds(on(B,D),+,A),
    holds(at(D,C),+,A).
holds(at(B,C),-,A) :-
    holds(at(B,D),+,A),
    diff(D,C).

```

According to the above clauses, only blocks on the ground can be pushed. The second clause correctly asserts that pushing a block, changes the room of the block immediately over it and the blocks over this one recursively. The current room of a block changes (indirectly) when it is moved onto another block that is in a different room, hence, the causal rule subsumes also the corresponding effect axiom for *move/2*. Note that the clause does not need the condition *diff(C,ground)* because *ground* is not at a particular location, hence when a block is moved onto the ground, its location does not change. The last clause represents that a block can be only at one room at a situation, thus it subsumes the corresponding effect axioms for *move/2* and *push/2*.

With respect to action *push/2* no complete set of effect axioms could be found for both the positive and negative values, except for the blocks that are directly pushed, i.e., those that are over the ground. Otherwise, we should have as many clauses as the number of blocks piled in the highest tower. In this domain, handling indirect effects is strictly needed for achieving generalization, given that the effects of a single action are propagated to a pile of blocks, so that the number of affected blocks, i.e., the number of effects, varies on each situation. In this case, the recursion replaces a relation *above/2* to represent the transitive closure of *on/2*.

A similar case happens in the popular game *Minesweeper*.

Example 8 (Minesweeper) *The object of Minesweeper is to find mines which have been hidden at random by the computer on a grid.*

□

The game includes, among others, the following definitions:

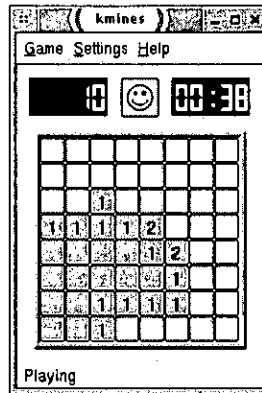


Figure 6.14: Minesweeper

```

row(1..maxrow).
column(1..maxcol).
content(0..8;bomb).

fluent(clear(A,B)) :- row(A),column(B).
fluent(marked(A,B)) :- row(A),column(B).
action(reveal(A,B)) :- row(A),column(B).
    action(mark(A,B)) :- row(A),column(B).
action(unmark(A,B)) :- row(A),column(B).

bombs(0..maxbombs).
in(A,B,C) :- row(A),column(B),content(C).
adjacent(A,B,C,B) :- row(A),column(B),row(C),
    C<=A+1,C>=A-1,C!=A,
adjacent(A,B,A,D) :- row(A),column(B),column(D),
    D<=B+1,D>=B-1,B!=D.
adjacent(A,B,C,D) :- row(A),column(B),row(C),column(D),
    C<=A+1,C>=A-1,C!=A,
    D<=B+1,D>=B-1,B!=D.

```

The game starts when the player moves his mouse and selects a square on the grid. With a click of the left-hand key of the mouse, the computer opens the square on that grid and reveals whether the square contains a mine or not. If that square does contain a mine, then that player has lost the game. If the square does not contain a mine, then the square will contain a number showing how many mines there are on the adjoining 8 squares. However, if the square is blank, that means that there are no mines on any of the adjoining 8 squares. In that event, the computer automatically helps out by opening all 8 adjoining squares and revealing their numbers. If it happens that any of those squares are also blank, then the squares surrounding those squares are also opened. This last case requires a recursive execution of the theory. For this case, *LRAC* learned the following clauses:

```

caused(clear(A,B),+,do(reveal(A,B),C)) :-
    holds(clear(A,B),-,C).

```

```

caused(clear(A,B),+,C) :-
    adjacent(A,B,D,E),

```

```
holds(clear(D,E),+,C),
in(D,E,0).
```

The effect axiom just clears the square selected by the player, whereas the causal rule propagates the effect of the action to the surrounding squares.

6.6.3 Negative cycles

So-called negative cycles are those where a fluent depends on its own negation. Negative cycles do not correspond (at least intuitively) to real situations. The presence of a negative cycle may manifest an error in the design of a system. In practice, they are used to achieve non-determinism based on the answer sets semantics or undefinedness in the Well-founded semantics [63]. Let us consider the following circuit given in [123].

Example 9 (Shanahan's circuit) Consider the modification of Thielscher's circuit depicted in Fig. 6.15. This circuit incorporates a potentially vicious cycle of fluent dependencies. \square

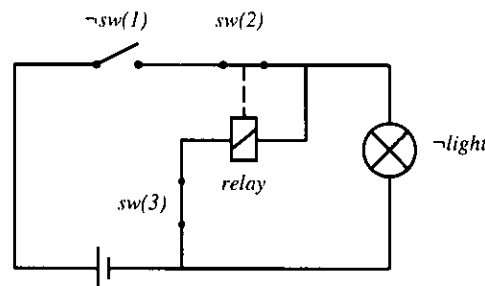


Figure 6.15: A modification of Thielscher's circuit

The circuit of Fig. 6.15 has only two states. When sw_3 is open, the circuit behaves like the basic circuit of example 2 where sw_1 and sw_2 control the state of the light. When sw_3 and sw_2 are closed, if sw_1 is closed, the *relay* is activated, opening sw_2 , however this prevents the *relay* from being activated. Thus, when the circuit stabilizes, the *relay* is not active. In this case, the *relay* is made active and inactive consecutively and sw_2 might be opened due to the momentary activation of the relay (state t_{2b} in table 6.2). As a consequence, for an external observer, the action of closing sw_1 does not cause the relay to become active. In the real system, it might happen that the *bulb* turns on for a very short period of time (t_{2a}), depending on the time it takes to activate the relay and to affect the second switch. Nonetheless, the *bulb* and the *relay* are definitely off in the resulting state. It would be even possible that the momentary value of the bulb has effects that remain visible, e.g., a light detector, although the *bulb* is off in the resulting situation.

A negative cycle produces the existence of *intermediate situations* in the computation of the resulting situation, where the constraints, e.g., that *light* is on when sw_1 and sw_2 are closed, are violated for very small periods of time until a stable state is reached.

The left column of table 6.2 assumes that the *bulb* has a smaller delay than the *relay*, then the latter opens sw_2 and consequently the *light* is off. The right column assumes that the *relay* has a smaller delay than the *bulb* and then two cases are possible: in the first case, the *light* is on until the relay opens sw_2 , whereas in the second case, the *light* is never on, i.e., the effect of the relay on sw_2 is faster than the effect of sw_1 on *light*.

t_{1a}	$\{sw_1, sw_2, sw_3, \overline{relay}, \overline{light}\}$	t_{1b}	$\{sw_1, sw_2, sw_3, \overline{relay}, \overline{light}\}$	t_{1c}	$\{sw_1, sw_2, sw_3, \overline{relay}, \overline{light}\}$
t_{2a}	$\{sw_1, sw_2, sw_3, \overline{relay}, \overline{light}\}$	t_{2b}	$\{sw_1, sw_2, sw_3, relay, \overline{light}\}$	t_{2c}	$\{sw_1, sw_2, sw_3, relay, \overline{light}\}$
t_{3a}	$\{sw_1, sw_2, sw_3, relay, \overline{light}\}$	t_{3b}	$\{sw_1, sw_2, sw_3, relay, \overline{light}\}$	t_{3c}	$\{sw_1, \overline{sw_2}, sw_3, relay, \overline{light}\}$
t_{4a}	$\{sw_1, \overline{sw_2}, sw_3, relay, \overline{light}\}$	t_{4b}	$\{sw_1, \overline{sw_2}, sw_3, relay, \overline{light}\}$	t_{4c}	$\{sw_1, \overline{sw_2}, sw_3, relay, \overline{light}\}$
t_{5a}	$\{sw_1, \overline{sw_2}, sw_3, relay, \overline{light}\}$	t_{5b}	$\{sw_1, \overline{sw_2}, sw_3, \overline{relay}, \overline{light}\}$		
t_{6a}	$\{sw_1, \overline{sw_2}, sw_3, \overline{relay}, \overline{light}\}$	t_{6b}	$\{sw_1, \overline{sw_2}, sw_3, \overline{relay}, \overline{light}\}$		

Table 6.2: Intermediate situations in the circuit 9 after closing sw_1

Let us consider another example given in [26].

Example 10 (Double relay example) *An electric circuit consisting of two interconnected sub-circuits. On one circuit there are two serially connected switches p and s and a relay $relay_1$, on the other, two switches r and q and a relay $relay_2$.* \square

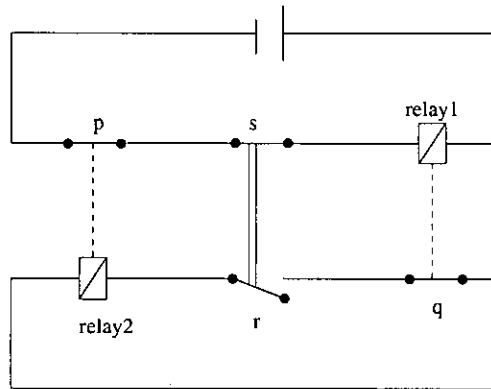


Figure 6.16: Double relay

All narratives used for learning start from an initial situation where switches p , q and s are closed, r is open, $relay_1$ is active and $relay_2$ is inactive. Under these conditions, *LRAC* returned:

```

caused(closed(swr), -, do(toggle(swr), A)) :-
    holds(closed(swr), -, A).
caused(closed(swr), +, do(toggle(swr), A)) :-
    holds(closed(swr), +, A).
caused(closed(sws), -, do(toggle(sws), A)) :-
    holds(closed(sws), +, A).
caused(closed(sws), +, do(toggle(sws), A)) :-
    holds(closed(sws), -, A).
caused(active(relay1), +, A) :-
    holds(closed(sws), +, A).
caused(active(relay1), -, A) :-
    holds(closed(sws), -, A).
caused(closed(swq), -, A) :-
    holds(active(relay1), -, A).
caused(closed(swq), +, A) :-
    holds(active(relay1), +, A).

```


No rules were learned for p and $relay_2$ because no action modified their values. Furthermore, only actions over r and s produced any change on the circuit's state. Thus, despite the complexity of the circuit, it exhibits a quite simple behavior, where switches r and s are mechanically connected such that always exactly one of them is open. The cycle of s and r can be solved in several ways.

According to the learned rules, the circuit above has only two states. Closing r leads to the opening of s , hence $relay_1$ is deactivated and q opens. Furthermore, $relay_2$ remains inactive and p remains closed. If r gets opened, s will be closed, so $relay_1$ will receive current and opens q . In neither case p is open. As a result, $relay_2$ can never become active. The first relay ensures that q is closed if and only if there is current in the first circuit, whereas $relay_2$ makes sure that p is open if there is current in the second circuit.

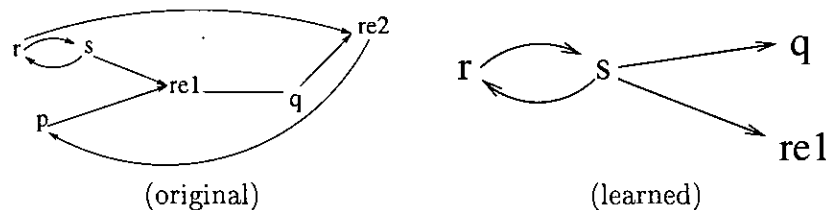


Figure 6.17: Dependency graphs for the circuit 10

We actually do not learn the cyclic dependences (left-hand side of Fig. 6.17), but just a very simplified behavior (right-hand side of Fig. 6.17). Note that trying to toggle p or q , makes the circuit enter an infinite cycle of oscillations. For instance, after $toggle(p)$, p closes, $relay_1$ becomes inactive which opens q , then $relay_2$ remains inactive thus forcing p to be open, which activates $relay_1$, and so on, and the circuit never reaches a stable state (until something breaks, most likely), which seems to reflect a bad design of the circuit. This is the reason that no such actions are included in the observations. Note that without the assumption that switches r and s are connected, after closing p if both s and r are closed, the circuit also enters an infinite cycle of oscillations

Most formalisms of action domains, produce an *undefined* value for the fluents involved in the oscillation [26] and in general for negative cycles. In both circuits, the negative cycle causes an odd behavior which seems to correspond to a bad design of the circuit rather than to a particular purpose. We have not considered learning negative cycles as part of a domain description, however it would be possible by analyzing the sequence of intermediate situations –effect propagations– and using a special procedure.

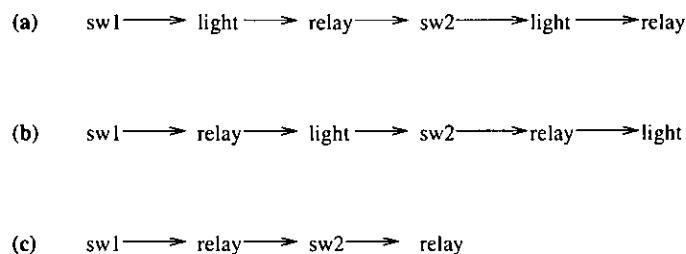


Figure 6.18: Effect propagation in circuit 9

6.7 Conclusions

Previous approaches to learning action models are restricted to predicting a single outcome or effect of an action. This forces the explicit representation of all the effects of an action as direct effects, producing the so-called *Ramification problem*. This makes the descriptions of actions cumbersome and difficult for complex domains. In this chapter we have shown how to learn action theories with direct and indirect effects. For this task, we have used a dialect of the Situation Calculus based on causality, where causal laws are used for the propagation of effects. Indirect effects also provide a benefit as to learnability, because the final theory is shorter and then easier to learn. We have also dealt with special cases of indirect effects, such as mutual recursion or negative cycles.

Chapter 7

Learning Default Action Theories

The previous chapters dealt with effects of actions on a world where these are strictly specified. In this chapter we show how to incorporate defeasibility into the specifications and introduce *defeasible* constraints and effect propositions. The explicit use of exceptions allows to learn rules that are more generally applicable, mainly when there are occasional qualifications or noise in the observations which may decrease the quality of the learning results, or when a theory must be specialized minimally [5].

7.1 The Qualification Problem

The *Qualification Problem* in Reasoning about actions, first identified by McCarthy in 1977 [79], concerns how to express the preconditions for actions without having to account for the many conditions which, albeit being unlikely to occur, may prevent the successful execution of an action [127]. McCarthy uses the following example: that it is necessary to have a ticket to fly on a commercial airplane is rather common to express. That it is necessary to be wearing clothes needs to be kept inexplicit unless it somehow comes up. A proposition like that should not be treated as a strict precondition in the formal specification of the action, so that a reasoning agent always has to verify this condition before assuming that the action can be successfully executed. In this case, exceptions represent qualifications to the actions that are not intimately related to the action. Moreover, it is often difficult if not impossible to even think of all conceivable disqualifications in advance.

Any solution to the Qualification problem must incorporate *defeasible* specifications. It is important to consider defeasible specifications for the same reason as the necessity of non-monotonic theories in knowledge representation and commonsense reasoning. The solution to the Qualification problem is based on the fact that most conditions are so likely to be satisfied that they are assumed away in case there is no evidence to the contrary, so that all of the qualifications for an action are grouped under a *disabled* or *abnormal* predicate, that is assumed false by default. Assuming away so-called *abnormal disqualifications* by default implies that if further knowledge is available about one of such disqualifications, the previous assumption that the action is executable must be withdrawn. These cases will be considered as *exceptions* to the general rules. Since any exception is considered unlikely, we do however wish to ignore it unless there is evidence to the contrary.

The ability to assume away, by default, exceptional disqualifications requires some non-monotonic features. In Logic Programming, negation-as-failure is used to represent absence of

information about exceptions. For instance, the clause to determine if a year is a leap year can be expressed as follows:

```
normal(Y):- not ab0(Y).
ab0(Y):- divisible(4,Y), not ab1(Y).
ab1(Y):- divisible(100,Y), not ab2(Y).
ab2(Y):- divisible(400,Y).
```

where a hierarchy of “abnormalities” is used to represent exceptions to the general rule.

The use of defeasible specifications adds elaboration tolerance to a domain description. For instance, the addition of new actions to a domain description may make some previous assertions no longer valid. As pointed by V. Lifschitz, an elaboration tolerant formalism should allow us to retract invalid assertions, in the spirit of non-monotonic reasoning, by adding new postulates. In particular, if a new condition is observed where an action is disqualified, that is not considered in the effect axiom, we do not have to modify the effect axiom but just to add new *abnormal* clauses.

In the next sections, we will analyze the impact of the Qualification problem in learning, indeed, we will see that for learning, the need for defeasible propositions has more motivations than just coping with exceptional disqualifications.

7.2 Default theories

In other than artificial environments, complete knowledge of all the relevant facts cannot be assumed. With regard to example 2, when we toggle a switch then, contrary to our expectations, the light may actually not turn on –due to, for instance, a broken bulb, a malfunction of the battery, or loose wiring etc. According to the consistency condition (df. 3.1), the presence of just a single “abnormal” example avoids the *general rule* that apply to normal cases to be learned. In an ideal case, the learned rules should contain the general conditions and explicitly enumerate all the possible exceptions. For instance, in the circuit of example 2:

```
caused(active(light),+,do(toggle(sw1),A)) :-
    holds(closed(sw1),-,A),
    holds(closed(sw2),+,A),
    not holds(light_broken,+,A),
    not holds(battery_malfunction,+,A),
    ...
```

The presence of large sets of preconditions for an action can lead to low accuracy results unless a high number of good counterexamples is available. The bias of the learning algorithm is a determining factor as even if the background includes all possible influencing factors, the number and quality of the negative examples can make an induction algorithm to find a different clause rather than the general one with all the possible exceptions, usually including other less relevant conditions. Furthermore, the successful execution of actions depends on many more conditions than we are usually aware of. In an extreme case, data includes observations where the prediction of the same action under the same (known) conditions may succeed at one time but fail at another, adding non-determinism to data. In these cases, it is useful to relax the consistency requirement and learn more general clauses that might cover a small amount of counterexamples. In dynamic domains, this is particularly interesting as it allows the possibility of discovering *default rules* that describe the most common situations, without having to account for the many conditions

which, albeit being unlikely to occur, may prevent the successful execution of an action, and that can avoid the general rule to be found.

This argumentation is similar to the motivation for the Qualification problem in Commonsense reasoning, however, allowing exceptions to the rules is interesting in other situations. For instance, another source of exceptions, apart from abnormal qualifications, is *noise*. The effect that noise produces is the impossibility to learn a definition, because no clause is contained in the language bias that is consistent, or an overspecific definition is learned, composed of very many specific clauses instead of a few general ones. In these cases it is also advantageous to learn clauses that are more generally applicable.

On the other hand, exceptions can be used as a method of specialization on its own, where negative examples are used as exceptions to general rules, or as exceptions to exceptions and so on. This contrasts with the normal method of specialization where a full set of preconditions is learned for each rule. For instance, a clause to predict animals that fly can be expressed with or without exceptions (Fig. 7.1).

flies :- bird, not ab1. ab1 :- penguin, not ab2. ab2 :- superpenguin.	flies :- bird, not penguin. flies :- superpenguin.
(a)	(b)

Figure 7.1: Birds and penguins

The theory (a) makes explicit *bird* as a general (necessary) condition, where *penguins* are abnormal birds and *superpenguins* are abnormal penguins. Thus, the theory represents a sequence of tests to be passed, from general to specific, so that when presented with a new animal, the first test to be applied is *bird* and not a more specific condition like *superpenguin*. Unlike this, in theory (b), *bird* and *superpenguin* are at the same level. This particular form of representation is appropriate when conditions are not independent, for instance, *penguin* is an specialized *bird* and *superpenguin* is an specialized penguin.

7.3 Default Action Theories

In order to account for exceptions in action theories we introduce for each fluent and each action a unique 'abnormality' predicate ab_i [10]. In this case, each effect axiom is enhanced by a normality condition, which restricts the axiom to all but abnormal circumstances (using NAF) and some facts for ab_i are added to the theory.

$$Caused(f, \text{true}, do(a, s)) \leftarrow \pi^+, \text{not } Ab(f, \text{true}, a, s) \quad (7.1)$$

$$Caused(f, \text{false}, do(a, s)) \leftarrow \pi^-, \text{not } Ab(f, \text{false}, a, s) \quad (7.2)$$

where the *Holds* atoms in π^+ and π^- are only of the form $[-]Holds(f', s)$. The *abnormal/4* atom includes the fluent and the action because a fluent can be abnormal only when a particular action is executed. Similarly, there is a corresponding *abnormal/4* atom for the positive and the negative value to distinguish when a fluent is abnormal in the positive value or in the negative one, otherwise *abnormal/4* would block rules for both values, for instance, when the conditions for the rule of $Ab(f, \text{true}, a, s)$ and $Caused(f, \text{false}, do(a, s))$ hold simultaneously. In the final theory,

the lowest layer corresponds to inertia, the next layer corresponds to the defeasible propositions and the highest layer is the effect and causality propositions.

We have so far considered *defeasible* action laws, however, the use of causal rules can be also done defeasible when abnormalities exist [10]. Fluents are considered abnormal after executing an action, hence *abnormal/4* literals include always the action, however, causal rules range over every possible action, thus we have:

$$\text{Caused}(f, \text{true}, s) \leftarrow \pi^+ \dots, \text{not } \text{Ab}(f, \text{true}, A, s) \quad (7.3)$$

$$\text{Caused}(f, \text{false}, s) \leftarrow \pi^- \dots, \text{not } \text{Ab}(f, \text{false}, A, s) \quad (7.4)$$

where $do(A, s)$ will refer to the resulting situation after performing any action A , where A is a variable.

Our approach to learn action theories with exceptions, coincides with those used for learning ELPs with exceptions [54, 63]. Induction first generates effect axioms from positive and negative examples and background knowledge in an ordinary ILP framework and returns a definition consisting of default rules, together with definitions for the abnormality literals. The extension of *abnormal/4* is generated from negative examples covered by the rules and output as a set of ground atoms

By doing so, the definitions learned for the positive and negative concepts may overlap (Fig. 7.2). The rectangle represents all situations where an action a is executed whereas the circle represents the observed part (the training set). The area labeled F (resp. \bar{F}) represents the situations where the fluent F is caused to hold (resp. not to hold).

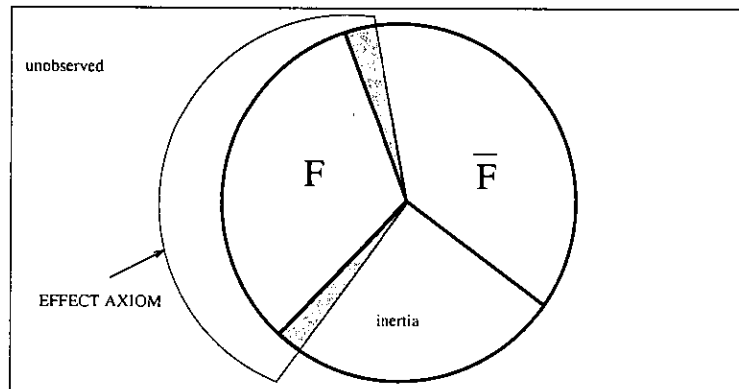


Figure 7.2: Overlap of positive and negative definitions

For the observed cases, i.e., for the overlapping in the training set, the inconsistency is removed by explicitly considering exceptions to the rules. A second case is possible in action theories where learned rules overlap with the inertia axiom in the observed part. This happens when a learned rule infers “caused” for an inertia value. In this case, the overlap can be produced with respect to the truth and “causality” value or just with respect to the “causality” value (i.e., an inertia value is said to be caused). As inertia values are used as negative examples, they are dealt with as any other negative example and converted to exceptions to the rules for F and \bar{F} .

In the normal ILP semantics, it is required that the learned program is consistent only with respect to the examples but not necessarily for unseen atoms. For non-observed literals, the two overlapped classifications are equally strong. Following [54, 63], the conflict can be resolved by classifying them as undefined as we saw in section 4.3.5.

observed	predicted	abnormalities
caused(F)	caused($\neg F$)	ab($\neg F$)
caused($\neg F$)	caused(F)	ab(F)
holds(F)	caused($\neg F F$)	ab($\neg F F$)
holds($\neg F$)	caused($\neg F F$)	ab($\neg F F$)

Table 7.1: Possible abnormalities

7.4 Learning rules with exceptions

The problem of learning non-monotonic logic programs has been already addressed in the ILP literature by several authors [63, 29, 54, 5, 31]. In [5], the typical top-down specialization procedure is substituted by the introduction of an *abnormal* predicate and negation-as-failure. In [54], a system LELP is developed to learn non-monotonic programs with exceptions in the form of Extended Logic Programs. LELP learns default rules for the positive value or the negative, or for both (parallel default rules), according to the ratio of positive and negative examples, whereas in [63], the system LIVE, which learns extended logic programs under the Well-founded semantics with negation (WFSX), learns rules for both the target concept and the negation. Kakas et al [29] use defaults with priorities where a priority order is used in the proof procedure. Finally, in [31], Default Logic is used.

To allow for the induction of default rules, different techniques exist. In [54, 63] the Golem [94] algorithm is used to generate the RLGG, however, during the phase of rule generation to cover positive examples no negative examples are used to specialize rules. Since the clause is not tested on negative examples, it may cover some of them. The specialization of overgeneral rules is performed only by creating exceptions identified as objects contained in negative examples. Thus, abnormality literals substitute the normal specialization methods based on adding literals to a clause until it does not cover negative examples. In methods that learn the most general generalization, like Progol, we can relax the consistency criterion and set an *upper bound* on the number of negative examples that can be covered by any acceptable clause, such that, specialization needs not be fully completed.

By relaxing the consistency criterion we need to explicitly give the number of exceptions allowed, which should coincide with the number of real exceptions. The aim of systems adopting RLGG is using abnormality literals as the only means of specializing a clause. For instance, in [5], the aim is to perform a minimal specialization in incremental learning systems. Unlike this, by relaxing the consistency criterion, specialization uses partially the negative examples. In any case, a decision must be taken about when default rules should be generated, that is, when to convert some examples into exceptions to a more general rule or complete the specialization process. In the approaches that use Golem, since Golem is computing the least general generalization, the concept will not cover any more negative instances than necessary. When the number of exceptions is explicitly provided, it would be possible to learn the most general rule caused($f, +, do(a, s)$) that covers all positive and all negative examples, such that all negative examples correspond to exceptions. A simple heuristic is that the set of exceptions must be smaller than non-exceptions [54]. Usually, a percentage of exceptions over the number of positive examples covered, is provided instead of the exact number of exceptions. Another useful criterion is compression, such that, when a clause must be specialized too much in order to make it consistent, we should prefer to transform it into a default rule and consider the covered

negative examples as exceptions. For instance, Progol's compression measure uses the length of the hypotheses as a negative factor.

7.4.1 A small example

Let us consider again the circuit of example 6. According to the circuit, the relays, in case of activation, attract the switches located above, thus changing the state of lamps.

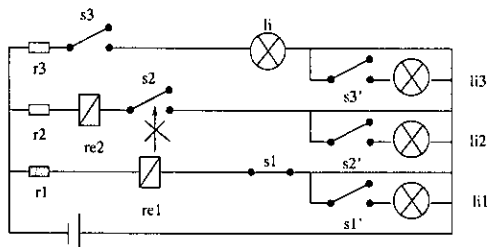


Figure 7.3: Another electric circuit

In extreme cases, exceptions can make that the prediction of the same action under the same (known) conditions may succeed at one time but fail at another, adding no-determinism to data. We included some examples where resistor r_1 failed randomly, hence, when s_1 is closed and r_1 is working abnormally, relay re_1 no longer works correctly (Fig. 7.3). These abnormal examples resulted in the impossibility to learn a definition for s_2 (among others). We repeated the learning process relaxing the consistency criterion and found:

```

caused(closed(s2),+,A) :-
    holds(active(re1),+,A),
    not ab(closed(s2),+,B,A).
caused(closed(s2),-,do(toggle(s2),A)) :-
    holds(closed(s2),+,A),
    holds(active(re1),-,A).

caused(closed(s2),-,do(toggle(s2),...s0)).
...
ab(closed(s2),+,A,do(...,s0)...).
...

```

For the positive value of s_2 , the causal rule (s_2 is closed when re_1 is active) fails when the re_1 is active but working abnormally. In this case, *LRAC* learned the causal rule but with exceptions. A different case happens with the negative value of s_2 , given that no exceptions are produced because the occasional cases are those where the rule does not need the precondition of re_1 being inactive. In this case, these abnormalities do not qualify the action but make the action successful in some additional situations. That is, the learned clause does not cover those examples where re_1 is active and not working correctly, thus s_2 can be abnormally opened. Thus, the abnormal cases correspond to *uncovered* examples or *uncompressed* in the Progol terminology, because, the uncovered positive examples are less than the negative examples covered if we remove re_1 from the body. Both exceptions and uncovered examples are returned as ground facts.

The abnormal behavior of re_1 is not propagated to other components. For instance, li_2 is not affected because it depends on s_2 and s_{2p} , and the state of li_2 is always consistent with respect to them. However, if we learn li_2 as a direct effect of any action that eventually modifies it, for

instance, $toggle(s_1)$, then li_2 would be also affected by the abnormal behavior of re_1 and some exceptions would be produced.

In the previous example, relaxing the consistency criterion is strictly needed for learning. In less extreme cases, for instance when the exceptions represent qualifications to the actions that appear occasionally, or when there is noise or when important data are missing, generalization is still possible without considering exceptions, however, poor generalizations might be obtained. As a consequence, the use of a predicate $abnormal/4$ that represents exceptions explicitly, contributes to learnability, as it favors that more general rules are learned.

7.5 Learning about exceptions

If the exceptions form a class, i.e., if exceptions have some common properties, the simple enumeration is not informative and rules about exceptions can be obtained that define the circumstances under which a particular abnormality occurs. This will be possible provided that we have data on their common properties and the language bias so allows. This is accomplished by repeating the learning process for $abnormal/4$ literals and learning additional constraints each of which relates some $abnormal/4$ to the conceivable causes [54, 63]. Rules about exceptions have such abnormal predicates in their head and are results of generalizations of some abnormal atoms:

$$Ab(f, \text{true}, a, s) \leftarrow \pi^+ \quad (7.5)$$

$$Ab(f, \text{false}, a, s) \leftarrow \pi^- \quad (7.6)$$

where the $holds/2$ literals in π^+ and π^- are only of the form $[neg]Holds(f', s)$. Positive (resp. negative) examples for $abnormal/4$ are obtained from the set of negative (resp. positive) examples covered by the overgeneral learned rules. Thus, an overgeneral rule for $abnormal/4$ specializes the rule for $caused/3$, i.e., increases the number of exceptions. Rules about exceptions should be used to derive only exceptions. In fact, exceptions are usually minimized in non-monotonic reasoning. When such a common rule cannot be generated or there are some exceptions that cannot be covered by such a rule, those exceptions are left as they are and returned without generalization.

However, the aim of using $abnormal/4$ as the only method of specialization is to learn a theory composed of a default rule that is generally applicable, and to represent exceptions to this rule, exceptions to these exceptions and so on. By doing so, exceptions to the definitions of $abnormal/4$ might be found and so on, thus leading to a hierarchy of exceptions [63, 54, 29] (denoted as Ab_i).

$$Ab(f, \text{true}, a, s) \leftarrow Holds(f', s), \dots, \text{not } Ab_1(f, \text{true}, a, s) \quad (7.7)$$

$$Ab_1(f, \text{true}, a, s) \leftarrow Holds(f'', s), \dots, \text{not } Ab_2(f, \text{true}, a, s) \quad (7.8)$$

...

These predicates Ab_i actually represent exceptions to exceptions, such that, taking $Ab \equiv Ab_0$, we have that for i even, the generalization of Ab_i makes the rule for Ab more general and hence the effect axiom more specific, whereas for i odd, the generalization of Ab_i makes the rule for Ab more specific and hence the effect axiom more general.

The need for explicitly providing a maximum number of exceptions is a naive form of learning rules with exceptions and it is not appropriate at all to learn about exceptions. A more natural option is to compute the RLG as in the systems LIVE and LELP. For this task, we implemented

E_{ab}^+	is generated from	E^- covered by R
E_{ab}^-	is generated from	E^+ covered by R
$E_{ab_1}^+$	is generated from	E^+ covered by $R \cup R_{ab}$
$E_{ab_1}^-$	is generated from	E^- covered by $R \cup R_{ab}$
	...	

Table 7.2: Generation of examples for *abnormal/4* predicates

a very basic version of the Golem algorithm that accepts ground observations in the Situation Calculus, such that, specialization is not fully completed, i.e., no negative examples are used to specialize rules. Clauses are obtained by randomly taking couples of examples, computing their *rlgg* and choosing the consistent one that covers the biggest number of positive examples. This clause is further generalized with new positive examples until the clause covers some negative examples [54]. The specialization of overgeneral rules is performed only by creating exceptions identified as objects contained in negative examples. Repeating the learning process for the *abnormal/4* atoms, we eventually obtain a theory that is consistent.

Let us consider a variation of the Blocks world, where a robot (with one or two grippers) is used to move the blocks, such that some blocks are too heavy to be moved by robots having a single gripper. In this case, we obtained an overgeneral version of the intended effect axiom.

```
caused(on(A,B),+,do(move(A,B),C)) :-
  holds(clear(A),+,C),
  holds(clear(B),+,C),
  not ab(on(A,B),+,move(A,B),C).
```

The algorithm takes a pair of examples from which it computes the RLGG, so that the two examples contribute to the generation of one clause. Even if they do not form part of the same clause in the target concept, e.g., the block is heavy only in one of them, the RLGG includes a common part that is applicable to both, i.e., the default rule. By repeating the learning process for *abnormal/4* literals until a rule without exceptions is found, we obtained a consistent theory in the form:

```
caused(on(A,B),+,do(move(A,B),C)) :-
  holds(clear(A),+,C),
  holds(clear(B),+,C),
  not ab(on(A,B),+,move(A,B),C).
ab(on(A,B),+,move(A,B),C) :-
  heavy(A),
  not ab1(on(A,B),+,move(A,B),C).
ab1(on(A,B),+,move(A,B),C) :-
  grippers(2).
```

The effect axiom represents the most general conditions necessary for moving blocks, whereas the rule for *ab* represents exceptions to the general rule and the rule for *ab₁* exceptions to exceptions. With the normal specialization, we obtained the following theory:

```
caused(on(A,B),+,do(move(A,B),C)) :-
  holds(clear(A),+,C),
  holds(clear(B),+,C),
```

```
not heavy(A).
caused(on(A,B),+,do(move(A,B),C)) :-
  holds(clear(A),+,C),
  holds(clear(B),+,C),
  grippers(2).
```

In this case, common conditions are replicated in both rules, and the need for two grippers is not associated to the weight of the blocks.

Without additional considerations, such an algorithm would continue specializing and adding abnormality predicates until a consistent theory is obtained for some Ab_i . Thus, if observations contain noise, non-real exceptions would be created to make the theory consistent. This will necessarily result in fitting the noise [125]. Similarly, if there is non-determinism in the training set due to the absence of relevant information, relaxing the consistency criterion is essential for learning as the only means to achieve generalization, however, learning about these abnormalities makes no sense and it must be considered as another source of noise in the observations. Any attempt to generalize from the set of exceptions will produce very low compression rates, hence exceptions should be returned as ground facts.

In this case, it is important to have a reliable method of deciding whether to treat errors as noise or to include them as exceptions. One way to approach the problem is to see if the exceptions to the current theory *exhibit a pattern* by adopting one of the heuristic necessity stopping criteria proposed in ILP to handle noise, such as the *encoding length restriction*, a significance test and so on. Methods based on an information theoretic measure are based on the following idea [125]: “data found to be incompressible are deemed to be noise”. Each specialization performed by the algorithm is an attempt to improve the accuracy of the theory, at the expense of increasing its size. If the specialization was worthwhile, the gain in accuracy should outweigh the cost incurred in increasing the theory size. By doing so, when a rule for ab_i achieves a good compression it is very unlikely that this is due to noise.

7.6 Conclusions

In this chapter we have shown how to incorporate defeasibility into the specifications and have introduced *defeasible* constraints and effect propositions. The explicit use of exceptions allows to learn rules that are more generally applicable, mainly when there are occasional qualifications or noise in the observations which may decrease the quality of the learning results, or when a theory must be specialized minimally [5]. However, we have seen that there are more reasons to learn rules with exceptions. Indeed, learning about exceptions is a particular form of specialization where the final theory consists of a set of default rules that are generally applicable and might cover some negative examples, and a set of rules for a predicate *abnormal/4* that represents exceptions to the general rules as well as exceptions to exceptions and so on, which in many cases correspond to the natural description of a domain. We have just considered how to learn rules with and about exceptions in action domains, by applying previous work for static domains, however, much work is still needed to make these methods applicable in more real data containing noise etc.

Part III

Extensions



Chapter 8

Complex actions

We have so far assumed that actions are atomic, durationless and have instantaneous effects. Furthermore, we have assumed that changes in the values of fluents can only be caused by the execution of actions, otherwise the state of the system is assumed to be stable.

If, for instance, *concurrent actions* are allowed, an effect may depend on a particular *combination* of actions, an action may qualify another action's effects, effects can be cancelled and so on. Most scenarios in the real world include the concurrent execution of basic actions, hence it is important for a learning agent to be able to represent and learn from them.

On the other hand, if there are action sources other than the agent itself, an environment may seem active to an observer. Exogenous changes are quite common, for instance, if the domain has properties that are non-amenable to manipulation (weather conditions), a second agent is acting in the domain, and so on. In this case, the learner must consider not only the environment but also the sources of other actions.

Finally, most actions (e.g., picking up a block, going from one location to another) take time. Similarly, effect propagations usually incorporate very small delays. For most practical purposes these delays can be abstracted away and the effects assumed to be simultaneous and instantaneous. This is convenient when the resulting model is simpler or necessary when there is no actual knowledge for providing an accurate model. However, when they affect the results we are interested in, the delays must be explicitly represented and incorporated into the learning process.

In this chapter we will consider some extensions to the framework presented in the previous chapters that allow to handle most of these issues.

8.1 Concurrent actions

We have so far restricted to so-called atomic actions (or non-concurrent actions), hence we have learned an effect axiom for each fluent and for each action that affects it. One of the criticisms most frequent to the Situation Calculus was that it could not handle concurrent actions. In narrative formalisms, an additional predicate *happens/2* is introduced to state that an action *a* is executed (happens) at situation *s* and so the action becomes part of the situation together with the values of the fluents. The form of the direct effects in a narrative formalism would be:

```
caused(closed(sw1),+,A) :- happens(toggle(sw1),A),holds(closed(sw1),-,A-1).
```

where A-1 refers to the previous situation. Concurrent actions are considered by adding multiple *happens/2* literals to the body.

Some extensions exist to the Situation Calculus that allow to handle concurrent actions [48, 70, 9, 121]. An initial solution to cope with concurrent actions in the Situation Calculus consists of creating a new sort for *compound actions* with respect to simple actions, consisting in sets of actions, so that the *do/2* function takes now compound actions as arguments. The term $\{a, a'\}$ denotes a compound action. Let us consider a classical example in the reasoning about actions literature.

Example 11 (Soup bowl) There are two actions in the domain $\{\text{LiftLeft}, \text{LiftRight}\}$ that represent the action of lifting the left (resp. right) side of a soup bowl. The fluent *OnTable* holds when the bowl is on the table, the fluent *HasWater* holds when the bowl has water, and the fluent *Spilled* holds when the water is on the table but not in the bowl. \square

Some possible observations are:

```
holds(spilled, -, do({liftleft, liftright}, A)).
holds(spilled, +, do({liftleft}, A)).
```

With a compound action for each possible combination of the actions, a combination is considered as a new atomic action. Unfortunately this avoids that generalization works over actions, thus having an effect axiom for every possible combination of actions, including not relevant combinations. Following [70], concurrent actions are introduced into the Situation Calculus through a new function $in(a, c)$ denoting that a compound action c includes a simple action a . We need a new sort *compound* to implement the function $in/2$, relying on the standard interpretation of sets and relations (membership, subset, etc) and their operations (union, intersection, etc).

A distinction is made between concurrent actions whose effects are *cumulative* and concurrent actions whose effects *cancel* each other out. In the first case, several actions must be executed concurrently to achieve a particular effect (allowing multiple $in/2$ literals in the body). In the case of *cancelling* actions, we need to state explicitly that an individual action is *not* part of a compound action. For this reason, negation as failure must be explicitly used for $in/2$ literals. Following the notation in [121], a second new predicate is introduced in the Situation Calculus to cope with *cancelling* effects, where $Cancel(c, a, s)$ represents that normal effects of action a are cancelled by the components of c if a and c are performed concurrently in situation s . Formally we have:

Definition 8.1 (Situation Calculus Program with concurrent actions) A Situation Calculus program is the conjunction of:

- A finite set of general clauses

$$[\neg] Holds(f, s_0) \tag{8.1}$$

where s_0 denotes the initial situation.

- A finite set of clauses of the form

$$Caused(f, v, do(c, s)) \leftarrow \pi \tag{8.2}$$

where c is a compound action, π does not mention the *Affects* or *Cancel* predicate, at least a literal $In(a, c)$ is included in π where a is an atomic action, and every occurrence of the *Holds* predicate in π is of the form $[\neg] Holds(f', s)$ or $not\ Cancel(c', s)$ where c' is a compound action.

- A finite set of *Cancels* clauses of the form

$$\text{Cancels}(c, c', s) \leftarrow \pi \quad (8.3)$$

where π does not mention the *Affects* predicate, c and c' are compound actions, at least a literal $\text{In}(a, c)$ is included in π where a is an atomic action and every occurrence of the *Holds* predicate in π is of the form $[\neg]\text{Holds}(f, s)$.

- The universal frame axiom.

$$\text{Holds}(f, \text{do}(c, s)) \leftarrow \text{Holds}(f, s) \wedge \text{not } \text{Caused}(f, v, \text{do}(c, s)) \quad (8.4)$$

$$\neg \text{Holds}(f, \text{do}(c, s)) \leftarrow \neg \text{Holds}(f, s) \wedge \text{not } \text{Caused}(f, v, \text{do}(c, s)) \quad (8.5)$$

where c is a compound action.

- A clause that propagates caused values to *Holds*.

$$\text{Holds}(f, \text{do}(c, s)) \leftarrow \text{Caused}(f, \text{true}, \text{do}(c, s)) \quad (8.6)$$

$$\neg \text{Holds}(f, \text{do}(c, s)) \leftarrow \text{Caused}(f, \text{false}, \text{do}(c, s)) \quad (8.7)$$

where c is a compound action.

□

Effects of concurrent actions can be also seen in terms of inheritance [9], i.e., compound actions normally inherit effects from their subactions, and cancelling actions cancel inheritance of the effects of atomic actions. For instance, if an action is not possible then unless otherwise specified, a bigger action containing that action is also not possible. A predicate *noninh/2* explicitly accounts for the non-inheritance of effects for a compound action, through so-called *inheritance axioms*.

8.1.1 Learning of concurrent actions

Learning effect axioms for concurrent actions requires a new shape for the axioms, where the action is variabilized and references to individual actions included in it are done in the body of the clause by means of positive and negative *in/2* literals. We will explicitly use positive and negative *in/2* literals instead of the predicate *Cancels/3* or inheritance axioms, given that the latter express separately the positive occurrences of actions and their preconditions from the cancelling actions, and it is not obvious how they can be learned separately. In some cases, negative examples correspond to those situations where the effect is not caused because there are preconditions that fail, actions that are missing or actions that are executed additionally.

Without concurrency, the action in the head of a clause is a single term extracted from a seed, whose arguments can be variabilized according to the bias. If any example contains concurrent actions, then the *compound* sort must be used for actions and thus the action in the effect axioms –for the fluent and the truth value in the seed– must be variabilized. Once we variabilize the action, those situations where an atomic action is executed are dealt with as a particular case of concurrent actions with a single action, as it may still need negative *in/2* literals. Thus, we will have effect axioms in the form

```
caused(F,V,do(A,S)):- in(a1,A),...
```

instead of

```
(F,V,do(a1,S)) :- ....
```

Without negative *in/2* literals, the former is a stronger generalization, as it assumes that action a_1 is not affected if executed concurrently with any other action, unless explicitly stated in data, whereas the second one does not apply to compound actions.

A bias declaration for example 11 is:

```
bias(caused(spilled,#,do(+,+)),[holds(spilled,#,+),in(#,+),not(in(#,+))]).
```

At least a positive *in/2* literal must be added initially to the body, which must be granted, for instance, with *prune/2* statements. When actions are structured terms, e.g., in the Blocks world, the construction of the \perp -clause must include the variabilization of every action both for *in/2* and *not-in/2* literals. This requires a bias to be specified also for actions. For instance, in the Blocks world, we have:

```
bias(caused(on(+,+),do(+,+)),[in(move(+,+),+),not(in(move(+,+),+)),...]).
```

thus resulting in rules like

```
caused(on(A,B),+,do(C,D)) :- in(move(A,B),C), not in(move(D,B),C),...
```

Additional pruning statements can be provided to avoid actions like *move(B,A)* if *move(A,B)* was already included, and so on.

To enable concurrent actions in the computation of the stable smodel, we rename every possible compound (c_i) –because the interpreter does not support lists– and clauses are added to the background that relate each compound to its components. For instance, for the compound $c_1 = \{t_1, t_2\}$, we have: *in(t1,c1)* and *in(t2,c1)*.

According to the bottom clause construction [95], only those actions included in the seed will appear in the \perp -clause through *in/2* literals. However, with respect to the literals *not in/2*, –used to refer to actions that must not be executed– actions must be included through an special procedure when constructing the bottom clause. This requires to retrieve actions that are *not* included in the positive example used as seed. However, the recall phase of these literals *not in/2* cannot be carried out by Prolog because when querying the Prolog interpreter, the action term a must be instantiated previously in *not in(-a,+c)*, and then it must be done through an special procedure, e.g., by adding a predicate *not_in(-a,+c)* that uses a set of actions not included in the compound c . Literals *not_in(-a,+c)* are constructed for each seed and provided to Prolog in the background before building the bottom clause.

Actions included in positive *in/2* literals are taken from the seed. The negative counterpart of *in/2* will refer to what actions should not be executed concurrently. Thus, actions included in negative *in/2* literals can be generated from the complete set of actions, from actions related somewhat to the actions executed in the seed, or from actions included in the negative examples.

In the first case, actions are retrieved from the set of possible actions, including actions that are totally independent from the action executed. In relational scenarios like the Blocks world, we need to retrieve all possible instantiations of *move/2* for the blocks constants. Furthermore, these *not in/2* literals will introduce new terms, i.e., the arguments of the actions, and the bottom clause may become very large.

In the second case, an heuristic is used to reduce the number of actions, by allowing only those actions related to the executed action or to the effect, thus focusing on actions that might

affect the executed action. For instance, by removing actions that do not share any argument with those referred to in the seed.

Thirdly, we can use the information provided by the negative examples by allowing actions not included in the seed that occur in any negative example, thus, only those actions that will allow to discriminate positive from negative examples, are considered. The set of negative examples can be incomplete, however, the relevance of the negative actions is clearer. However, not all negative examples are of interest but only those that have *any action in common* with the seed. Unfortunately, this might still include some cases where the common action(s) are not relevant in the positive example or in the negative examples. We can reduce this possibility if we dynamically adjust the process with respect to the current hypothesis at each step of specialization, so that cancelling actions are taken from those negative examples that include all actions included in the current hypothesis.

As to the negative examples, we cannot assume that any other combination of actions different from those included in a positive example does not cause the effect, because if other irrelevant actions are executed simultaneously or if any action is dropped from the example, it may still be a positive example.

8.1.2 Scenarios for concurrent actions

The task of the inductor is now to find a set of actions that must be executed (resp. not executed) and a set of preconditions, that produce a given effect. Several possible scenarios are usually considered for concurrent actions [9], that we will consider individually in the rest of the section.

- *Independent actions*. This is the most simple case, where actions can be executed concurrently and their effects are independent.
- *Cancelling actions*, i.e., the case when the effect of a compound action cancels the effect of the atomic actions. The effects of compound actions are cancelled in essentially the same way.
- *Accumulative actions*, i.e., a compound action produces effects that none of the sub-actions produces separately.
- *Conflicting subactions*, i.e., a compound action whose sub-actions have contradictory effects.

The soup bowl example is the most classical scenario of cancelling actions. In this case *LRAC* returned:

```
caused(spilled,+,do(A,B)) :-
    in(liftright,A),
    not in(liftright, A),
    holds(has_water,+,B).
caused(spilled,+,do(A,B)) :-
    in(liftright,A),
    not in(liftright, A),
    holds(has_water,+,B).
```

We observe that the actions that lift the bowl are not independent, so that any of them cancels the other. The above theory can be rewritten as follows:

```

caused(spilled,+,do(A,B)) :-
    in(liftright,A),
    holds(has_water,+,B),
    not cancels(A,liftright,B).
caused(spilled,+,do(A,B)) :-
    in(liftright,A),
    holds(has_water,+,B),
    not cancels(A,liftright,B).
cancels(A,liftright,B) :-
    in(liftright,A).
cancels(A,liftright,B) :-
    in(liftright,A).

```

In terms of inheritance, the theory can be re-expressed in the form:

$$\begin{aligned} & \text{holds}(\neg\text{spilled}, \{\text{liftright}, \text{liftright}\}) \\ & \text{noninh}(\text{spilled}, \{\text{liftright}, \text{liftright}\}, s) \end{aligned}$$

where *liftright* (resp. *liftright*) explicitly cancels inheritance of the effect *spilled* by the compound action. Baral and Gelfond [9] model the effect $\neg\text{spilled}$ as a consequence of the compound $\{\text{liftright}, \text{liftright}\}$ which is not reflected in the learned theory, because the observer does not distinguish “caused not to hold” from “not caused to hold” (inertia), when there is no change of value for *spilled*, i.e., the observer does not perceive the cancelled effect.

In the next example, we handle the case when the effect of an atomic action cancels the combined effect of a compound action. Let us consider a variant of example 11 where an additional action *flip* is added. In this case, *LRAC* learned the following theory:

```

caused(spilled,+,do(A,B)) :-
    in(liftright,A),
    not in(liftright,A),
    holds(haswater,+,B).
caused(spilled,+,do(A,B)) :-
    in(liftright,A),
    not in(liftright,A),
    holds(haswater,+,B).
caused(spilled,+,do(A,B)) :-
    in(liftright,A),
    in(liftright,A),
    in(flip,A),
    holds(haswater,+,B).

```

Action *flip* needs the combined execution of lifting the bowl to spill the soup. Actually the combined effect of the compound is a combined cancellation of the atomic actions, because *flip* does not cancel the atomic actions but the combined effect of the compound $\{\text{liftright}, \text{liftright}\}$. However this is not explicitly represented in the learned rules. In [9], *flip* explicitly cancels the inheritance of $\neg\text{spilled}$ to the compound $\{\text{liftright}, \text{liftright}\}$.

$$\text{noninh}(\neg\text{spilled}, \{\text{flip}, \text{liftright}, \text{liftright}\}, s)$$

In the next example, we handle another case when the effect of an atomic action cancels the combined *visible* effect of a compound action. Let us consider an scenario where a person, Mary,

is trying to lift the bowl, so that Mary is unable to lift a heavy bowl with one hand, while she can lift it using both hands. We add a fluent *lifted* to represent that Mary is holding the bowl and a fluent *heavy* to represent heavy bowls. In this case *LRAC* returned:

```
caused(lifted,+,do(A,B)) :-
    in(liftright,A),
    in(liftright,A).
caused(lifted,+,do(A,B)) :-
    in(liftright,A),
    holds(heavy,-,B).
caused(lifted,+,do(A,B)) :-
    in(liftright,A),
    holds(heavy,-,B).
```

If we add an action *jump* that if performed simultaneously avoids the bowl to be lifted, we obtain:

```
caused(lifted,+,do(A,B)) :-
    in(liftright,A),
    in(liftright,A),
    not in(jump,A).
```

where *jump* cancels the combined effect of the compound {liftright,liftright}. The theory can be rewritten as:

```
caused(lifted,+,do(A,B)) :-
    in(liftright,A),
    in(liftright,A),
    not cancels(A,{liftright,liftright},B).
cancels(A,{liftright,liftright},B) :-
    in(jump,A).
```

In this case, the cancellation of the compound action is explicitly included. The translation implicitly assumes that the cancelling action cancels the actions in the positive *in/2* literals. If otherwise, *jump* cancels just the atomic action {liftright}, the compound {liftright,liftright} would inherit the cancellation, so that according to [9], the clause for *cancels/3* above is not explicitly included in the theory but it is inherited from the cancellation of the atomic action. *LRAC* would learn the cancellations of the atomic action and the compound action independently.

In general, learning about concurrent actions is a complex task because the size of the theory to be learned increases significantly to express every possible interaction between actions. Even the description of very simple domains becomes significantly complex when concurrent actions are allowed. Recall example 2 and let us suppose that actions *toggle(sw_i)* can be executed concurrently. For the fluent *light*, *LRAC* returned the following effect axioms:

```
caused(active(light),+,do(A,B)) :-
    in(toggle(sw1),A),
    not in(toggle(sw2),A),
    holds(closed(sw1),-,B),
    holds(closed(sw2),+,B).
caused(active(light),+,do(A,B)) :-
    in(toggle(sw2),A),
```

```

    not in(toggle(sw1),A),
    holds(closed(sw1),+,B),
    holds(closed(sw2),-,B).
caused(active(light),+,do(A,B)) :-
    in(toggle(sw1),A),
    in(toggle(sw2),A),
    holds(closed(sw1),-,B),
    holds(closed(sw2),-,B).
caused(active(light),-,do(A,B)) :-
    in(toggle(sw1),A),
    holds(active(light),+,B).
caused(active(light),-,do(A,B)) :-
    in(toggle(sw2),A),
    holds(active(light),+,B).

```

In this case, the consequences of executing, e.g. *toggle(sw₁)*, depend on the previous value of *sw₂* distinctly depending on whether *toggle(sw₂)* is also executed or not simultaneously. Similarly for *toggle(sw₂)* wrt. *toggle(sw₁)*. When the switches are closed concurrently, they produce independent effects and the accumulative effect of activating the light, however, if one of the switches is closed and the other is opened, the light remains off. In this case, the switch that is opened cancels the indirect effect of closing the other switch.

As a consequence, many more clauses have to be learned to cope with every possible interaction between actions. Note however that the corresponding causal rules for *light* (in chapter 6) avoid this problem given that the action is also variabilized, hence the use of indirect effects greatly simplify the final theory because a single clause covers all combinations of actions. Thus, concurrent actions makes more evident the interest of learning indirect effects.

Let us recall the circuit of example (3). If we execute *toggle(sw₂)* and other action that activates the *relay* at the same time (e.g., *toggle(sw₃)*) the effect axiom infers *sw₂* (because the relay was previously inactive) whereas the causal rule infers $\neg sw_2$, because *relay* becomes active. As a consequence, the resulting theory has no stable model. Thus, the effect axiom for *sw₂* is no longer valid if other actions are executed concurrently in some states. Actually, the behavior of the circuit will depend on the propagation delays. If the relay has precedence over the switch, we need to add to the effect axiom of *sw₂* all possible cancelling actions that activate the relay (avoiding *sw₂* to close) together with their preconditions, thus making the description cumbersome in most cases.

```

caused(closed(sw2),+,do(A,B)) :-
    in(toggle(sw2),A),
    holds(closed(sw2),-,A),
    not in(toggle(sw1),A),
    not in(toggle(sw3),A).
caused(closed(sw2),+,do(A,B)) :-
    in(toggle(sw2),A),
    holds(closed(sw2),-,A),
    in(toggle(sw1),A),
    in(toggle(sw3),A)
    holds(closed(sw1),+,B).
caused(closed(sw2),+,do(A,B)) :-
    in(toggle(sw2),A),
    holds(closed(sw2),-,A),
    in(toggle(sw1),A),

```

```

    in(toggle(sw3),A)
    holds(closed(sw3),+,B).
caused(closed(sw2),+,do(A,B)) :-
    in(toggle(sw2),A),
    holds(closed(sw2),-,A),
    in(toggle(sw1),A),
    not in(toggle(sw3),A)
    holds(closed(sw3),-,B).
caused(closed(sw2),+,do(A,B)) :-
    in(toggle(sw2),A),
    holds(closed(sw2),-,A),
    in(toggle(sw1),A),
    not in(toggle(sw3),A)
    holds(closed(sw1),+,B).
...

```

Actually, this set of clauses could be replaced by an *interconstraint* in the form:

```

caused(closed(sw2),+,do(A,B)) :-
    in(toggle(sw2),A),
    holds(closed(sw2),-,B),
    holds(active(relay),-,do(A,B)).

```

where the constraint and the effect axiom are integrated into a single axiom, such that the value of *relay* in the resulting situation replaces all actions that could activate it.

In some cases, the combined execution of actions may produce complex effects very difficult to predict accurately. Let us consider the Blocks world. Compound actions like $\{move(a,b), move(c,b)\}$ and $\{move(a,b), move(b,c)\}$, etc., contain conflicting subactions. Let us suppose that the simultaneous execution of actions, e.g., $move(a,b)$ and $move(c,b)$, cancel each other out, such that if another block c is moved onto a at the same time that a is moved onto b , it is assumed that no movements are done. Then, the effect axioms for $move/2$ must cope now for many more preconditions, namely, when a block a is moved onto another block b , no other $move/2$ actions involving blocks a and b must be included in the compound action.

```

caused(on(A,B),+,do(C,D)) :- in(move(A,B),C),
    holds(on(A,B),-,D),
    holds(clear(A),+,D),
    holds(clear(B),+,D),
    not in(move(F,B),C),
    diff(F,A),
    not in(move(B,G),C),
...

```

In a real scenario, where a robot can use both arms to move blocks, it might happen that after $\{move(a,b), move(b,c)\}$, we have $on(b,c)$ and $on(a,table)$ in the resulting situation, provided $on(b,table)$ was true in the previous situation. It might also happen that after $\{move(a,b), move(c,b)\}$, the blocks bump into each other and they fall onto the table. In this case, the effects of concurrent actions may produce very complex effects that depend on many factors.

In some cases, the working of well-behaving systems can be broken when actions that cause contradictory changes can be executed arbitrarily. In [9] an example with contradicting subactions is shown where the effect of performing *close* and *open* concurrently in a door is left

undefined, i.e., the theory concludes that it is not defined whether the door is open or not after executing $\{open, close\}$. As a consequence, if an action *paint* is executed concurrently with $\{open, close\}$, the theory does not entail either *painted* or $\neg painted$, because the compound inherits the non-executability from $\{open, close\}$. In this case *undefined* is the more intuitive answer rather than inertia, because the answer must be produced by the theory. From the point of view of learning, the most intuitive answer is that produced by the observed system, thus, the representation language used for learning should be able to express it. However, in a real system, the effects of the concurrent execution might depend on small delays in the effect propagation. In fact, the concurrent execution of conflicting subactions may cause the existence of *intermediate situations* in the computation of the resulting situation, similarly to negative cycles (section 6.6.3).

8.2 Exogenous actions

We have so far assumed that changes in the values of fluents can only be caused by execution of actions, otherwise the state of the system is assumed to be stable, that is, each effect is caused, either by a primitive action or by another effect that eventually is caused by an action. If there are action sources other than the agent itself, an environment may seem active to an observer (*autonomous change*), for instance, if the domain has fluents that are non-amenable to manipulation (weather conditions), a second agent is acting in the domain, and so on. Exogenous actions are common when dealing with autonomous robots. In this kind of applications there are two kinds of actions: one that the robot can perform, and the other that may happen independent of the robot and which is beyond the control of the robot.

These exogenous actions introduce changes in a situation without an agent-initiated action, whereas in the Situation Calculus, a situation is created only by executing an explicit action. Exogenous changes can be represented through so-called *natural actions*, i.e., creating a situation with a fictitious action to cope for the changes. The special action *nat* is included when no action is observed and some fluents are seen to change. For instance, let us suppose that the action *unplug* is executed by a second agent in example 2, causing the fluent *light* to be off. Then we have a corresponding observation:

```
caused(active(light), -, do(nat, A)).
```

This topic has been paid relatively little attention in the community of reasoning about actions [104, 110]. With respect to the observer, exogenous actions must be clearly distinguished in the observations. This can be safely done when no agent-initiated action *a* is executed at the same time, otherwise the observer might assign such exogenous effects to the action executed. In general, a fluent can be both agent and autonomously initiated when some of the actions affecting to it are hidden.

From the point of view of learning, we need to be able to represent these “external” observations and in some cases to consider new shapes for the learned clauses. Exogenous effects are to be considered part of the background, so that they are not used for learning additional clauses, however, when these effects cause further changes in the domain (indirect effects), the latter can be managed like any other effect no matter the original cause. In this case, the exogenous effects have to be dealt with as if they were actions, which is out of the scope of the causal rules we considered in chapter 6. Let us consider a typical example given in [14, 26].

Example 12 (Alarm system) *There is an alarm system that detects if somehow someone enters, and that is not amenable to manipulation by the observer. We add the observer with capabilities for sensing whether the alarm is active or not and whether it is ringing or not. We assume the building has many possible entrances (doors, windows, etc.). In other words, there are many actions able to bring someone in the building and these actions may not even all be known. We formalize the system using the fluents in (stating that there is someone inside), active (the alarm is active) and ring (the alarm bell is ringing). □*

Let us first suppose that the alarm is not manipulated when someone enters the building. Under these conditions, *LRAC* learned the following effect axiom, as expected, where anyone entering the building triggers the alarm.

```
caused(ring,+,do(enter,A)) :-
    holds(active,+,A).
```

No rules are learned for the autonomous fluent *active* because it is always caused by an exogenous action. Learning from exogenous effects makes sense when we are interested in learning the source of the exogenous effects, for instance, the behavior of an external agent and so on. In this scenario, if the exogenous *nat* is considered like any other action, some clauses are learned for *active*:

```
caused(active,+,do(nat,A)) :-
    holds(active,-,A).
caused(active,-,do(nat,A)) :-
    holds(active,+,A).
```

meaning that only 2 transitions are possible for it, i.e., the hidden action inverts the state of the alarm. This is possible because the exogenous *nat* can be mapped to a single hidden action that (*de*)activates the alarm. However, there are no different *nat* actions for different hidden actions, hence in most cases learning exogenous effects is not possible.

Let us suppose the alarm does not ring if it is deactivated at the same time when someone enters. When the training data includes these “conflicting examples”, the above clause for *enter* is no longer returned. The basic form of the effect axioms is not enough, because the value of the exogenous fluent in the previous situation is not informative and the exogenous effect is not connected to the agent-initiated action. This led us to consider different shapes for learned axioms as we need a mix of effect axiom (we need the action) and a causal rule because the resulting situation must be referred to in the body of the clause, at least for the fluent *active*. We considered different shapes for learned clauses, one where the resulting situation can be used in the body of a clause, and other where negative caused literals other than the head are allowed. Depending on the bias selected, we learned different forms of the same axiom.

```
caused(ring,+,do(enter,A)) :-
    holds(active,+,do(enter,A)).

caused(ring,+,do(enter,A)) :-
    holds(active,+,A),
    not caused(active,-,do(enter,A)).
```

These axioms represent that an explanation of some effects must be searched for through other (possibly exogenous) changes in the same situation –apart from the action and the previous

situation— because a precondition for the action is modified at the same time, i.e., an exogenous effect “substitutes” the hidden action that was executed at the same time. Note that these clauses subsume the “normal” effect axiom —where someone enters and the alarm was active— apart from that where the alarm was activated simultaneously. Note also that, in both clauses, the action must be explicitly included because the axioms does not work when other actions are considered.

This example is actually due to the concurrent execution of an exogenous and an agent-initiated action. When both exogenous and agent-initiated actions co-occur, exogenous effects will be considered as effects of the agent-initiated action. For instance, *LRAC* tried to learn an effect axiom for `caused(active,+,do(enter,A))` which makes no sense, and it is not obvious how it could be avoided. For this task, the observer would need to use additional mechanisms to identify effects not related to the agent-initiated action.

Let us consider a variation of this scenario [26] where if the alarm is activated and someone was already in the building, then the alarm also rings. In this case *LRAC* obtained no generalization, i.e., examples for *ring* where *enter* is not executed were returned uncovered. The alarm is an *indirect effect* of the exogenous change in *active*. An standard causal rule is not enough because we need to refer explicitly to *caused* values in the body of the clause, otherwise no consistent clause will be returned. When *caused/3* is allowed in the body of clauses, the following rule was learned:

```
caused(ring,+,A) :-
    caused(active,+,A),
    holds(in,+,A).
```

The *caused/3* literal represents that the state of the alarm was “somewhat” changed (not necessarily by an observed action). We need explicitly *active* to be *caused* because the alarm can be stopped through an additional action, thus both *active* and *in* can be true and the alarm is not ringing. Actually, in a particular scenario, it would be possible to learn *ring* as an exogenous effect, because *nat* still corresponds to a single hidden action.

```
caused(ring,+,do(nat,A)) :-
    holds(active,-,A),
    holds(in,+,A).
```

However, the effect axiom loses part of the declarative meaning of the causal rule, and most importantly, if other hidden actions are executed that do not affect *active*, then the effect axiom becomes inconsistent.

The hardest case occurs when both an agent-initiated action and an exogenous action happen simultaneously and they affect the same fluent possibly in contradictory forms. The exogenous action might undo the effect of the agent-initiated action when actions are opposed, or it can change the outcome. In the first case, the agent will not observe any change and so it will assume the action produced no effects. As a consequence, a positive example is forced to become a negative one thus introducing non-determinism.

8.3 Sensing actions

In real environments, an agent does not necessarily perceive always all the features it can sense. For instance, in the office scenario, the robot can sense if the elevator is open whenever it is physically close to the elevator. If no exogenous actions modify it, the robot can safely assume

the last value sensed as the current one, otherwise, contradicting situations may arise. Suppose the robot opens a door and it moves to the other side of the office. When it returns, the door might (surprisingly) have been closed, thus contradicting the last value sensed. If we use the inertia assumption, the robot will assume that the door is still closed unless evidence against it is found. When the robot changes its belief, inertia must be disabled and the new observation recorded as an exogenous change. Unfortunately, when the robot moves ahead and sees the elevator, the new state of the elevator might be assumed as an indirect effect of moving. In these cases, the robot should be able to identify exogenous effects [99].

In other cases, the robot must execute some action to explicitly sense (*sensing* or *knowledge producing* actions), such that changes in the environment are not synchronized with the actions. While the sensing is not performed, a value *undefined* is to be assigned to the fluent even when the robot could physically sense it, thus, the robot must be able to represent uncertainty about the information of the sensors. For non-sensing actions, we learn effect axioms that describe the effect of the actions and for sensing actions we need axioms that describe the knowledge that may be gained by executing that action [11]. Often the various sensing that needs to be done may not be doable in all situations, thus, it would be also possible to learn *when the sensing is possible* and in some cases even the value sensed, provided the sensing can be inferred from other fluents. Consider a robot can perform the actions: *check_door_lock*, *flip_lock* and *push_door*.

- If the robot performs the action *check_door_lock*, it will know if the door is locked or not.
- If it performs *flip_lock* then the door becomes locked (resp. unlocked), if it was previously unlocked (resp. locked).
- If it performs *push_door* when the door is unlocked, the door opens.

When the sensing action is not performed, the outcome of actions that depend on the sensed information might become non-deterministic. For instance, the robot can perform the action *push_door* when it does not know whether the door is locked, and still the door might be opened in the resulting situation. Thus, any learned rule will be only valid for the cases where the robot knows explicitly that the door is locked, whereas the decision for the rest of examples becomes undefined.

Unlike hidden actions, sensing actions are executed by the own agent to explicitly sense a piece of information. However, in some cases, the sensing actions can be expensive and their execution must be minimized. In this case, learning must cope with the uncertainty in the values of fluents.

8.4 Actions with duration and delayed effects

We have so far assumed that all atomic actions are instantaneous and have instantaneous effects. In real domains, we can find other types of actions:

- Durative actions.
- Actions with delayed effects.
- Actions with sequential effects.

The topics of delayed effects and actions with duration have received so far little attention in the Reasoning about Actions literature [57, 48]. In general, a theory of actions in which all actions and the effects are instantaneous is not enough to cope with some real domains.

Most actions take time, e.g., picking up a block, going from one location to another. The *trick*, as observed in [104], is to conceive of such actions as *processes* represented by fluents, and to introduce *durationless* actions which initiate and terminate these processes. This allows to represent actions with duration and still respect the form of the Situation Calculus programs. Let us consider the action of picking a block. We have instantaneous actions $start_pickup(x)$ and $end_pickup(x)$, and the process of picking up x is represented by the fluent $picking_up(x)$. Action $start_pickup(x)$ causes the fluent $picking_up(x)$ to be true, $end_pickup(x)$ causes it to be false. In those situations at which $picking_up(x)$ is true, we can describe those properties of the world which must be true *during* the evolution of the process $picking_up$, or even, another action could interrupt the process while it is being executed.

This allows that a big deal of behaviors are representable in the Situation Calculus. For instance, assume a room has a door with a spring latch. This is easy to do in the Situation Calculus if we view the action of turning and holding the latch open, which intuitively would have a duration, as a composite of two instantaneous actions, $turn_latch(t)$ and $release_latch(t)$. The door can be opened by turning the latch, but the agent must *keep the latch turned*, i.e., the action must be executed for some time, for if not, the spring loaded mechanism returns the latch to its original position. The “concurrent” latch turning and door pushing causes the door to open.

On the other hand, *effect propagations* usually incorporate very small delays. If some change occurs after the end of an action, then there must be some underlying process going on. By considering this change as a delayed effect of the action, one can sometimes abstract away the details of the process and still be able to obtain an adequate description [57], that is, assuming that the effects are simultaneous and instantaneous. Abstracting small delays is often convenient because the resulting model is simpler or there is no actual knowledge on the delays for providing an accurate model, otherwise they must be explicitly represented. The problem is to find out when this is the case. In general, the delays should be considered explicitly in a theory of delayed causation.

Some extensions exist to the Situation Calculus that model delayed effects similarly to durative actions. When a fluent is known to be a delayed effect, we can use a similar strategy to that used for actions with duration, where a new fluent is added to mean that the effect is “in progress”. This is necessary if other actions can happen in-between that cancel the effect. An example due to Gelfond, Lifschitz and Rabinov of a delayed effect is a pedestrian light, which turns green 30 seconds after one presses the button at the crosswalk. This domain is represented in the narrative formalism TAL-C [57] as follows (C_T should be read as “changes to true”, $[t, t + i]\phi$ denotes that ϕ is true along the interval, whereas $[t]\phi$ refers to a single time-point):

- $dep1 \quad C_T(\text{pressed} \wedge \neg \text{tick}) \Rightarrow R([t + 1]\text{tick})$
- $dep2 \quad C_T([t]\text{tick}) \wedge [t, t + 29]\text{tick} \Rightarrow R([t + 29]\text{color}=\text{green})$
- $dep3 \quad C_T([t]\text{tick}) \wedge [t, t + 58]\text{tick} \Rightarrow R([t + 59]\text{color}=\text{red} \wedge \neg \text{tick})$

Press is a durative action, so that Pressed is a fluent that will be true while the Press action is performed, that starts the “ticking”. If the ticking goes on for 30 seconds, the pedestrian light becomes green ($dep2$). Pressing the button a second time will not result in a second period of green. This represents an interaction between a delayed effect of one action and a second intermediate action.

Finally, we can also find actions that produce *sequential effects* over time, where the propagation of effects takes place in a particular ordering. When the intermediate situations have to be explicitly recorded, the action theory must predict the sequence which requires a theory with explicit delays. In cases like a spring that immediately comes to its initial position, the delay between the effects must be explicitly stated, otherwise the resulting state would be inconsistent. Similarly, in the pedestrian light example, we have that the light switches back to red after 20 seconds (*dep3*).

However, the ability to represent the effects of durative actions or delayed effects, is not enough. In general, they represent a harder problem from the point of view of the observer than for the representation methods. The observer must know which of the actions need some time to execute, otherwise no instantaneous effects will be observed, the execution will be labeled as failed, and the eventual effects will not be related to the action. Furthermore, from the point of view of an observer, it is not clear if we can consider a fluent that changes without explicit action as a delayed effect or as an exogenous change. For long delays, we still do not know which of the previously executed actions is responsible for it, for instance when some situations are created between the cause and the delayed effect. However, this requires advanced identification methods and the ability to make experiments in the domain.

8.5 Conclusions

To our knowledge, concurrent actions have not been paid much attention when learning the effects of actions. However, most scenarios in the real world include the concurrent execution of basic actions, hence it is important for a learning agent to be able to represent and learn from them. In general, concurrent actions make the learning process harder, where additional actions (cancelling or not) are to be used as preconditions for the executed actions. In this chapter, we have shown how to incorporate concurrency to the Situation Calculus to learn the combined effects of multiple actions and showed it in small representative domains that cover the different types of interactions between actions shown in the literature.

On the other hand, it is important to have a representation method that allows to represent and reason with exogenous actions. In particular, the Situation Calculus deals with these actions and their effects in a very natural form. We have also seen how it is sometimes possible to learn about such exogenous effects. But, in practice, the quality of the learning results is intrinsically related to the capabilities of the observer. The major task of the observer is to identify the effects of his actions from exogenous effects caused by hidden actions.

Similarly, actions with duration or delayed effects are representable into the Situation Calculus and other action formalisms, however, the main difficulty is to know which actions have such durations or delays. Without this knowledge, the assumption that actions are instantaneous and have instantaneous effects may produce invalid theories [15, 103]. The learning of preconditions for actions must be combined with a procedure for deciding which action was responsible for a particular effect, however, this may result difficult in many cases. The ability of an agent to interact with the domain during learning may reduce the uncertainty, through the execution of additional experiments.

Chapter 9

Learning declarative control rules for planning

In this chapter, we consider the task of learning action-selection rules for planning where both plans and the control rules are represented in the Situation Calculus. We show how the problem of learning action theories and the problem of learning action selection rules relate to each other. The use of Action Languages has a significant impact in the generality of the approach, so that both tasks can be approached in a similar way.

9.1 Introduction

In AI Planning, a planner is given an initial state and a goal, and finds a sequence of actions that maps the state into the goal. For instance, turning a configuration of blocks into another is a planning problem in the Blocks world. Planners generally search through a list of domain actions until a correct sequence of those actions has been found that can achieve the desired goals. This problem has been tackled by a number of algorithms and in recent years substantial progress has been made [133, 59]. However, the problem is still computationally hard and the best algorithms are bound to fail on certain classes of instances.

An alternative that has been proposed is to use knowledge of the planning domain for guiding the planning process [113, 2, 51]. Planners that rely on domain-dependent control knowledge can outperform the best domain-independent planners. Search control information for the blocks-world may say things like

‘pick up a *misplaced* block if clear’

‘put current block on destination if destination block is clear and well placed’

etc.

This control is completely domain-dependent but it is completely independent of particular scenarios of the blocks world [76]. Similar control rules can be defined for many planning domains. These hand-coded declarative constraints do not make explicit reference to the workings of the planner, but only refer to the solution space like the constraints that define the original problem instance, so that they can be used by fundamentally different planning architectures [51]. Furthermore, these constraints can provide dramatic reductions in solution times, for instance, whereas 11 or 12 blocks seem to be the limit of current planners in the blocks world [2], the use of such control rules allows planners to deal with instances involving up to 20 blocks in less than a

minute. Declarative control achieves its objective in a very elegant way because the control rules are quite intuitive and purely declarative. Such control knowledge is not always easy to provide, in fact, developing the right control formulas is a non-trivial task even for simple domains.

Many planners include basic control rules to, e.g., avoid that an effect achieved in the current situation is removed in the next situation. However, even such a basic control cannot be assumed generally. For instance, in the blocks world, it does not make sense to achieve $on(a, b)$ and $\neg on(a, b)$ in two consecutive situations, however it does with $clear(a)$ and $\neg clear(a)$, where block a is cleared to put another block onto it. Some useful information can be extracted from an analysis of the domain description in the form of invariants, type information, symmetry properties and so on [114, 40] that helps reducing the search. However, the above control information for the blocks world is not included in the domain description, because it depends on a planning problem, e.g., *well_placed* is not used to predict the effects of moving blocks.

A natural question that was posed is whether this kind of declarative control knowledge can be learned. Recent results have shown that it is possible to automatically acquire high level declarative rules for action selection in planning in a purely declarative way. This problem has been very recently tackled by Kautz and Selman [52], Geffner [76] and Khardon [60], among others, using different schemes for representation and learning. More precisely, they deal with the problem of learning how to solve a planning problem in a domain, given solutions to a number of small instances of it. Learned rules must infer the action to be executed based on the current situation where the action will be executed and both the initial and the goal situation, so that the control encodes the solutions to a class of problems that differ on both the initial and the goal situations.

Previous works focused on learning control rules that are specific to the details of the underlying planner being used [22, 131, 64], hence the control rules cannot be reused by other planners. Declarative control makes it possible to separate the control information from the implementation of the planner which is a clear benefit [113]. Other approaches require the user to explicitly supply specialized background knowledge to the learner [60, 34]. For instance, we need to introduce extra predicates like *misplaced*, to represent the idea that there is a block that needs to be moved. However, the complexity of building these predicates is at least comparable to that of learning the control rules. In fact, the task of learning is somewhat to discover such predicates that relate the current situation with the goal situation. Thus, background knowledge for learning involves only the primitive predicates that appear in the domain description.

In this chapter, we consider the task of learning action-selection rules using Action Languages, where both plans and the control rules are represented in the Situation Calculus. We show how the problem of learning action theories and the problem of learning action selection rules relate to each other. Indeed, learning action selection rules can be seen as the inverse of the former, where the theory to be learned is included in the background, and the learned rules must infer the action to be executed based on the situation where the action will be executed and both the initial and the goal situation, rather than predict the resulting situation, based on the action executed and the current situation. For the problem of learning action-selection rules it is assumed that a correct action theory is provided.

We believe that Action Languages constitute a natural framework for this task. A language that makes the notion of *situation* more central can provide a more compact description of the observations, where multiple plans with different goals starting at different initial situations are represented in the Situation Calculus in a homogeneous way. We also show how it is possible to easily incorporate a form of temporal logic to enhance the expressivity of the action selection rules, with respect to previous approaches. And lastly, we analyze in detail the benefits and

drawbacks of this approach and outlook future improvements.

9.2 Action-selection rules

There are two possible approaches to the task of learning search control rules. One where the objective is to learn the planner, i.e., the control rules will substitute the planner after learning [60, 76]. This requires thousands of training examples and long training times. Furthermore, the control to be learned can be quite complex and thus hard to learn. In the second approach the control rules assist [52] but not replace search. We will focus on the latter approach.

Control rules found in the literature [2] are usually in the form:

```
when we move a vehicle, one of these conditions should be met:
  the next location is the goal location for the vehicle
  or there is an object such that:
    the object is at the new location and needs a pick-up
    or the object is in the vehicle and needs to be unloaded
otherwise the vehicle is not moved
```

Kautz and Selman [52] learn for each action in the planning domain, two complementary concepts, “select action” and “reject action”.

- “Select rules” indicate conditions under which the action must be performed.
- “Reject rules” indicate conditions under which it must not be performed.

They also distinguish two classes of rules to be searched, *static* and *dynamic*. Static rules allow references only to the initial and goal situation, whereas dynamic rules refer also to the current situation where the action will be selected/rejected. Static rules thus, determine actions that should be selected/rejected always given a planning problem, whereas dynamic rules need also the current situation to infer the best action to execute. Dynamic rules are needed when actions have to be executed in sequence. The intuition of all these rules is that they encode information about the *difference* between the current (and the initial) state and the goal state.

The use of explicit “reject” terms can be argued to be unnecessary, because they will theoretically represent the negation of the conditions for the *select* rules. Without explicit reject rules, those actions not selected are implicitly rejected. However, this needs that learned select rules are complete, which is a strong assumption in most cases, when relevant information is missing or the control required is complex and thus hard to learn. In general, *reject* rules allow some degree of *uncertainty* in the search control rules. For instance, a planning agent executing the action-selection rules can know that a particular action must be rejected and still does not know which action should be selected, i.e., *select* is not defined. We will also see that some control is easier to learn for the reject rules than for the select rules and viceversa.

Khardon [60] deals with deterministic control rules where a unique action is selected at each situation. We deal with non-deterministic control, where possibly several actions are eligible at one situation, however, the planner is not forced to execute them all concurrently. Multiple actions can be eligible at a situation, for instance, when multiple subgoals can be achieved in the next situation or a subgoal can be achieved through more than one action.

9.3 Extraction of examples from plans

Unlike traditional approaches that find examples of the select and reject concepts by examining a trace of the planner, Kautz and Selman extract examples heuristically from solved problem instances. First of all, a planner is used to generate a set of plans for *some small instances* of a planning problem, and a set of positive and negative examples are extracted from the plans. Ideally, positive examples should correspond to any action that *contributes* significantly to the achievement of the goal, by causing some needed intermediate effects, such that the goal becomes "closer". On the contrary, negative examples should be those actions that make impossible to achieve the goal, or at least undo some intermediate effects, or introduce redundant states, making the plans unnecessarily long. In practice, the extraction of examples is purely *heuristic* and depends highly on the quality of the plans used. In [52] the heuristics is based on the notion that there is a good chance that the particular actions that appear in an optimal solution must be selected, and those that do not appear must be rejected. For the extraction of examples, Kautz and Selman distinguish three kinds of actions:

1. A **real** action was executed at situation s of a plan.
2. A **virtual** action was not executed at situation s , and however its preconditions hold at s .
3. A **mutex virtual** action was not executed at situation s , its preconditions hold at s but it is incompatible with any of the actions selected in s .

For dynamic rules, positive examples are actions of class *real* and negative examples are of class *mutex virtual*. The insight is that examples based on actions that do not occur are less reliable than ones based on actions that do occur. Mutex virtual actions tend to be more relevant than others because their preconditions and effects overlap with those of actions that were executed. The condition of "executable" for negative examples is important, otherwise the "select" rules will replicate the executability conditions of the actions. Two types of errors may appear:

- False negatives or reject "good" actions.
- False positives or select non-optimal actions.

The quality of the plans [68] used for learning is a key factor. Ideally, plans should be *optimal*, i.e., without repeated or useless states, to avoid false positives. This means that the length of the plan must be minimal. The use of optimal plans restrict in some cases the training set to small instances of planning problems, otherwise finding optimal plans is a costly operation.

If the plans are not optimal, false positives might be generated. False negatives are produced when there is a need for an action and this is not executed. To avoid false negatives, an action should be selected at any situation where the action is relevant and could be executed (possibly concurrently), although there are other actions that are also applicable. For instance, when considering plans of length l , false negatives may appear due to the existence of *alternative plans* of the same length. In this case, several actions are *eligible* at a situation, hence the alternative plans should be provided. Alternative plans of different length, such that no one *subsumes the other* are not considered, because they are not optimal plans¹.

¹We have that a plan p subsumes p' if the former can be obtained from p' by removing some segments of p' .

If the alternative plans are not provided, then a heuristics like the mutex virtual condition is essential to minimize the effect of incorrectly labeled examples. However, the mutex heuristics alone is not strong enough and false negatives can still be produced. For instance, in the logistics domain (section 9.4.4), when a truck must be used to drive a package to a different location and there are two trucks at the current location of the package, the planner will just take one of the trucks. As the actions are mutually exclusive –a package can be loaded only into one truck– a negative example will be produced for the truck that is not used. This false negative can be only avoided if the alternative plan is provided.

9.4 Learning action-selection rules in the Situation Calculus

In this section, we adapt the approach used for learning action theories for the purpose of learning action-selection rules. In this case, input data corresponds to plans –situations in the Situation Calculus–

$$s_g = do(a_1, do(a_2, \dots, s_0) \dots)$$

such that $holds(goal, s_g)$ is true, where $goal$ represents a conjunction of fluent literals. We create an special term *select* (resp. *reject*) to be added to every situation of each plan, that can hold or not at different situations depending on whether an action was selected or rejected. The goal has not to be exactly the same through all plans. For instance, in the blocks world, a goal state corresponds to a particular arrangement of the blocks, that can involve different positions for each particular block. However, the plans must correspond to a same planning problem, e.g., turning one configuration of the blocks into another. For this reason we add a constant g_i to represent each goal situation, so that information about the goals can be represented as conventional facts instead of using a *goal modality*. Examples are represented by ground facts of the form

$$select(a, p(s_0^j, g_j), s_i) \mid reject(a, p(s_0^j, g_j), s_i)$$

to mean that an action a was executed (resp. rejected) at situation s_i at any plan $p_j = p(s_0^j, g_j)$, where s_0^j is the initial situation of the plan and g_j is the goal situation.

Terms *select/reject* are represented as special fluents in the Situation Calculus, in the sense that they are not subject to the inertia law. The reason is that it cannot be generally assumed that an action must be selected unless another rule explicitly says that it should be rejected. On the contrary their values do not persist from one situation to the next, but depend on the inertia of the conditions in the body, i.e., there is still a need for the action.

Now, it is straightforward to adapt the form of the Situation Calculus programs to be learned for the action-selection task.

Definition 9.1 (Action-selection rules in the Situation Calculus) The form of the action-selection rules in the Situation Calculus is:

$$select(a, p(s_0^i, g_i), s) \leftarrow \pi^+ \tag{9.1}$$

$$reject(a, p(s_0^i, g_i), s) \leftarrow \pi^- \tag{9.2}$$

where π^+ (resp. π^-) does not mention the *select* (resp. *reject*) predicate and every occurrence of the *holds/3* predicate in π^+ (resp. π^-) is of the form $[\neg]Holds(F', s)$, $[\neg]Holds(F', g_i)$ or $[\neg]Holds(F', s_0^i)$. \square

The description states that, in any situation, if the precondition holds then the action a will be selected (resp. rejected). The final set of learned rules is output in the form of logic programs which can be used by either the original planner or a variety of other recent planning engines.

The goal situation is usually a partial state, so that no particular value is assumed for non-specified facts. In this case, no references to negative goal literals are considered. The execution of a plan might complete the goal situation with values for other fluents that are not included in any goal condition, however, we cannot include conditions of the goal situation that were not provided in the definition of the planning problem, because these values are indirect results of a particular plan. The representation of the goal situations as special situation constants not connected to particular plans avoids it². The initial situation needs not either be assumed to be complete. In this sense, it would be also feasible to consider explicit undefined values for fluents in the control rules if it contributes to a better planner. Every plan includes as background a (possibly partial) description of the situations of each plan which is "computed" by the execution of the action theory in the situations of the plans. For this reason, we use as background the initial situations, the goal situations, the action theory of the domain and the inertia axiom.

We formally define the problem of learning action selection rules in the Situation Calculus.

Definition 9.2 (Learning Action Selection Rules in the Situation Calculus)

Given:

- A *domain description* consisting of two nonempty sets: a set \mathcal{F} of fluent names, and a set \mathcal{A} of action names.
- A set of plans p_{ij} starting at situation s_i and with goal g_j
- A set $E_{a_k}^+$ of positive examples (ground facts) $select(a_k, p_{ij}, s)$, representing observations for each action a_k that was executed in any situation $s \in p_{ij}$.
- A set $E_{a_k}^-$ of negative examples (ground facts) $reject(a_k, p_{ij}, s)$ representing observations for each action a_k that was rejected in any situation $s \in p_{ij}$.
- Background knowledge (BK), including *holds/3* ground facts for fluents at every initial situation s_i and every goal situation g_j , the action theory of the domain in the form of a Situation Calculus program together with axioms (6.6,6.7,6.8,6.9).

Find the most general Situation Calculus program $H = (\cup_{k=1}^n H_{a_k}^{sel}) \cup (\cup_{k=1}^m H_{a_k}^{rej})$ composed of axioms in the form 9.1 and 9.2, such that:

$$(\forall e^+ \in E_{a_k}^+) \quad BK \cup H_{a_k}^{sel} \models e^+ \quad (9.3)$$

$$(\forall e^- \in E_{a_k}^-) \quad BK \cup H_{a_k}^{rej} \models e^- \quad (9.4)$$

and respectively

$$(\forall e^- \in E_{a_k}^-) \quad BK \cup H_{a_k}^{sel} \not\models \neg e^- \quad (9.5)$$

$$(\forall e^+ \in E_{a_k}^+) \quad BK \cup H_{a_k}^{rej} \not\models \neg e^+ \quad (9.6)$$

where $\neg reject \equiv select$ and viceversa. □

²An exception is those facts resulting of the application of constraints to the goal situation, e.g., $\neg clear(b)$ if $on(a, b)$ is true.

The consistency condition avoids that *select* and *reject* rules overlap in the training set. However, it might happen that search control rules overlap –when applied to plans not used for training– such that an action is both selected and rejected. These contradictions can be managed by making rules mutually defeasible so that no defined answer is given for the action. It would be also possible that multiple actions are eligible at one situation, that are mutually exclusive. In this case, without additional control, the planner will take one of the actions non-deterministically.

Control rules are non-recursive logic programs, hence extensional coverage testing can be used to test the generality of the hypotheses. The description of the situations included in the plans can be derived intensionally from the background. For efficiency reasons, we can use the *caused/3* ground facts that describe the effects produced by the actions in the plans, instead of the action theory of the domain, together with the inertia axiom. This avoids that many *holds/3* facts are repeatedly derived to test the coverage of the rules.

9.4.1 Using a planner to generate the training set

A new generation of planning systems that formulate planning as a *constraint satisfaction problem* has appeared that improve both speed and scalability [59]. *Answer set planning* [66] differs from satisfiability planning in that it uses Logic Programming rules instead of propositional formulas. The key element of answer set planning is the representation of a planning problem as a program whose answer sets represent possible plans. An answer set planner operates by searching plans that fit within a certain number of steps.

The planner we have used for generating training plans invokes a system for computing answer sets (*smodels* [97]), several times with different values of n (length of the plan) until some solution is found. Thus, we consider only optimal plans. Only planning problems with solutions under a certain length were used to ensure that all problems could be solved in a reasonable amount of time.

Plans for learning were generated by using a narrative implementation of the Situation Calculus, so that every stable model of the program is a possible plan, i.e., a narrative. A predicate *happens/2* is added, similarly to the Event Calculus, to indicate the actions that happened at each time point in a narrative. Action selection is carried out by so-called *choice rules* which is a short-form for disjunctive rules, which non-deterministically select one of the executable actions.

```
situation(0..n).

{ happens(move(B,L),S) } :- block(B),location(L),situation(S),
                             executable(move(B,L),S).

executable(move(A,B),S) :-
    situation(S),block(A),location(B),A!=B,
    holds(clear(A),+,S), holds(clear(B),+,S),
    holds(on(A,B),-,S).
```

The specification of a planning problem consists of the descriptions of the initial and goal situations. For instance:

```
init(s0).                                goal(g0).
holds(on(a,b),+,0).                       holds(on(b,c),+,g0).
holds(on(b,table),+,0).                   holds(on(a,table),+,g0).
```

```

holds(on(c,table),+,0).      holds(on(c,table),+,g0).
holds(clear(a),+,0).
holds(clear(c),+,0).

```

Plans are extracted from each stable model and converted to the non-narrative Situation Calculus before learning, so that a situation constant is used for the initial and goal situations of each plan.

```

Answer: 1
Stable Model:
happens(move(d,table),1) happens(move(a,table),2) happens(move(d,a),3) ...
holds(clear(c),-,0) holds(clear(a),-,0) ...
Answer: 2
Stable Model:
happens(move(d,b),1) happens(move(a,table),2) happens(move(d,a),3) ...
holds(clear(c),-,0) holds(clear(a),-,0) ...

```

The algorithm to extract positive and negative examples from these plans is depicted in Fig. 9.1, where positive examples are real actions and heuristics like the mutex virtual condition are considered in step 6, which also considers the existence of alternative solutions to a same planning problem.

1. initialize queue Q_p to contain all plans $p(s_0^i, g_i)$ such that no two plans subsume each other.
2. remove a plan p from Q_p .
3. initialize queue Q_a to contain all pairs (a_i, s_i) where a_i is an action executed at situation s_i of p .
4. remove a pair (a, s) from Q_a .
5. add $select(a, p, s)$ to E^+ .
6. add $reject(a', p, s)$ to E^- for all actions $a' \neq a$ that are (mutex) virtual at s and $\exists (a', s)$ in an alternative plan.
7. unless the queue Q_a is empty goto 4.
8. unless the queue Q_p is empty goto 2.

Figure 9.1: Extraction of examples from plans

9.4.2 Using learned control rules to speed-up planning

The learned declarative constraints can be used by fundamentally different planning architectures [51]. However, the form that the learned control is actually used depends on the particular planner. For instance, control knowledge can be incorporated to an answer set planner in a purely declarative form by encoding it as additional constraints [66] that are specified as separate rules and are simply added to the domain description. For instance, in the Blocks world, we can use *select* rules as conditions for choice-rules whereas *reject* rules are used as constraints to avoid rejected actions to be executed as well as a possible overlap of select and reject rules.

```

{ happens(move(B,L),S) } :- block(B),location(L),situation(S),goal(G),
                           select(move(B,L),S,G).

```

```
:- block(B),location(L),situation(S),goal(G),
   happens(move(B,L),S),reject(move(B,L),S,G).
```

By doing so, possibly several actions are eligible at one situation, however, the planner is not forced to execute them all concurrently. In case of overlap of select and reject rules, undefinedness is forced to avoid the contradiction.

The use of learned control rather than hand-coded one may introduce incompleteness and unsoundness into the planner, so that, when faced with a new instance of a planning problem, the planner might fail to find a solution. Ideally, selected actions should be sufficient for solving any instance of the planning domain. However, when no rules are applicable, or when the rules are unsound, the planner might fail on some instances. Multiple causes are possible. Sometimes, the training plans are not representative enough, or the noise in the extraction phase is high, or it is just that the control to be learned has a complex representation to be learned. In general, learning control rules that replace the planner is difficult in general, so that some search is unavoidable.

The alternative is to execute the planner without control rules. The existence of explicit reject rules provides a third category of actions apart from selected and not selected actions, i.e., non-rejected actions. These are to be given preference over the rejected actions. Thus, several situations are possible when applying the control rules in practice.

- Execute selected actions.
- Execute non-rejected actions.
- Execute all executable actions.

In using non-rejected actions, the planner is not so dependent on the completeness of the select rules.

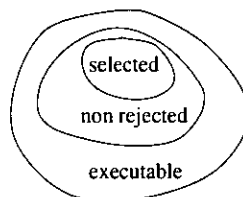


Figure 9.2: Application of the learned control

Despite this, unsound reject rules might still make the planner incomplete. In this case, the control rules have to be refined with the failing plans. A refinement phase of learned rules is an important factor to produce a robust planner, as showed in [52, 76].

9.4.3 The Blocks world

Let us consider a possible planning problem in an scenario of the blocks world. The scenario includes a description of the initial and goal situations (Fig. 9.3).

All possible plans of length 2 for this scenario are:

```
p1: do(move(b,a),do(move(a,table),s0))
p2: do(move(b,a),do(move(a,c),s0))
```

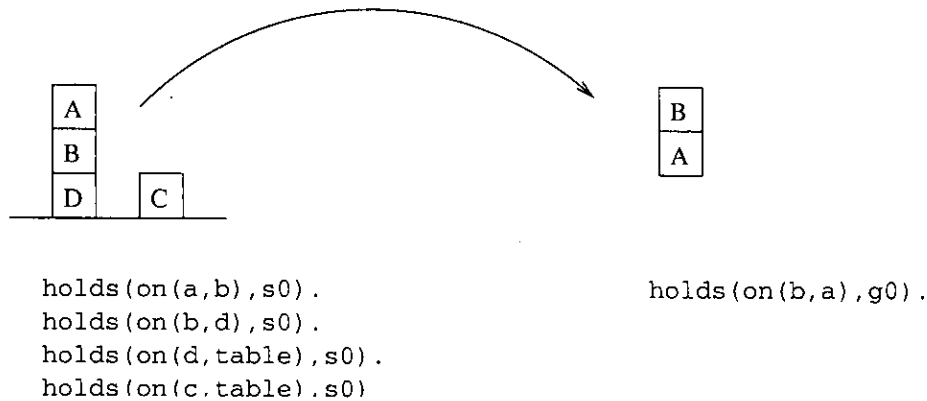



Figure 9.3: A possible plan in the Blocks world

From these two plans we can extract the following examples.

```

select(move(a,table),s0,p1).
select(move(b,a),do(move(a,table),s0),p1).
select(move(a,c),s0,p2).
select(move(b,a),do(move(a,c),s0),p2).

```

We see that two actions are eligible at situation s_0 . Similarly we can generate the corresponding negative examples, among others:

```

reject(move(c,a),s0,p1).
reject(move(b,table),do(move(b,a),s0),p1).
...

```

The negative example `reject(move(a,table),s0,p1)` is not considered because it is a positive example in the alternative plan p_2 . Following the mutex heuristics,

```
reject(move(c,a),do(move(a,table),s0),p1)
```

is a negative example, however

```
reject(move(c,e),do(move(a,table),s0),p1)
```

(where e is a clear block not related to the goal) is not considered because it is not incompatible with `move(b,a)`.

Background for learning is enlarged with domain predicates *block/1* and *table/1* as well as the predicate *diff/2*. A bias must be specified for the select/reject rules and for each action. The use of structured actions and fluents makes possible to consider several levels of generalization. For instance, it would be possible to allow *constants* in the place of variables thus focusing on particular objects of the domain.

```

bias(select(move(+,+),+,+),
      [holds(on(+,#),#), holds(on(+,-),#), holds(clear(+),#),
        diff(+,#),table(+),not(table(+))]).

bias(reject(move(+,+),+,+),
      [holds(on(+,#),#), holds(on(+,-),#), holds(clear(+),#),
        diff(+,#),table(+),not(table(+))]).

```

Valid control rules must contain at least one literal of the goal situation³. The built-in predicate `remove/3` can be used to remove from search those hypotheses that are not well-formed.

```
remove(select(_,_,G),Body) :- not in(holds(_,_,G),Body).
remove(reject(_,_,G),Body) :- not in(holds(_,_,G),Body).
```

In the planning literature, the use of control rules is enhanced with the inclusion of additional predicates, so that the complexity of the temporal formulas is kept small. From the point of view of learning, where we do not assume the presence of such background predicates, e.g., that a block is *well-placed* and so on, rules need additional literals to express the same meaning, thus some of the clauses may be lengthy. Some literals are added just to introduce new objects in the clause, whereas others express some properties about those objects. For this reason, we need to allow a high number of literals in the body of rules. A very important factor is then to constrain the search space as much as possible. This means to make the bottom-clause as short as possible without removing the solution from the search space and consider only well-formed clauses.

Prune statements are extremely useful for stating which kinds of clauses should not be considered in the search. For instance, the bottom clause might also contain those literals that are strictly derived from the executability conditions of the actions. As such, they will not be used for discrimination –negative examples always correspond to executable actions that were not executed– but increase the search space significantly. They should be kept only when they introduce additional relevant variables to those of the action in the bottom clause. For instance, in the following rule:

$$\text{select}(\text{move}(A, B), \dots) \leftarrow \text{clear}(A), \dots$$

`clear(A)` is a condition for the execution of `move(A, B)` and no new variables are introduced. These literals can be removed from the bottom clause before search.

On the other hand, so-called *invariants* are statements that are true in the initial situation and whose truth is preserved by the application of every action. Algorithms for computing such invariants have been given, among others, in [114]. These invariants can be of help to prune the space search. Such information can be provided to the learning method in a declarative form through mode declarations or prune statements. Let us consider the following invariants for the Blocks world in Fig. 9.1.

$$x \neq z \rightarrow \overline{\text{on}(z, u)} \vee \overline{\text{on}(x, u)} \quad (9.7)$$

$$y \neq u \rightarrow \overline{\text{on}(z, u)} \vee \overline{\text{on}(z, y)} \quad (9.8)$$

$$\rightarrow \overline{\text{on}(y, y)} \quad (9.9)$$

$$\rightarrow \overline{\text{on}(z, u)} \vee \overline{\text{on}(u, z)} \quad (9.10)$$

$$\rightarrow \overline{\text{on}(y, z)} \vee \overline{\text{on}(y, \text{table})} \quad (9.11)$$

$$\rightarrow \overline{\text{clear}(z)} \vee \overline{\text{on}(x, z)} \quad (9.12)$$

Table 9.1: Invariants for the Blocks world

For instance, invariant (9.9) deals with literals in the form `holds(on(A, A), -, S)`, which are not of interest and introduce an overhead for the search process. To avoid these useless literals, we add a prune statement like the following.

³We only considered dynamic rules because static rules are a particular case of the former. Thus, no references are made to the initial situation of the plan.

```
prune(Head,Body) :- in(holds(on(A,A),_,_),Body).
```

Invariant (9.10) avoids *symmetric references*.

```
prune(Head,Body) :- in(holds(on(A,B),_,C),Body),in(holds(on(B,A),_,C),Body).
```

Invariants (9.7, 9.8 and 9.11) represent typical functional dependences, e.g., once a block z is onto a block u , the former cannot be onto a third block y , and so it is redundant condition.

```
prune(Head,Body) :- in(holds(on(A,_),+,B),Body),in(holds(on(A,_),-,B),Body).
prune(Head,Body) :- in(holds(on(_,A),+,B),Body),in(holds(on(_,A),-,B),Body).
```

Further pruning conditions are derived from relations among fluents (9.12).

```
prune(Head,Body) :- in(holds(on(A,B),+,C),Body),in(clear(B),_,C),Body).
prune(Head,Body) :- in(clear(B),+,C),Body),in(holds(on(A,B),_,C),Body).
```

The training set consists of the optimal solutions (including alternative plans) to a small number of planning problems consisting of up to 4 blocks requiring between 2 and 6 steps. Under these conditions, 6 rules were learned that covered 42 of the 47 select examples and 164 of the 169 reject examples. The rest were returned without generalization⁴.

```
select(move(A,B),C,D) :- holds(on(A,B),+,D),table(B).
select(move(A,B),C,D) :- holds(on(A,B),+,D),
                        holds(on(B,E),+,C), holds(on(B,E),+,D).
select(move(A,B),C,D) :- holds(on(A,E),+,C), table(B), holds(on(F,E),+,D), diff(A,F).

reject(move(A,B),C,D) :- holds(on(B,A),+,D).
reject(move(A,B),C,D) :- holds(on(E,B),+,D), diff(A,E).
reject(move(A,B),C,D) :- holds(on(B,E),+,D), holds(on(B,E),-,C).
```

The meaning of the rules is:

- The first rule represents the particular case where the table is the goal location of a block.
- The second rule selects the action because the immediate effect also holds in the goal and block B is well placed.
- In the third rule, block A should be moved onto the table if it is currently on a block E and there is a third block F that is on E in the goal. This rule is particularly interesting as the table is always a clear destination for blocks that must be temporally moved to any location different from the current one.
- The *reject* rules just avoid that additional actions are executed. In particular, the second *reject* rule rejects the action because a different block other than A should be over B , and in the third *reject* rule because block B is not well placed according to the goal.

⁴ILP algorithms compute accuracy over literals of the target predicate, e.g., *select/reject*. Theoretically, a rule might be very frequent in a single plan but rarely applicable in others, thus the support of the rule is very dependent on that plan. It is important that action-selection rules are general across multiple and diverse plans.

Some rules are missing, for instance, one to express that a block must be placed onto the table because the block immediately below is to be clear in the goal situation or on a different location. The analysis of the plans used for training revealed that these situations were not very commonly represented.

However, most importantly, the rules are not generally applicable when more than two blocks are to be piled up in the goal situation. When this is the case, the condition “well placed” of the first select rule does not work properly, and we should obtain different versions of the above rules for towers of four, five, etc.. blocks. As pointed by Khardon [60], a partial stack of blocks may have all its goal conditions satisfied while being above a block that must be moved since it belongs to a different stack. In such case all these blocks must be moved (Fig. 9.4).

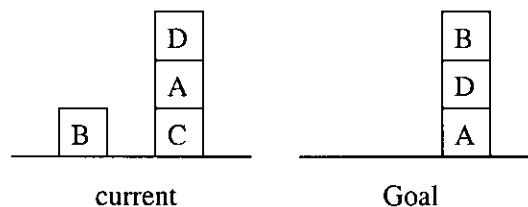


Figure 9.4: Another planning problem in the Blocks world

This can be avoided if the rules that select the blocks to be moved, move the blocks in the correct order [76]. For instance, in Fig. 9.4, a possible plan is:

$$\{move(d, table), move(a, table), move(d, a), move(b, d)\}$$

In this case, the subgoals to achieve –i.e., the facts that are different in the initial and goal situations– are $on(b, d)$ and $on(a, table)$ (Fig. 9.5). The learned rules select the action $move(b, d)$, however, this must be delayed until the blocks under d are all well placed. Unfortunately, a negative example will be generated for $move(b, d)$ at the initial situation that avoids to learn one of the rules above⁵.

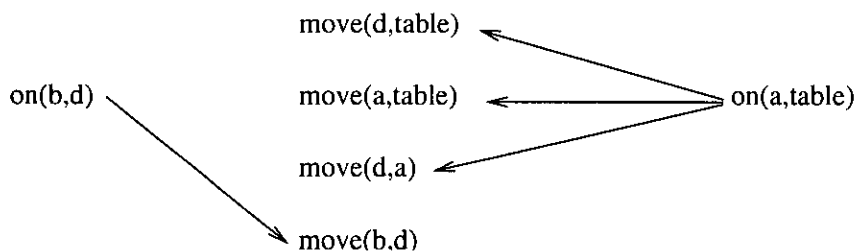


Figure 9.5: Pending subgoals and their associated actions

Furthermore, the exact precondition for the select rule cannot be expressed without specialized background predicates. When a plan for the scenario in Fig. 9.4 is used for training, a more specialized version of the second select rule is learned, that is not generally applicable. Note, however, that the third reject rule, that avoids that a block is moved onto a block that is not well placed, is still learned, which avoids b to be moved onto a not well-formed stack. Thus, the existence of explicit reject rules provides a third category of actions apart from select rules, i.e.,

⁵Note that the negative example will be also generated using the mutex heuristics, given that $move(b, d)$ and $move(d, table)$ are applicable and mutually exclusive.

non-rejected actions, that the planner can use to find a plan. However, this makes that some search is unavoidable apart from the control rules. As pointed by Geffner [76], a simple *policy* in which all blocks are put on the table and then stacked in order into their target positions would be more effective in these cases. Note that this strategy would be only learned if the training plans were generated according to it, however, the planner used for generating the training plans search for optimal plans and the mentioned strategy does not guarantee minimality.

Geffner [76] uses the enhanced expressivity of Description Logics in $(\forall on_g^*. (on_g = on_s))$ to express that “all blocks at the goal situation, are on the same locations than at the current situation”. However, the added expressivity of Description Logics makes that the space of concepts to be searched grows significantly with respect to logic programs, which is intrinsically large. Khardon [60] supplies the learner with support predicates like *inplacea/1* and *above/2* which allows to represent the concept *well_placed*.

$$G(on_table(x_1)), on_table(x_1) \rightarrow inplacea(x_1)$$

$$inplacea(x_2), G(on(x_1, x_2)), on(x_1, x_2) \rightarrow inplacea(x_1)$$

where G represents the goal situation. However, it is not obvious how this kind of control can be learned without extensive background knowledge or more expressive languages.

If we add the relation *above/2* to represent the transitive closure of the relation *on/2*, we still need universal quantification in the select rule’s body, because we need to refer to all the blocks under b . *Goal ordering* information [61] can be also used to find a plan. For instance, in Fig. 9.5, the subgoal $on(a, table)$ must be achieved before $on(b, d)$. This ordering could be used by the control rules so that some control rules are given preference over others. We will consider all these issues again in section 9.6.

9.4.4 The Logistics domain

Let us consider again the *Logistics* domain [130] which has been considered a benchmark problem in the planning literature. The problems in this domain typically start off with a collection of objects at various locations in various cities, and the goal is to redistribute these objects to their new locations⁶ (Fig. 9.6).

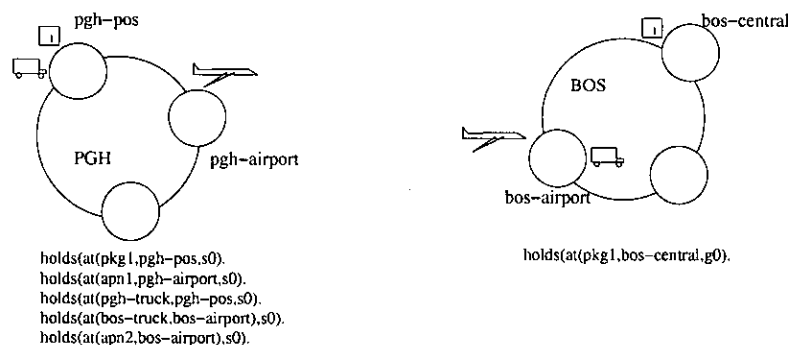


Figure 9.6: A possible plan for the logistics domain

The background is the same we considered in section 6.4.4. More elaborated predicates like *diffcity/2* and *samecity/2* which take two locations as arguments were included because they are

⁶Location of vehicles are not considered part of the goal.

used usually in the domain description and substitute a longer combination of *incity/2* and *diff/2* literals. These features [39] are useful for learning as they make clauses shorter and thus easier to learn⁷. All objects affected by actions should be included as arguments of these, –for instance, with action *fly(plane,from,to)* where the *from* argument is not strictly needed. By doing so, more relevant variables are introduced in the bottom clause without need for additional literals.

Logistics is a domain intrinsically parallel, where multiple vehicles can be moved concurrently. Indeed, the combinatorics of the domain arises from the problem of finding all ways to interleave the movements of packages and vehicles [51]. Concurrency is also possible if the scenario includes a single package that must be taken to a different city and the truck at the destination city is not at the airport. In this case, the action of driving the truck to the airport of the goal city can be interleaved with the subgoal of flying the package to that city. For concurrent plans, *minimality* of plans must be considered also in the number of actions executed at each situation, where only actions needed to achieve the goal must be executed to avoid false positives. Thus the planner must find a plan that minimizes the *parallel length*, i.e., the number of time steps in a plan, and then the *sequential length*, i.e., the number of actions, by removing unnecessary actions [37]. To minimize the sequential length, we use the cardinality constraints of *smodels* to limit the number of actions included in each stable model⁸, for instance:

```
{ happens(A,S): actsit(A,S) } slen.
```

where *slen* is a constant provided as an argument to *smodels*. If concurrent actions are not allowed in plans then, in those cases where actions can be executed concurrently, e.g., when multiple subgoals can be achieved independently, the *order independence* must be given to the learner by explicitly giving all possible plans, i.e., *interleaving* the subplans for several subgoals, otherwise false negatives might be produced.

When concurrent actions are allowed, the combinatorics is still present, and the number of plans possible can be very big, hence it is very costly to obtain optimal plans for learning. A policy to minimize the number of plans is to execute as many actions as possible in the first steps of the plan. Once the concurrency is fully exploited, the final part of the plan is sequential. Thus, the set of plans to be provided to the inductor is significantly smaller and still minimal. For instance, the following plan with parallel length 6 and sequential length 9, is optimal and all the actions are executed as soon as they are needed, thus no alternative plans need to be considered.

```
1 lot(pkg1,pgh_truck), lot(pkg2,la_truck)
2 drive(pgh_truck,pgh_po,pgh_airport), drive(la_truck,la_po,la_central)
3 unlot(pkg1,pgh_truck), unlot(pkg2,la_truck)
4 loa(pkg1,apn1)
5 fly(apn1,pgh_airport,la_airport)
6 unloa(pkg1,apn1)
```

This reduces significantly the data to be provided to the learner and provides with the same information as the whole set of plans. However, these operations can be very costly. Fortunately, the virtual-mutex condition avoids the false negatives produced if not all orderings are provided, given that actions that can be executed concurrently are clearly no mutually exclusive.

⁷It would be possible to analyze the domain description and build some predicates like this from conjunctions of literals that appear frequently.

⁸An auxiliary predicate *actsit/2* is used so that the scope of the constraint affects to every pair (action,situation).

We generated 28 plans of parallel length between 3 and 9 steps, –with 3 cities, 3 locations, 1 truck and 1 plane by city, and 3 packages– where the goals involve the location of at most 2 packages. From these plans, we obtained positive (select) and negative (reject) examples for each action (table 9.2).

Action	real	virtual	mutex virtual
lot/2	29	54	29
unlot/2	29	31	26
loa/2	21	26	9
unloa/2	21	21	21
drive/3	39	651	103
fly/3	21	420	90

Table 9.2: Positive and negative examples for the Logistics domain

Only 12 rules were learned that cover most of training examples and the rest were returned without generalization.

```

select(unlot(A,B),C,D) :- holds(at(A,E),+,D), holds(at(B,E),+,C).
select(unlot(A,B),C,D) :- holds(at(A,E),+,D), holds(at(B,F),+,C),
diffcity(E,F), airport(F).
select(lot(A,B),C,D) :- holds(at(A,E),+,D),
holds(at(B,F),+,C), samecity(F,E).
select(lot(A,B),C,D) :- holds(at(A,E),+,D),
holds(at(B,F),+,C), diffcity(E,F), not(airport(F)).
select(unloa(A,B),C,D) :- holds(at(A,E),+,D), holds(at(B,E),+,C).
select(unloa(A,B),C,D) :- holds(at(A,E),+,D), holds(at(B,F),+,C),
samecity(E,F).
select(loa(A,B),C,D) :- holds(at(A,E),+,D),
holds(at(B,F),+,C), diffcity(F,E).
select(fly(A,B,C),D,E) :- holds(in(F,A),+,D), holds(at(F,C),+,E).
select(fly(A,B,C),D,E) :- holds(in(F,A),+,D),
holds(at(F,G),+,E), samecity(C,G).
select(drive(A,B,C),D,E) :- holds(in(F,A),+,D), holds(at(F,C),+,E).
select(drive(A,B,C),D,E) :- holds(in(F,A),+,D), airport(C),
holds(at(F,G),+,E), diffcity(C,G).
select(drive(A,B,C),D,E) :- holds(at(F,C),+,D), holds(at(F,G),+,E),
samecity(C,G).

```

Their basic meaning is:

- Unload a package from a truck if the current location is the goal location or if the current location is an airport and the package in the truck must go to a different city.
- Unload a package from a plane if the current location is the goal location or if the package in the plane must go to a different location in the same city.
- Load a package into a truck if it must be in another location of the same city⁹.
- Load a package into a plane if the goal location is in a different city.

⁹Note that in the rule for *lot/2*, the literal *samecity(A,B)* also replaces $A \neq B$.

- Fly to the goal destination (an airport) of the package or to a city C if the package must be in a location of that city.
- Drive a package to its goal location or to an airport if the goal location is in a different city. Drive a truck to a location where there is a package that needs a pick-up.

In other words, the control to be added to the logistics domain consists of: if the object's new location is in the same city, it can be transported solely by truck, otherwise, if its new location is in a different city it has to be transported by truck to the city's airport, then by plane to the new city, and then by truck to its final location within the new city.

In some cases, rules for different actions have almost the same body. This happens when the actions form a sequence that is dependent on a common subgoal, so that the action selection rules for each one must somewhat replicate the need for the subgoal. For instance, when a package is in a wrong city we need to drive the package to an airport. In this case, we need, at least, the sequence of actions $\{lot(p, t), drive(p, l_p, a), unlot(p, t)\}$, where l_p is the current location of the package.

Comparing the rules learned with hand-coded ones included in the planning literature [2], some rules are missing or are not general enough. For instance, the rule learned for *drive/3* when a package needs a pick-up, covers only those cases where the package must be driven to a different location of the same city. The analysis of the plans used for training revealed that these situations were not very commonly represented. No similar rule was learned for flying planes to pick up packages, because cities are supposed to have one airplane. No rule was learned for moving vehicles to their goal location because the location of vehicles do not form part of the goal situation in the training plans.

With respect to *reject*, we obtained, among other, the following rules:

```
reject(unlot(A,B),C,D) :- holds(at(A,E),+,D),
                          holds(at(B,F),+,C), samecity(F,E).
reject(unloa(A,B),C,D) :- holds(at(A,E),+,D),
                          holds(at(B,F),+,C), diffcity(E,F).
reject(drive(A,B,C),D,E) :- holds(at(F,C),+,E), holds(in(F,A),-,D).
```

whose meaning is:

- Don't unload a package from a truck if it must be in another location of the same city.
- Don't unload a package from a plane if it must be in a different city.
- Don't drive a truck if there is a package that must be in the target location but it is not into the truck.

The number of rules is much higher than for the select rules. The reason is that reject rules correspond to the negation of the conditions in the select rules. In some cases, this makes that more reject rules are to be learned, hence the learned hypotheses might be less reliable and accurate. For instance, some reject rules, for which complementary select rules were learned, are missing, or are overspecific. Some of the conditions are difficult to express or even not possible. For instance, the condition that packages that are not referred to in the goal situation should not be loaded into vehicles, cannot be expressed easily. Similarly, a *reject* rule for avoiding movements of vehicles without a package, needs quantification over the packages to express that

there are no packages in a vehicle. We can also have the contrary situation, i.e., the select rules have complex conditions that are easier to learn when learning the opposite value. Without the mutex heuristics, rejected actions correspond to actions that interfere with the actions executed in the plan, whereas in other cases they correspond to actions not related to the goal. With the mutex heuristics, the number of these negative examples is much smaller because they are only generated when they interfere with the action executed in the plan. Actually, without the mutex heuristics we obtained much less rules than with the heuristics.

As pointed in [52], an *active learning* approach is applicable, so that it is possible to test the rules on additional plans and take those examples where some of the rules failed, to augment the training set and repeat the learning process. Some rules can be missing, overgeneral or overspecific. For instance, in the Logistics domain, one of the rules for *drive/3* is missing, however, the missing rule is learned after some new plans are added.

```
select(drive(A,B,C),D,E) :- holds(at(F,C),+,D), holds(at(F,G),+,E),
                             diffcity(C,G), not airport(C).
```

Thus, the current set of rules determine somewhat what instances were missing previously. By doing so, the control rules can be refined successively so that a robust planner is obtained. Plans for testing the rules are generated randomly, however, the refinement process could be made shorter if the additional plans are carefully selected. In section 9.6, we consider this issue again.

The form at which plans are generated for learning may produce different control rules in some cases. For instance, let us consider a package that is to be transported to another city but there is no a truck in the airport waiting for the package. Then, if the action of moving the truck in the goal city to the airport is performed as soon as possible or is delayed until it is the only eligible action, the positive example will be covered by different control rules, namely, “move the truck to a location in the same city where a package must be loaded” or “move the truck to the airport if there is a package in another city that must be in this city at the goal situation”.

On the other hand, some particular situations were not revealed by the rules mainly because in the plans used there is a single package, a single plane and a single truck by each city. Indeed, the learned rules produce optimal plans when a single package is to be transported or when there are no conflicting goals. For instance:

- When there are two packages at the same location of a city, that must be driven to different locations in the same city, a shorter plan is obtained if both packages are loaded and then transported, rather than loading and transporting each of them separately. Indeed, this is the plan produced by the learned rules.

However, if there are two trucks at that location, the control rules should load each package into a different truck so that the movement of trucks can be executed concurrently. A planner using the learned rules, might load both packages into the same truck.

- When there is a single truck in a city that contains a package that must be transported, and there is another package that needs a pick-up, the planner has two goals that are mutually exclusive, so it has to take one of them. Let us suppose the optimal plans require to pick up all the packages before. In this case, the control rules need to be applied in a particular ordering, otherwise the rule for *drive(t,_,l)*, when *l* is the target location of the package that is in the truck, must include as condition that there is no a package in the same city that needs a pick-up, however this cannot be expressed with the normal

quantification of logic programs. Note however, that the corresponding reject rule can be learned with the normal quantification, which avoids non-optimal actions to be executed, however this requires some blind search to find the intended plan.

- In all the plans of the Logistics domain that we have considered, there is a single truck by each city. Let us consider a situation where a package needs to be loaded into a truck and several trucks are at the same location. In this case, the package needs to be loaded into *any* of them. However, one of the control rules learned expresses that a truck must not be moved while there is a package that needs to be driven to another location of the same city, and other that the package must be loaded into all of them. Thus, when the rules are applied, they select to load the package into every possible truck, so that the planner will take just one of them. This is so because both the package and the truck are arguments of the actions that appear in the head of clauses, hence they are universally quantified. As a first consequence, some rules for the action *lot/2* will not be learned if all but the truck actually used in the plan, were considered negative examples. Unfortunately, these negative examples are generated even with the mutex heuristics, because a package cannot be loaded into two trucks at the same time. What we actually want to express is that *at least one* of the trucks must wait for the package and that the package must be loaded into one of the trucks. The problem can be avoided if all the alternative plans are provided, i.e., the package is loaded into a different truck in each plan. This seems the most natural solution, however it relies on the completeness of the training plans. Note that, the learned rules will still recommend actions that load the truck into every possible truck.

A second consequence is that non-optimal plans may be produced. For instance, let us consider that two packages at a same location need to be transported. A package is loaded into a truck, and simultaneously another truck has come to this location. As the learned rules avoid trucks to leave while there are packages that need to be loaded, then only when both packages have been loaded, the trucks can leave the location resulting in a non-optimal plan.

These situations are the result of the existence of interactions between potential goals when multiple packages are to be transported or when multiple vehicles can be used. The learned rules still produce a plan for these cases, but not necessarily optimal. In some cases, the detailed control would require highly complex rules, which are hard to be learned¹⁰, whereas in other cases, we need to give preference to some immediate subgoals over others by using preferences explicitly or by using more expressive languages. We will consider these issues again in section 9.6.

9.5 Control rules based on subgoals

In this section, we enhance the form of the action-selection rules to incorporate a restricted form of *temporal logic*. We will use as our language for expressing search control knowledge [2] a first-order version of linear temporal logic (LTL). The syntax is obtained from first order logic by adding temporal operators such as *Until* (\mathcal{U}), *Next* (\bigcirc) and some of the derived operators, such as *Always* (\square) and *Sometimes* (\diamond). It is shown that these temporal constructs are sufficient to represent a large class of temporal information [23].

The form of the search control rules that has been considered so far suggest the action to be executed/rejected. In the planning literature [113, 2], however, most of search control rules

¹⁰This complex control is not included in the control rules found in the literature.

are expressed in a form of temporal logic (with operators \square , \bigcirc and *Until*) where the head is a condition on some fluent, rather than a term of the form *select(a)* or *reject(a)*. In this section, we will analyze the effect on learnability of changing the form of the search control rules with respect to *select/reject* rules. These operators can be represented in the Situation Calculus through new literals (table 9.3)

always($e, v, p(s_0, s_g), s$)	\square
next($e, v, p(s_0, s_g), s$)	\bigcirc
eventually($e, v, p(s_0, s_g), s$)	\diamond
until($e, v, e', v', p(s_0, s_g), s$)	\mathcal{U}

Table 9.3: Temporal operators

where e represents a fluent, v a truth value and $p(s_0, s_g)$ a plan. These literals will become now the head of the search control rules.

Most of search control rules in the planning literature are expressed with the operator \bigcirc (*next*). Indeed, as pointed by Rintanen, all the Logistics control rules can be formalized with the operator \bigcirc . We consider two different semantics for \bigcirc . In the first one $\bigcirc\phi$ is true at situation i if ϕ is false at i and true at $i + 1$. By doing so, the operator $\bigcirc\phi$ actually replaces the *select* rules for all the actions that might eventually produce ϕ at the next situation. The effect of this kind of rules with respect to the *select/reject* rules is then similar to the *ramification problem* in commonsense reasoning. In terms of compression, such rules will be supported by more examples that previously were *shared* by the corresponding rules for each individual action.

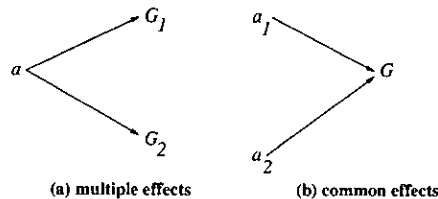


Figure 9.7: Subgoals for actions

Unlike this, when an action produces multiple effects, we just need a *select* rule whereas we need one rule for each simultaneous effect. However, by applying a classification method, the *select/reject* example will be covered just once, and the rule will correspond to just one of the reasons to execute it. The order at which hypotheses are built determines which of the rules will be learned. Let us consider a *select* rule for *move(b, table)*. The need for the action is that the table is the goal location of b , and that the previous location of b (e.g., c) must be clear in the goal situation. We need two selection rules for *move/2*, one for each subgoal, but the example will be assigned to just one of them, thus the example will represent just one of the reasons. However, if we learn rules for the subgoals $\bigcirc\text{clear}(c)$ and $\bigcirc\text{on}(b, \text{table})$, both reasons will be supported. because we have an example for each subgoal.

When ϕ is true in both the current and the next situation, the intended meaning is to preserve ϕ . For instance, in the blocks world, once a block is in the position it must be in the goal state, it should not be moved. In the logistics domain, for instance, as packages can be

moved through different actions, a single rule avoids the current location of the package to be changed, i.e., substitute multiple *reject* rules, one for each action that might change the location of the package.

Let us consider again the Logistics domain. The new literals that represent the temporal operators are built from the training plans by taking every pair of consecutive situations. The heuristics for the extraction of examples is the same when dealing with subgoals instead of actions. New *bias/2* statements are added to the BK.

```
bias(next(in(+,+),#,+),[holds(in(+,+),#,+), ...]).
bias(next(at(+,+),#,+),[holds(in(+,+),#,+), ...]).
```

We obtained 17 rules that cover all situations explained by the action-selection rules, among others:

```
next(in(A,B),+,C,D) :- truck(B), holds(at(A,E),+,D),
                        holds(at(B,F),+,C), samecity(F,E).
next(at(A,B),+,C,D) :- holds(at(A,B),+,D).
next(in(A,B),-,C,D) :- holds(at(A,E),+,D), holds(at(B,E),+,C).
next(at(A,B),+,C,D) :- holds(in(E,A),+,C), holds(at(E,B),+,D).
```

More rules are learned with respect to the *select* rules, because all the actions produce at least two effects. In some cases, the subgoal rules achieve more compression, e.g., those that generalize over the vehicles, so that multiple select rules would have to be learned. For instance, the second rule is applicable both to planes and trucks, so that when the location of the vehicle is the goal location of the package, then the next effect will be to unload the package from the vehicle, whether a truck or a plane. Similarly for the last rule, that is used to transport a package to its goal location. In other cases, some conditions in the body are specific for the truck (resp. the plane) an independent rule will have to be learned. For instance, the first rule is equivalent to the select rule for *lot/2*. As a counterpart, the use of subgoals makes rules more difficult to read sometimes, specially for *at/2* that is applicable to both packages and vehicles. Note also that the second rule for *in/2* is a consequence of the rule for *at/2* because of the invariant $at(Pkg, Loc) \Rightarrow \neg in(Pkg, V)$. This can be applied to many other cases. For instance, the action *drive(pkg, from, to)* produces the effects $at(pkg, to)$ and $\neg at(pkg, from)$, thus the corresponding *next* rules are almost identical.

With respect to the second meaning of the operator \bigcirc , we obtained similar results. However, in general, subgoal-selection rules favor the compression measure of learning algorithms and coincide with the control rules used in the planning literature. With respect to the operator \square (*always*), we have that $\square\phi$ is true at situation i if ϕ is true at j for all $j \in \{i, \dots, n-1\}$ where the goal holds at n . With respect to static select/reject rules, $\square\phi$ produces the same effect but starting from any situation in the plan and not from the initial situation. With respect to $\bigcirc\phi$, since the scope of the latter is only the next situation, it needs that once the condition C for the rule becomes true, C is true henceforth. For instance, a rule with the meaning “if a block is well-placed, do not execute actions that move it”. In this case, the operator \square can be replaced by \bigcirc without loss of meaning¹¹. However, if the condition of the rule can change but the rule must be still respected, then the rules are different and a form of memory should be added.

¹¹As to the planner’s performance, \bigcirc might provide a more efficient checking, because the control rule needs to be executed only once –e.g., removing the action from the set of possible actions– while the scope for the rule of \bigcirc will be executed at each state. However, this is very dependent on the planner used.

The operator *Until* is a generalization of \square where $\square\phi \equiv (\phi \mathcal{U} \text{goal})$. The use of the *Until* operator is illustrated with rules like *if you achieve C, then preserve it until C' is achieved*. Our semantics for $\phi\mathcal{U}\phi'$ is only one of several possible ones, where ϕ' is required to eventually become true. Formally we have that $\phi\mathcal{U}\phi'$ is true at situation i if ϕ' is false at i , ϕ' is true at some $j \in \{i, \dots, n\}$ and ϕ is true (resp. ϕ' is false) at k for all $k \in \{i, \dots, j-1\}$. For instance, in [113], *Until* is used in the Logistics domain to express: "a truck must stay at its current location until a package that needs to be moved, is loaded into it". However, a similar rule can be obtained with the operator \bigcirc .

Finally, the operator $\diamond\phi$ (*eventually*) does not add apparently too much expressivity to control rules because it is too imprecise to assert "an effect ϕ should be achieved in the future". Although operator \diamond is not commonly used in the literature for expressing search control rules, it can assume the role of identifying *future subgoals* [84] based on the current situation and the goal situation. For instance, in the Blocks world, we have that:

$$\text{on}(a, b), \text{on}_g(c, b), a \neq c \rightarrow \diamond\text{clear}(b)$$

where on_g refers to the predicate *on/2* in the goal situation. The rule above identifies a future subgoal to be achieved, that may need several situations to be achieved. In the logistics domain, a subgoal to achieve when a package is in a wrong city is to drive the package to an airport.

$$\text{in_wrong_city}(p), \text{airport}(a) \rightarrow \diamond\text{at}(p, a)$$

In this case, the same control can be also expressed with the basic action-selection rules for the sequence $\{\text{lot}(p, t), \text{drive}(p, l, a), \text{unlot}(p, t)\}$. However, subgoal identification can be interesting when the connection between the current and the goal situation is not obvious, but it is with respect to the subgoal. The combined use of these identified subgoals together with the action-selection rules may increase the quality of the control, so that a control rule can refer to future subgoals and not only to the final goal situation.

Let us consider Fig. 9.8. In this case, the only action to select is *move(c, table)*, however this is not directly related to the final goal *clear(a)* unless predicate *above/2* is considered. Any attempt to learn this control directly will be dependent on the number of blocks piled, as happened in Fig. 9.5. The final goal *clear(a)* needs previously *clear(b)*, but unlike Fig. 9.5, *clear(b)* is not included in the goal situation. Without a predicate *above/2*, it is not obvious how to express that all blocks above a must be removed. However, we can use control rules to find a necessary subgoal of *clear(a)*:

$$\text{on}(b, a), \text{clear}_g(a) \rightarrow \diamond\text{clear}(b)$$

because *clear(b)* has a clear relation to the final goal unlike *move(c, table)*. Given this new subgoal, we can now learn that:

$$\text{on}(c, b), \text{clear}_{sg}(b) \rightarrow \bigcirc\text{clear}(b)$$

If there is a block d over c , we need to identify previously *clear(c)* as a subgoal of *clear(b)*:

$$\text{on}(c, b), \text{clear}_{sg}(b) \rightarrow \diamond\text{clear}(c)$$

where the scope of the operator \diamond is delimited by *clear_{sg}(b)*. Note that the above rule can be applied iteratively over each new subgoal, so that we get a sequence of subgoals where the last one is the first to be achieved, thus the pile of blocks is unstacked in the correct ordering. However, the practical application of this approach will require further study.

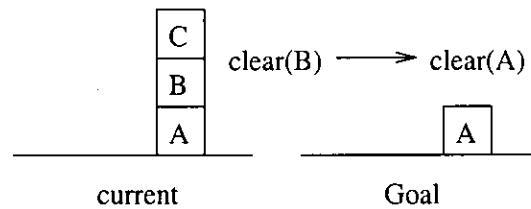


Figure 9.8: Subgoal identification

9.6 Discussion

In this section, we review some issues that must be considered to improve the quality of the learned control.

Concurrent actions

Kharon [60] deals with deterministic control rules where a unique action is selected at each situation. We deal with non-deterministic control, where possibly several actions are eligible at one situation. In many cases, these actions can be executed concurrently to achieve independent subgoals, e.g., in the Logistics domain. However, there are no so-called cancelling actions or actions that must be executed concurrently. According to the action-selection rules, actions that must be executed concurrently are not distinguished from multiple eligible independent actions. Thus, when multiple select rules are fired at a situation, the planner could execute them all concurrently, or just a subset. Unfortunately, some actions might be cancelling for others or even contradictory. For instance, actions a and a' can be eligible and however a' cancels a . Thus, *select/reject* rules are needed that work over non-atomic actions, i.e., selecting a set of actions that *must* be executed concurrently. Cancelling actions could be obtained from the domain description and added explicitly to the control rules. The set of possible compound actions, i.e., actions that must be executed concurrently, could be also extracted from the domain description and the effects produced in the plan.

Limited expressivity of Horn programs

Most ILP methods deal with a particular subset of first order logic, where variables in the head are universally quantified and variables in the body (but not in the head) are existentially quantified. In most applications, these logic programs are enough, whereas in others the background is carefully selected before learning (even building new predicates) to enhance expressivity.

The first limitation is that the arguments of the actions/fluents that appear in the head of clauses, are universally quantified. Sometimes, it would be interesting that some arguments of actions are existentially quantified, for instance, when a package can be loaded into several trucks. Kharon [60] uses an special interpretation of *decision lists*, where only an instantiation of the rules (in lexicographical order) is considered. If only a single action can be selected at each situation, this approach works well, because it takes one of the possible eligible actions. This type of control could be incorporated to the answer set planner in the form of *constraint rules*:

```
{select(lot(P,T),S):truck(T)} 1 :- package(P),situation(S),...
```

that forces at most one instantiation of the action *lot/2* to be executed in the current situation. For the intended interpretation, we can also *invent* a new action predicate *lot(p)* to mean that

a package p must be loaded into any truck. However, it would be possible to have other interpretations, i.e., all packages must be loaded into a single truck, which would require to try all quantifiers for the action arguments.

In other cases, *universal quantification* might be needed in the body of clauses, as we saw in the blocks world and in the Logistics domain. For instance, the rule for $drive(t, _, l)$ in the Logistics domain needs to establish a preference between driving a package to the goal location or driving to a location where a package needs a pick-up. When non-ordered rules are to be learned, the select rules need the explicit use of quantifiers to express the same meaning. Decision lists used by Khardon [60] can account for this kind of control.

```
select drive(...) if any package needs a pick-up
else select drive(...) if the next location is the goal location of a package
```

Bacchus and Kabanza [2] also use an ordered set of control rules. We could similarly impose an order for the application of the control rules, so that these are to be learned in a particular ordering. However, by doing so, the control becomes strictly *sequential* because only one rule is applied at each situation. Furthermore, the fact that control rules are to be applied in a particular ordering, makes that rules cannot be learned independently and makes them lose part of the declarative meaning.

The use of integrity constraints as a means of specialization [28] can also replace (partially) the explicit use of quantifiers. For instance, the overgeneral rule “drive the truck to the target location of the package contained in it” is specialized by the integrity constraint “there cannot be a package in the same city that needs a pick-up”. This requires to use reject rules to specialize overgeneral select rules. Note that the select and the reject rule learned separately do not achieve the same effect.

Similarly, in the blocks world (with a predicate $above/2$), the following rules capture the concept that the target block B is well-placed:

```
select(move(A,B),C,D) :- holds(on(A,B),+,D).

:- select(move(A,B),C,D),holds(above(B,E),+,C),
   holds(on(E,F),+,D),holds(on(E,F),-,C).
```

where the constraint avoids that A is moved onto B when a block under it is not currently well-placed. However, these approaches cannot be considered a general solution because variables can be quantified in many different forms. Thus, in general, more expressive languages are needed.

The need for quantifiers is a consequence of the absence of specialized background knowledge like $well_placed$ in the blocks world. In the planning literature, the use of temporal operators is enhanced with the inclusion of these *additional predicates*, e.g., $wrong_city$, $well_placed$ and so on, so that the complexity of the temporal formulas is kept small. However, as pointed by Kautz and Selman [52] an essential part –and the most difficult– of the learning process is to discover these predicates. From the point of view of learning, where we do not assume the presence of such background predicates, it is important to consider additional expressivity in the form of the search control rules. Geffner and Martin uses *Description Logics* [76, 3], however, the added expressivity makes that the space of concepts to be searched grows significantly with respect to normal logic programs, which is intrinsically large. A promising idea is the use of a constructive induction [39] approach that builds a set of new features both from data or from the learned rules. In fact, from the learned rules for the Logistics domain, we can obtain the

definition for specialized predicates used in the literature like *in_wrong_city*, *in_wrong_location*, *need_to_move_by_truck* and so on, that could be used as additional predicates for learning.

Active learning

Active learning seems particularly relevant for learning control rules. Unlike other learning applications, the whole space of instances is available, at least in theory, and we can easily generate new instances of a planning problem. Thus, it is possible to test the rules on additional plans and take those examples where some of the rules failed to augment the training set and repeat the learning process. Unfortunately, there is no an oracle that will provide us with the best counterexample or the most typical example, hence the additional plans are generated randomly.

Work in active learning has shown that just allowing the system to pick which example to label next can greatly reduce the number of examples the user needs consider. A set of well-chosen plans can reduce the refinement phase significantly by, e.g., identifying unexplored regions of the space of possible plans, or providing good counterexamples. A possible approach consists of analyzing the learned rules and altering conditions that appear in the learned rules. For instance, the rule learned initially for *drive/3* when no package is in the truck but there is a package that needs a pick-up, was not general enough. By negating one of the conditions (e.g., *difficulty*) in the rule, we obtain a new planning problem that will produce a false negative. We can use more general or more specific conditions on domain types, for instance, an airport instead of a location, and so on. Another possible approach that does not consider the learned rules, is to discover associations or patterns in the training plans, that help identifying plans that were not considered for training. For instance, the following association says that there is always a truck at the initial location of a package that needs to be moved.

$$\leftarrow at(pkg, loc), goal(at(pkg, loc')), loc \neq loc', (\exists truck : at(truck, loc))$$

Similar associations can be obtained, for instance, “there is always a truck at the airport of the goal city”, “packages to be moved are never at the same city in the initial situation”, and so on¹².

Further topics

The kind of search control rules we have explored shows the interestingness of declarative learning methods for planning. However, the approach relies upon the fact that a connection can be found between the current state and the goal state. For instance, in the 8-puzzle, search-control rules recommend one of actions {north,south,west,east}, however, only some movements actually contribute directly to the objective, while many others are necessary but apparently disconnected from the goal. When such a connection cannot be found, no control rules will be learned or the rules will tend to overfit the training data. The discovery and use of intermediate subgoals can be a promising approach, as we showed in the Blocks world.

In some domains, the first actions in any plan are taken almost randomly, at least the first action, e.g., in the 8-queens domain, thus introducing noise in the training set. In this domain, the goal condition is not provided as ground facts, because the plan is just to put the queens in the locations referred to in the goal situation. In these domains, any control rule cannot refer to

¹²Some other associations of no interest are also found, for instance, “trucks are never in the goal situation”, etc..

the goal situation. However, it is still possible to learn control rules based only on the current situation.

In other cases, only weak control rules can be found, i.e., control rules that work in most cases, but they are prone to fail in particular cases. However, this type of control would require the planner to be able to recover from situations where the control rules have failed, and apply full search locally.

9.7 Conclusions

Several authors have shown that it is possible to learn declarative control rules using inductive learning methods. Our work is mostly based on those of Kautz and Selman, and Geffner and Martin, that unlike Khardon's approach, do not use specialized background knowledge and do not need large training sets.

Kautz and Selman report positive results in the Logistics domain, among others, using action-selection rules with the mutex heuristics and the FOIL system. For instance, the following rule captures the concept of "an object that is not in its goal city".

$$\neg \text{Unload-Airplane}(\text{o p a}) \leftarrow \\ (\text{in-city a c}) \wedge (\text{goal}(\text{at o l})) \wedge \neg(\text{in-city l c})$$

Geffner and Martin focus on the Blocks world where they show the benefits of using a concept language and take full advantage of the enhanced expressivity. For instance, the following rule says to pick up a block if it is clear and its target block is clear and well-placed.

$$\text{PICK}((\forall \text{on}_g^*. (\text{on}_g = \text{on}_s)) \wedge (\forall \text{on}_g. \text{clear}_s) \wedge \text{clear}_s)$$

However, the added expressivity makes that the space of concepts to be searched grows significantly with respect to normal logic programs, which is intrinsically large. In the Logistics domain, the space of concepts may become prohibitive. On the other hand, the control learned needs to be extensively rewritten to be used by most planners. Other restrictions due to the language, as the restriction to unary actions is also an obstacle for it to be applied to other planning domains. In both works, the learned rules are refined selectively until a robust set of rules is obtained.

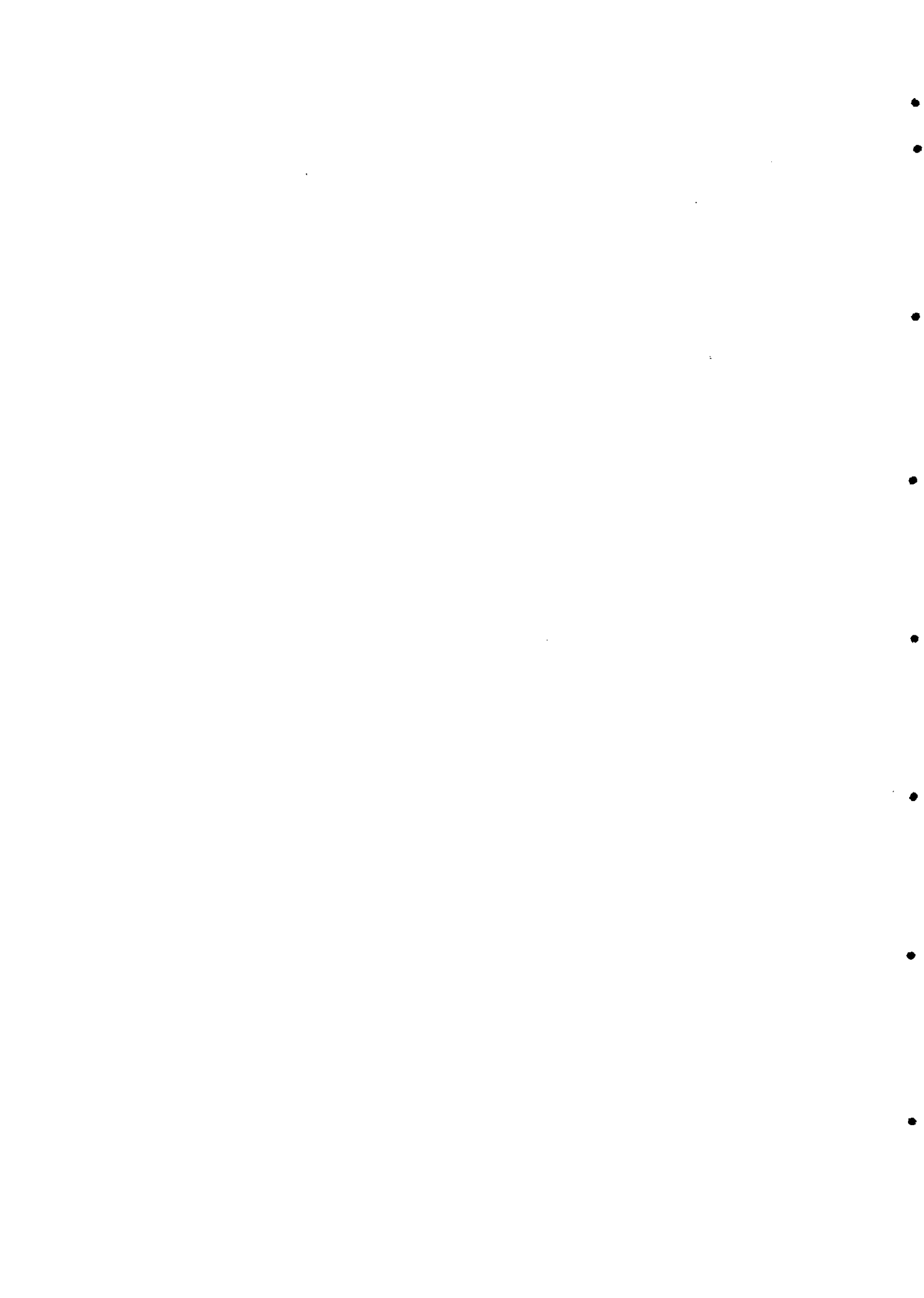
We have shown how the task of learning declarative control rules for planning can be reformulated into our framework for learning action theories in a natural way. We believe that Action Languages are better framed for this task because they provide with a natural and homogeneous representation, as well as a solution to the Frame problem in the observations. Furthermore, it is not obvious how to consider multiple plans for learning, in the commented works. We also extended the form of the action-selection rules to consider a reduced form of temporal logic, where the control rules select the next subgoal to be achieved. These rules are closer to the control rules used in the literature and make that less control rules are to be learned that are more generally applicable than the action-selection rules. We identified the importance of the use of select and reject rules, and its combined use to enhance expressiveness.

We made an exhaustive analysis of the Blocks world and the Logistics domain. We were able to learn the control rules for the Logistics domain used in the literature, from a small set of examples, and identified some limitations as to expressivity and as to the optimality of the learned planner. Similar results are obtained in other benchmark domains used in planning competitions,

like the tireworld or the grid domains. Our current work includes a more careful and detailed empirical evaluation of the approach, and the automatic generation of good counterexamples from the learned rules to speed-up the refinement process. This task raises specific challenges for current methods in ILP, mainly with respect to the limited expressivity of Horn clauses, when no specialized background knowledge is assumed to be provided. The use of more expressive languages [3] or the use of constructive induction methods [39] seem particularly relevant for this task.

Part IV

Related work and Conclusions



Chapter 10

Related work

In this chapter, we will cover the most relevant work in the field of *Dynamic system identification* [13], directly related to the objectives of this thesis. Dynamic system identification has been studied in a variety of disciplines including control theory, neural networks, and automata theory. The inferred model might correspond to a system of differential equations [129], a set of production rules [124], or a set of states and transition probabilities [13].

10.1 Learning operators effects for planning

Many approaches exist to learning domain knowledge for planning [124, 87, 22, 103, 132, 99, 27, 7]. They learn or refine operator effects through experimentation or from traces from experts' problem solving. Most of these approaches fall somewhere between those aimed at building a complete world model and those aimed at learning a complete *behavioral policy*. Some of them require that the agent has initial approximate planning operators or just a weak initial domain model [132]. In many of them, most emphasis is put on the autonomy of learning, where planning, learning and exploratory steps are interleaved [15, 124, 103, 132, 22], whereas others combine a phase of exploration, e.g., with a *wander* program that records observations [7, 99], and a phase of generalization carried out by inductive learning systems. The difference is that the latter approaches consider the learning process and the exploratory process as two separate processes, so that the learning algorithm has absolutely no control over the sample of labeled examples drawn, i.e., they do not perform actions to control the system, and so learning is not really active. Our approach falls into this second approach, i.e., to learn preconditions for the actions and their effects from a given set of observations, rather than with learning an environment by acting on it. The methods of representation used include mostly STRIPS-based operators [124, 22, 132], but also teleo-operators (TOP) [15], *multitokens* [99], regression trees [7] (decision trees with continuous variables) and so on.

LIVE [124] uses a dialect of first-order logic, where the prediction rules are very much like STRIPS operators [36]. An special scenario of the Towers of Hanoi is presented that consists of three disks and three plates. The disks have different sizes and they can be moved from one plate to another according to some rules. An example of prediction rule may look like the following:

Condition	$\text{INHAND}(\text{diskx}) \wedge \neg \text{ON}(\text{diskx plate/table})$
Action	$\text{PUT}(\text{diskx plate/table})$
Prediction	$\text{ON}(\text{diskx plate/table}) \wedge \neg \text{INHAND}(\text{diskx})$

It says that the action PUT can put a disk in hand onto a plate or the table if the disk is not already there. When no relations are changed, LIVE will still create a new rule where its condition and prediction will be the same and it says that if disk x is in hand and there is a disk y on plate p , the action will have no effect on disk x , that is, disk x will be still in hand.

```

Condition  INHAND(x)∧ON(y p)
Action    PUT(x p)
Prediction INHAND(x)

```

Gil [22] presents an approach to refining a planner's incomplete knowledge base. Given a knowledge base consisting of STRIPS-style operators –with a list of state features (or postconditions) that are added and another list of features that are removed by applying the operator– her approach uses exploration and experimentation, so that an inductive learner is trained on the instances of successful and unsuccessful operators and the states where the operators were chosen to fill in missing preconditions and effects. The example domain consists of crafting a primary telescope mirror from raw materials, where operators are in the form:

```

(GRIND-CONCAVE
  (params (<obj>))
  (preconds
    (is-solid <obj>))
  (effects (
    (add (is-parabolic <obj>))
    (del (is-planar <obj>))
    (del (is-reflective <obj>))
    (del (is-polished <obj>))))))

(CLEAN
  (params (<obj>))
  (preconds
    (is-solid <obj>))
  (effects (
    (add (is-clean <obj>))))))

(POLISH
  (params (<obj>))
  (preconds
    (and (is-clean <obj>)
          (is-glass <obj>))
    (~ (is-reflective <obj>))))
  (effects (
    (add (is-polished <obj>))))))

(ALUMINIZE
  (params (<obj>))
  (preconds
    (and (is-clean <obj>)
          (is-solid <obj>)))
  (effects (
    (add (is-reflective <obj>))
    (del (is-clean <obj>))))))

```

Wang's OBSERVER system [132] is able to work from an empty knowledge base, using traces from experts' problem solving to fill in the empty operator descriptions. The learned operator GOTO-DR in a small robotics domain, when the observation in Figure 10.2 is given to the learning module, is:

```

(operator goto-dr
  (preconds ((<v1> door) (<v2> object) (<v3> room) (<v4> room))
    (and (inroom robot <v4>)
          (connects <v1> <v3> <v4>))
    (connects <v1> <v4> <v3>))
  (dr-to-rm <v1> <v3>))
  (dr-to-rm <v1> <v4>))
  (unlocked <v1>))
  (arm-empty)
  (inroom <v2> <v3>))
  (pushable <v2>)))
  (effects ((<v5> object))
    ((add (next-to robot <v1>))
      (del (next-to robot <v5>))))))

```

One of the limitations of these approaches is that they assume pre-conditions are conjunctive and they do not handle domain noise. TRAIL [15] extends this work by allowing disjunctions in the preconditions, noise in an action's effects, and also durative actions. TRAIL uses so-called teleo-operators (TOP) [15] that are an action representation different from STRIPS operators, which are appropriate to reactive agents. The TOP below describes the process of moving a robot forward to the proper distance from which a construction bar can be successfully grabbed. The TOP also records the probability assigned to each side effect. TRAIL learns the probability that an outcome of an action will occur but it does not allow the probability of the outcome be dependent on the preconditions.

```

Action:      move-forward
Postcondition: at-grabbing-distance(?x)
Precondition: facing-bar(?x) OR too-near-to(?x)
Side Effects: NOT(too-near-to(?x))    100%
              NOT(too-far-from(?x))  100%
              NOT(facing-bar(?x))    10%

```

desJardin's PAGODA system [27] is also able to learn relationships between values of the preconditions and the probability that the action will succeed. However, like TRAIL, it only learns whether or not the action will be successful.

More recently, Balac's ERA system [7, 6] assumes that the agent first explores its domain by taking random actions and recording state descriptions, and includes the ability to model noise and the use of continuous variables in the action descriptions. For instance, the tree at Fig. 10.1 identifies the environmental conditions that influenced a robot's ability to turn. The tree includes nodes for each combination of roughness and rain that the robot encountered. One of the main limitations is however that ERA is still restricted to predicting a single outcome or effect of an action.

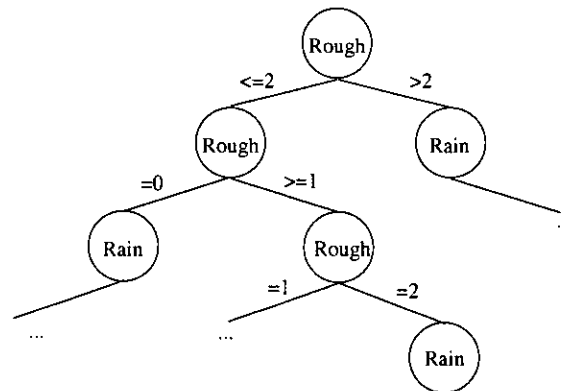


Figure 10.1: A Regression tree for a robot's domain

Oates [99] learn planning operators only from the agent's own past interactions with its environment, using an special representation method (multitokens). He deals with noise introduced by exogenous events and allows uncertainties associated with the outcomes of actions. For instance, the multitoken below describes that a robot can pick up a block if it has not a dry gripper and it is not holding another block.

```
(pickup * * NOT-GD NOT-HB) ⇒ (* * * * HB)
```


IMPROV [103] learns procedural planning knowledge and supports complex non-instantaneous actions (actions with sequential effects). For instance, the following code represents the complex effects of braking a car.

```

                (a) Conditional Effects
IF   current-operator(brake)
and  weather(raining)
and  isa(<c>,my-car)
and  tires(<c>,worn)
THEN add skid(<c>)

                (b) Sequential Effects
IF   current-operator(brake)
and  brake-pressure(<c>,0)
and  isa(<c>,my-car)
THEN add  brake-pressure(<c>,10)
      remove brake-pressure(<c>,0)

IF   current-operator(brake)
and  brake-pressure(<c>,10)
and  isa(<c>,my-car)
THEN add  brake-pressure(<c>,30)
      remove brake-pressure(<c>,10)

                (c) Iterative Effects
IF   current-operator(brake)
and  isa(<c>,my-car)
and  speed(<c>,<speed>)
and  greater-than(<speed>,0)
and  time(<t>)
THEN add  speed(<c>,<speed> - 5)
      add  time(<t> + 1)
      remove speed(<c>,<speed>)
      remove time(<t>)

```

EXPO, LIVE and OBSERVER share a similar STRIPS-like representation, thus, these systems suffer from the limitations of STRIP-based formalisms. IMPROV and TRAIL allows complex operator effects, e.g., they are able to cope with sequential effects, durative actions, or exogenous effects, however, they rely on procedural representations. This lack of declarativeness makes the results of learning very dependent on the particular formalisms and difficult to transfer. Furthermore, despite teleo-operators [15] or regression trees [7] are more expressive to representing action domains than traditional STRIPS operators, it is still not obvious how to represent indirect effects, recursive effects, cycles or exceptions.

With respect to the observations, in most approaches each example in, e.g., the blocks world, describes the properties of each block before and after the operation, together with the subject of the operation (i.e., the block to be moved) and its destination, so that each example consists of a separate dataset. As a simple illustrative example, consider the two states before and after an agent applied an operator as shown in Figure 10.2 where the delta-state is the difference between the pre- and post-operator execution states [132].

Thus, positive examples must be extracted from narratives so that the information provided by the narrative, i.e., the sequence of actions, is lost. As a consequence, they need the explicit representation of unchanged properties from one particular situation to another (*frame problem*). In the Situation Calculus, positive examples are the individual effects of actions, so that learning is done directly from the narratives, hence there is no need for preprocessing methods, and it allows inference on narratives to, for instance, implement inertia, complete missing values, or learn recursive effects.

On the other hand, some of the previous approaches use partially an ILP algorithm to generalize action preconditions, however in most cases they use a different representation method that needs a translation procedure into a form that ILP algorithms can understand. For instance,

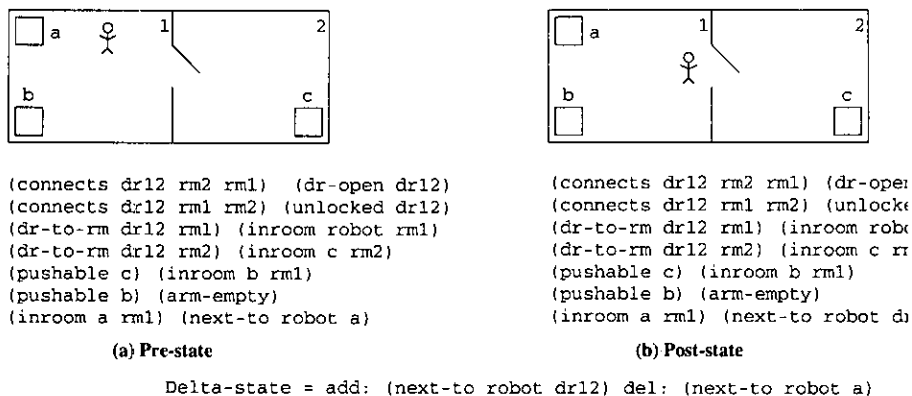


Figure 10.2: An observation of the state before and after the application of an operator that moves the robot to the door.

in TRAIL, as instances are not directly in the format of positive and negative examples to be covered, a special *PreImage* predicate is used that is labeled as either positive or negative for each state depending on whether the corresponding instance was a success or a failure, with an additional state argument to allow for generalization over situations. TOP preimages have to be converted into a form that the ILP method Foil can use and once a concept has been learned this is converted again into a TOP preimage. In other cases, the language used has not an easy translation into first-order logic. Unlike this, the use of logic programming theories of actions, makes that an action theory is learned in the same way as it will be used, thus there is no a different representation for learning and another different for reasoning.

10.2 Learning action models using ILP methods

In the ILP field, there are some approaches to learning dynamic systems [19, 53, 35, 88]. In the approach of Feng in [35], examples are introduced for each situation and the sequence ordering is implicitly represented by the *situation* argument of the predicates in the example. The predicate *succ(S,S)* is also defined to relate two consecutive situations. In the application domain, components of a power subsystem are equipped with sensors, and the sensor readings are transmitted at regular intervals to the ground, for instance:

```
sensor1(s2,+).
```

The simulator starts with an initial state of the system and a component which is assumed to be faulty. It then moves to another state which is a legal transition from the previous state. After each transition, the states of components are evaluated and then propagated until the system reaches a stable state, i.e., there is no change in the states of components through several propagations. In each transition the states of the sensors are recorded. A typical rule that is learned is:

```
fault(timeA):- succ(timeB,timeA), fault(timeB).
```

Although this rule is interesting because it describes the situation that a fault lasts over time, it is operationally useless and it cannot help to detect faults from sensor readings. In general, approaches that use the first-order features to merely represent the sequence in the set

of examples have restricted ability to model dynamic domains because several strong restrictions must be made about the behavior of the system. For instance, when literals of past states (other than previous) become relevant, these are only introduced in a clause by means of chains of *succ/2* predicate, thus the temporal interval between any consecutive situations is fixed. This restriction does not make sense in most dynamic systems.

Bratko [19] describes an application of inducing qualitative models [62] of dynamic systems from example behaviors using ILP methods. In [53] Sammut and Hume have studied a reactive approach to learning of action sequences in a simulated robot world, that produces a theory to recognize sequences of actions by describing the pre-conditions and post-conditions of each action in the construction sequence.

However, the most relevant work in the ILP field is [88] that shows an approach to learn the effects of actions using a logic programming implementation of the *Event Calculus* from observed time traces of property values from an existing dynamic system. Background knowledge includes *domain core axioms* (DSA) for Event Calculus, while domain specific axioms are learned from a narrative of events. The DSA's determine which events trigger the state of particular fluents. The initial state is given by facts like `holds_at(on(block_b,block_a),+,0)`, whereas a narrative includes separately the actions `happens(move(block_a,block_b),5)` and the effects. They reify the truth value of fluents and unify the initiation and termination of properties into a single predicate definition *flips/3* to state that an event flips the value of a fluent, thus allowing to use a single predicate for every learning task. Since the observations of an event calculus program are recorded as different predicates from those that can be used to define legitimate DSAs, a theory revision mechanism provided by logical back-propagation is applied, where examples of the observational predicate *holds* are used to augment the definition of a related predicate *flips*. The work includes experiments in the blocks world. A typical rule has the following the form:

```
flips(move(X,Y),on(X,Y),+,Time) :-
    holds_at(clear(X),+,Time),holds_at(clear(Y),+,Time),...
```

which is very similar to the corresponding effect axiom in the Situation Calculus but for a narrative formalism. Moyle and Muggleton showed that it is possible to learn Event Calculus programs in ILP. Our work extends theirs for learning rules about action and change, and it is intended to be a general study on the learnability, in particular in formalisms based on Logic Programming. That includes how to cope with inertia, indirect effects, exceptions to general rules, concurrent actions, non-deterministic and other complex effects of actions, and so on.

10.3 Reinforcement Learning

Reinforcement learning [126] has a different objective other than learning a model for the effects of taking actions but *learning to act* in the environment, hence it requires a complete and correct model for the effects of each action. In reinforcement learning, simply stated, the learned rules tell you what action you should take in any state. A reward is assigned to each action after execution, so that the sum of rewards from all the visited states is guaranteed to be maximized. The current known learning algorithms back propagates the reinforcement values from the goal states to the non-goal states. The model of the system learned in this fashion is in the form states to actions, rather than states and actions to next states, as in action model learning.

10.4 Inference of deterministic finite-state automata

Dynamic system identification can be seen as inference of deterministic finite-state automata from sequences of input/output pairs. In this case, state transitions are assumed to be instantaneous and occur when some agent executes an action, otherwise the state of the system is assumed to be stable. The corresponding automaton for the circuit 2 is the following (where 1 and 2 denote sw_1 and sw_2 respectively, and t_i denotes $toggle(sw_i)$):

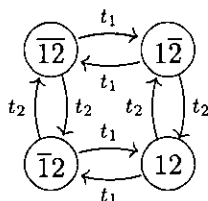


Figure 10.3: Automaton for the circuit 2

Automata identification methods have been applied to the problem of learning dynamic systems [41, 124, 115, 13]. These methods relied on the assumption that the agent can interact with the environment he is trying to learn [1] or can ask so-called *membership queries* and *equivalence queries* to an oracle. Efficient algorithms exist for the identification of automata, mainly in the area of Grammatical inference.

However, automata are not appropriate for modeling dynamic systems in many senses, e.g., limited expressivity, no elaboration tolerance, and so on. Furthermore, the use of (global) states instead of using state variables (fluents) makes unmanageable the representation of even small domains. For instance, certain episodes are difficult to manage with an automaton, such as those containing events where *no exact ordering* can be determined. These parallel episodes come as a consequence that most parts of the world are independent of most other parts because there is little interaction. In these cases, it is necessary to consider every possible ordering of the actions involved in the episode.

10.5 Learning Qualitative models

In [112], a method is presented to learn continuous dynamic systems in the form of a qualitative model. Qualitative reasoning (QR) is an elegant approach to studying the behavior of a physical system without going into as much detail as in a numerical simulation. In contrast to ordinary differential equations, QR captures distinctions that make an important qualitative difference and ignores others. For instance, the level at which a bathtub overflows is a qualitatively important point. The real number line in which variables take their values is described in terms of a finite set of qualitatively significant landmark values and the intervals between them. For instance, a variable describing water temperature might have landmark values for the freezing and boiling points.

Quantitative data is given and is converted into qualitative behaviors, so that, all but the “interesting” points in the data are discarded, i.e., points where some variable reaches a maximum, a minimum, or zero. For each variable at each time point, the quantitative value is turned into a *qualitative value* consisting of a qualitative magnitude and a direction-of-change. Qualitative states are added to behaviors as needed. If, for instance, a variable is at a minimum at one

time point and at a maximum at the next, the qualitative state for the interval during which the variable is increasing is added to the behavior. Then, individual constraints are generated and tested creating constraints consistent with the input behaviors (Fig. 10.4).

<u>Variable</u>	<u>Initial Qspace</u>	<u>Dimensions</u>
Inflow	$0, in1, \infty$	mass/time
Outflow	$0, \infty$	mass/time
Netflow	$-\infty, 0, net1, \infty$	mass/time
Amount	$0, \infty$	mass
<u>State 0 (initial state)</u>		successors: state 1
<u>Variable</u>	<u>Magnitude</u>	<u>Direction-of-change</u>
Inflow	$in1$	steady
Outflow	0	increasing
Netflow	$net1$	decreasing
Amount	0	increasing
<u>State 1</u>		successors: state 2
<u>Variable</u>	<u>Magnitude</u>	<u>Direction-of-change</u>
Inflow	$in1$	steady
Outflow	$(0, \infty)$	increasing
Netflow	$(0, net1)$	decreasing
Amount	$(0, \infty)$	increasing
<u>State 2</u>		successors: none
<u>Variable</u>	<u>Magnitude</u>	<u>Direction-of-change</u>
Inflow	$in1$	steady
Outflow	$out1$	steady
Netflow	0	steady
Amount	$amount1$	steady

Figure 10.4: Qualitative behavior of a simple bath tub

A qualitative differential equation (*QDE*) is valid over some *operating region*, expressed in terms of the values of the variables involved (*operating conditions*). Many systems cannot be described by a single QDE, but they are explained by different QDEs that hold under different operating conditions or regions. The movement of a system from the operating region of one QDE to that of another is called a *region transition*. A method for the detection of region transitions is added [109] that involves to detect the time points where the behavior moves from one region to another with a set of heuristics. Once some regions have been identified, an inductive learning algorithm is used to identify the operating conditions for each QDE. Actions can cause these transitions, but in many other cases, the transition can be fluent-triggered, where an special event in a fluent –e.g., reaching a landmark– causes the transition.

The identification of the operating conditions for the region transitions is indeed an action model learning task, whereas the identification of QDE and region transitions are necessary preprocessing steps for induction to work over qualitative data.

Chapter 11

Summary and Conclusions

11.1 Summary

The problem of learning complex action descriptions is an open question for the Machine Learning community. In this thesis, we have shown that it is possible to learn non-monotonic action theories in the form of logic programs, where Inductive Logic Programming methods can be applied effectively. We have shown it in simple simulated domains containing indirect effects, exceptions, and complex actions, that are considered benchmark problems in the literature of reasoning about actions and change.

Previous approaches suffer from the limitations of STRIPS-based formalisms or rely on procedural representations. Our approach is based on non-monotonic action theories, which are a formal and expressive representation for action domains, where system's behaviors are naturally viewed as appropriate logical consequences of the domain's description, which makes the learned specification of actions and their effects quite intuitive and natural. These languages make the notion of *situation* more central which provides a more compact description of observations and theories and a more convenient hypothesis space for learning. In particular, the form at which actions, effects and situations are represented in the Situation Calculus allows that multiple narratives starting from different initial situations can be used for learning in an homogeneous and natural way. This also allows that learning is done directly from the narratives, hence there is no need for preprocessing methods, and it allows inference to be done on the narratives to, for instance, implement inertia, complete missing values or learn recursive effects.

In particular, the use of Action Theories that are based on Logic Programming, allows a natural integration with Inductive Logic Programming methods, which provide a high expressivity and other special techniques, e.g., noise handling techniques, without losing representational power. This makes that an action theory is learned in the same way as it will be used, thus there is no a different representation for learning and another different for reasoning, planning and so on. We adopted Extended Logic Programs as the form of programs to be learned, where two kinds of negation –negation as failure and classical negation– are effectively used in the presence of incomplete information.

These logic-based formalisms allow to deal with issues like inertia, indirect effects and defeasible representations. Firstly, the use of non-monotonic formalisms avoids the explicit representation of unchanged properties from one particular situation to another in the observations (*Frame Problem*), hence observations need only be explicitly given for those situations where a fluent changes, whereas the inertia axiom propagates non-affected truth values from one situation to the next one, completing every situation. Secondly, the use of domain constraints together

with effect axioms allows the learner to infer how properties of a domain are (directly/indirectly) affected by the execution of actions, or otherwise are subject to the general law of inertia, thus avoiding the *Ramification Problem*. And thirdly, the use of a defeasible representation for the effects of actions allows the possibility of discovering *default rules* that describe the most common situations and the ability to manage and learn about exceptions. Furthermore, extensions to the Situation Calculus are used that allow to learn the effects of concurrent actions and other complex actions, as well as representing exogenous effects, without losing declarativeness.

On the other hand, the use of Action Languages allows that diverse learning tasks can be approached in an homogeneous way, for instance, to learn the effects of actions based on the action executed and the situation where the action is executed, but also the dual problem, i.e., *learning to act*, where the action theory of a domain is included in the background, and the learned rules must infer the best action to be executed based on the current situation and both the initial and the goal situations. The learned declarative control achieves its objective in a very elegant manner because the control rules are quite intuitive and purely declarative, and makes it possible to separate the control information from the implementation of the planner.

Action Theories represent a challenge to Machine Learning methods, where non-monotonic properties are introduced into the learning process, theories to be learned are mutually recursive (including cyclical dependences), two kinds of negation are used so that a three-valued setting is used for learning, and exceptions are possible through default negation. While the techniques introduced in this thesis are far from being a complete solution to the problem of learning in dynamic domains, we believe that it is a significant step forward in that direction.

11.2 Areas for future work

The work presented in this dissertation can be extended in several ways. The following lists a number of areas for future work that follows naturally from the work described in this thesis. Some of them address specific limitations, while others are broader issues that were beyond the scope of the thesis.

- A more efficient implementation should be developed to reduce the time used for learning. The most costly operation in the search process is the *coverage testing* made for every candidate hypothesis. The current implementation still adopts extensional coverage because of the computational cost of intensional coverage. Further work needs also to be done to assure global properties of the learned theory, i.e., the management of cyclical dependences and the global inconsistency problem.
- With regard to the learning of default theories, a reliable method of deciding whether to treat errors as noise or to include them as exceptions should be developed.
- We have used a non-narrative formalism for learning. This allows for instance, to use a single model in the background such that multiple narratives form a tree, thus avoiding to repeat common branches for each narrative. However, for long narratives, observations in the Situation Calculus become not manageable by some programs, for instance, the *smodels* interpreter. In narrative formalisms, the size of narratives does not affect the size of the observations and have other advantages, for instance, a simpler adaptation to manage concurrent actions and continuous change. By doing so, however, multiple models are to be provided in the background for coverage testing. In particular, we will consider the integration of the learning methods into the system PAL [21] for causal representation in

action domains, based on the concept of pertinence [101], which provides with an expressive and declarative formalism.

- The assumptions we have made are appropriate for tasks involving agents with higher level cognitive functions that involve reasoning about goals, actions, etc., but are a limitation of our approach when applied at the lowest level of control. The management of multi-valued and specially numerical fluents as well as the explicit management of delays and sequential effects in a theory of delayed causation, would allow to deal with more basic-level tasks.
- We have neither considered parameters that vary continuously as a consequence of a process execution. The topic of continuous change has received relatively little attention in the Reasoning about Actions literature [86, 120]. As pointed by R. Miller, a satisfactory general framework has to be developed which reconciles logic-based techniques for reasoning about action with the standard mathematical approach to modeling dynamic systems, using differential calculus. Some previous logic-based approaches to reasoning about actions do allow limited types of mathematical expressions involving continuously varying parameters to be embedded within domain descriptions which combine logic and differential equations. An alternative is the integration of Qualitative reasoning (QR) [62] into action theories, where fluents would represent *qualitative states* that hold under different operating conditions or regions which are triggered by events.
- Further work needs to be done to show the adaptation to more and increasingly more complex scenarios, and with different noise levels, e.g., to improve its adequation for dealing with real robot's environments. For a more practical application, exploration methods could potentially be added to reduce the dependence on an specific set of observations, where exploration and learning steps are interleaved. Furthermore, the implemented algorithm is not incremental, so the learned action theory must be completely re-learned every time a new instance is available.

Appendix A

Logic Programming concepts

This is a summary of basic concepts from logic programming. A *logic program* is a finite set of rules of the shape:

$$H \leftarrow L_1, \dots, L_n$$

where $n \geq 0$, H is an atom called the *head* of the rule and the L_i 's are program literals which receive the name of *body* of the rule. When $n = 0$, we say that H is a *fact*. An *atom* is of the form $p(t_1, \dots, t_k)$, where p is a predicate symbol and t_i 's are called *terms*. A *term* can be a constant, a variable, or $f(t_1, \dots, t_n)$, where t_1, \dots, t_n are terms and f is a n -ary function symbol. A *ground term* is a term without variables. Ground terms are used to describe objects. *Predicates* are about relations over these objects. A program *literal* is either an atom p or its default negation *not* p . A logic program is said to be *positive* (or definite) when it does not contain any negated literal. A clause is a *denial* if it has no positive literal. A Horn clause is either a definite clause or a denial.

A *substitution* $\theta = \{X_1/t_1, \dots, X_k/t_k\}$ is a function mapping variables to terms. The application $C\theta$ of a substitution θ to a clause C means replacing all the occurrences of each variable X_j in C by the same term t_j . A ground term is a term without variables. The *Herbrand universe* is the set of all possible terms that the theory can make assertions about, whereas the *Herbrand base* is the set of all possible ground facts that the theory can represent. It is obtained by taking the set of all predicate letters in the program and forming all possible ground instances of them using the Herbrand universe.

The following notation will be adopted: predicates, functions and constants start with a lowercase letter, while variable symbols start with an uppercase letter. Given the following program:

```
father(b,a).
father(e,d).
mother(c,a).
parent(X,Y) :- father(X,Y).
parent(X,Y) :- mother(X,Y).
```

the Herbrand universe is $\{a, b, c, d, e\}$, and the Herbrand base:

$\{father(a,b), father(b,a), \dots, parent(a,b), parent(b,a), \dots\}$.

An *interpretation* of a logic program is an assignment of truth or falsity to each element of the Herbrand base. A *model* of a logic program is an interpretation such that all the rules

of the program hold. In other words, if for each ground instance of each rule of the program $A \leftarrow B_1, B_2, \dots, B_n$, A is assigned true whenever all the B_i are assigned true. For instance, $\{\}$, $\{a\}$ and $\{a, b\}$ are models of $a \leftarrow b$. For positive programs, there exists a unique minimal model, called the *Least Herbrand Model* (LHM), which is the intersection of all the models and is taken as the declarative meaning of the logic program. The existence of a unique LHM is not guaranteed for non-positive programs. For instance, the simple program $a \leftarrow \text{not } b$ has two models $\{a\}$ and $\{b\}$ and both are minimal.

Appendix B

The stable models semantics

A normal logic program is a set of rules in the form:

$$A_0 \leftarrow A_1, \dots, A_m, \text{not } A_{m+1}, \dots, \text{not } A_n$$

where $0 < m < n$. If a program does not contain the *not* operator is called *positive*. Such positive programs have a unique stable model that coincides with the least Herbrand model. Even with normal logic programs, when the least Herbrand model exists, then that is the unique stable model. This is the case with stratified programs, i.e., programs without negative cycles. For the remaining cases, a normal logic program T can have none, one or several stable models. A program having exactly one stable model is called *categorical*. A unique stable model is a minimal model and it coincides with the least model in a Horn logic program. A program is *consistent* (under the stable model semantics) if it has a stable model, otherwise a program is inconsistent.

The stable model of a normal logic program is defined in the following form [44]: given a set of atoms I , for each atom a that is true in the I , we remove from the program all those rules that include *not a* as a condition, whereas for the remaining atoms, we remove all the conditions in the form *not a*. By doing so, the resulting program is a positive program. Thus, if the least model of the program coincides with I , then I is an stable model of the program. For instance, $\{a\}$ is the unique stable model of the program:

$$a \leftarrow \text{not } b$$

because the resulting program is $\{a \leftarrow\}$ whose least Herbrand model is $\{a\}$. The set $\{b\}$ is not a model of the implication, whereas $\{a, b\}$ is a superset of $\{a\}$. The insight is that for an atom to be in an stable model of a program, it needs to be in the head of one of the rules, such that all the conditions of the rule are part of the stable model.

A program with a positive cycle:

$$\begin{aligned} a &\leftarrow b \\ b &\leftarrow a \end{aligned}$$

has as stable model $\{\}$, i.e., the empty set, whereas by adding the rule $a \leftarrow$, the resulting program has a unique stable model $\{a, b\}$. And similarly for the program:

$$a \leftarrow a$$

A program with a negative cycle:

```
a ← not b
b ← not a
```

has two stable models $\{a\}$ and $\{b\}$, which represent two alternative *beliefs*. If we add a rule like $c \leftarrow a \vee b$, then c is included in both models and thus it is entailed by the program.

Normal logic programs may behave very differently when new rules are added, with respect to their previous meaning. For instance, the program:

```
a ← not a
```

has no stable model, however, the following theory is consistent and the stable model is $\{a\}$.

```
a ← not a
a ← not b
```

When multiple models exist, two forms of reasoning are possible. With *cautious* reasoning, a program P entails a ground atom A if A is true in every stable model of P . With *brave* reasoning, a program P entails a ground atom A if A is true in any stable model of P . Entailment refers to the common part of all models,

Answer set programming [66] is based on the view of program statements as constraints on the solution of a given problem, where each model of the program encodes a solution to the problem itself. Let us consider a node coloring problem: given a graph given as a set of nodes and edges find a way to color the nodes with 'n' colors such that two adjacent nodes are not colored with the same color. This problem can be represented with the following normal logic program:

```
color(red).
color(blue).
color(yellow).
node(a).
node(b).
node(c).
node(d).
edge(a,b).
edge(b,c).
edge(c,d).
edge(d,a).

col(X,red) :- node(X), not col(X, blue), not col(X,yellow).
col(X,blue) :- node(X), not col(X, red), not col(X,yellow).
col(X,yellow) :- node(X), not col(X, blue), not col(X,red).

:- edge(X,Y), color(C), col(X,C), col(Y,C).
```

The program above has 18 stable models, that represent possible colorings of the graph, among others:

```
Stable Model: col(d,red) col(c,blue) col(b,yellow) col(a,blue)
Stable Model: col(d,red) col(c,yellow) col(b,red) col(a,blue)
Stable Model: col(d,red) col(c,blue) col(b,red) col(a,blue)
...
```

The program *smodels* [97] is an efficient implementation of the stable models semantics. Computation of stable models is preceded by a phase of grounding of the program. Let us consider the following program:

```
winning(Y) :- move(Y,X), not winning(X).
move(a,b).
move(b,a).
```

During the grounding phase, the program is simplified in several forms. The grounded version of the program above is in the form:

```
move(b,a).
move(a,b).
winning(b) :- not winning(a).
winning(a) :- not winning(b).
```

From this grounded program, *smodels* computes two stable models.

```
smodels version 2.26. Reading...done
Answer: 1
Stable Model: move(b,a) move(a,b) winning(b)
Answer: 2
Stable Model: move(b,a) move(a,b) winning(a)
```


Bibliography

- [1] D. Angluin. On the complexity of minimum inference of regular sets. *Information and Control*, 39(3):337–350, 1978.
- [2] F. Bacchus and F. Kabanza. Using temporal logics to express search control knowledge for planning. *Artificial Intelligence*, 116(1–2):123–191, 2000.
- [3] L. Badea and S. Nienhuys-Cheng. A refinement operator for description logics. In J. Cussens and A. Frisch, editors, *Proceedings of the 10th International Conference on Inductive Logic Programming*, volume 1866 of *Lecture Notes in Artificial Intelligence*, pages 40–59. Springer-Verlag, 2000.
- [4] L. Badea and M. Stanciu. Refinement operators can be (weakly) perfect. In S. Džeroski and P. Flach, editors, *Proceedings of the 9th International Workshop on Inductive Logic Programming*, volume 1634 of *Lecture Notes in Artificial Intelligence*, pages 21–32. Springer-Verlag, 1999.
- [5] M. Bain. Experiments in non-monotonic first-order induction. In S. Muggleton, editor, *Proceedings of the International Workshop on Inductive Logic Programming*, pages 195–206, 1991.
- [6] N. Balac, D. Gaines, and D. Fisher. Learning action models for navigation in noisy environments. In *ICML Workshop on Machine Learning of Spatial Knowledge*, 2000.
- [7] N. Balac, D.M. Gaines, and D. Fisher. Using regression trees to learn action models. In *IEEE Systems, Man and Cybernetics Conference*, 2000.
- [8] C. Baral. Relating logic programming theories of actions and partial order planning. In *Theories of Action, Planning, and Robot Control: Bridging the Gap: Proceedings of the 1996 AAAI Workshop*, pages 18–27, Menlo Park, California, 1996. AAAI Press.
- [9] C. Baral and M. Gelfond. Reasoning about effects of concurrent actions. *Journal of Logic Programming*, 31(1–3):85–117, 1997.
- [10] C. Baral and J. Lobo. Defeasible specifications in action theories. In *Proceedings of the 15th International Joint Conference on Artificial Intelligence (IJCAI-97)*, pages 1441–1446, San Francisco, 1997. Morgan Kaufmann Publishers.
- [11] C. Baral and S.C. Tran. Relating theories of action and reactive control. *Linköping Electronic Articles in Computer Science*, 3(9), 1998.
- [12] C. Baral and L. Tuan. Reasoning about actions in a probabilistic setting. In *Symposium on Logical Formalizations of Commonsense Reasoning*, 2001.

- [13] K. Basye, T. Dean, and L. P. Kaelbling. Learning dynamics: System identification for perceptually challenged agents. *Artificial Intelligence*, 72, 1995.
- [14] K. Van Belleghem, M. Denecker, and D. T. Dupré;. Ramifications in an event-based language. In W. Daelemans K. Van Marcke, editor, *Proceedings of the Ninth Dutch Conference on Artificial Intelligence (NAIC'97)*, pages 227–236, 1997.
- [15] S. Benson. Inductive learning of reactive action models. In *Proc. 12th International Conference on Machine Learning*, pages 47–54. Morgan Kaufmann, 1995.
- [16] F. Bergadano, D. Gunetti, M. Nicosia, and G Ruffo. Learning logic programs with negation as failure. In L. De Raedt, editor, *Proceedings of the 10th International Conference on Inductive Logic Programming*, pages 33–52, 1995.
- [17] F. Botana and A. Bahamonde. SHAPE: A machine learning system from examples. *International Journal of Human-Computer Studies*, 42:137–155, 1995.
- [18] I. Bratko. *Prolog Programming for Artificial Intelligence*. Addison-Wesley, Reading, Massachusetts, second edition, 1990.
- [19] I. Bratko, S. Muggleton, and A. Varšek. Learning qualitative models of dynamic systems. In S. Muggleton, editor, *Inductive Logic Programming*, pages 437–452. Academic Press, 1992.
- [20] P. Brazdil and A. Jorge. Exploiting algorithm sketches in ILP. In S. Muggleton, editor, *Proceedings of the 3rd International Workshop on Inductive Logic Programming*, pages 193–204. J. Stefan Institute, 1993.
- [21] P. Cabalar. *Pertinence for causal representation of action domains*. PhD thesis, Departamento de Computación, Facultade de Informática, Univ. A Coruña, 2001. To appear.
- [22] J. G. Carbonell and Y. Gil. Learning by experimentation: The operator refinement method. In R. S. Michalski and Y. Kodratoff, editors, *Machine Learning: An Artificial Intelligence Approach, Volume III*. Morgan Kaufmann, San Mateo, California, 1990.
- [23] J. Chomicki. Efficient checking of temporal integrity constraints using bounded history encoding. *ACM Transactions on Database Systems*, 20(2):149–186, 1995.
- [24] L. Damas and V. S. Costa. Yap reference manual version 4.2.0, 1989-1999.
- [25] L. De Raedt and M. Bruynooghe. On negation and three-valued logic in interactive concept-learning. In L.C. Aiello, editor, *Proceedings of the 9th European Conference on Artificial Intelligence*, pages 207–212. Pitman, 1990.
- [26] M. Denecker, D. Theseider Dupré, and K. Van Belleghem. An inductive definition approach to ramifications. *Linkoping Electronic Articles in Computer Science*, 3(7), 1998.
- [27] M. desJardins. Knowledge development methods for planning systems. In *Planning and Learning: On to Real Applications: Papers from the 1994 AAAI Fall Symposium*, pages 34–40. AAAI Press, Menlo Park, California, 1994.

- [28] Y. Dimopoulos, S. Džeroski, and A. Kakas. Integrating explanatory and descriptive learning in ILP. In *Proceedings of the 15th International Joint Conference on Artificial Intelligence (IJCAI-97)*, pages 900–907, San Francisco, 1997. Morgan Kaufmann Publishers.
- [29] Y. Dimopoulos and A. Kakas. Learning non-monotonic logic programs: Learning exceptions. In N. Lavrač and S. Wrobel, editors, *Proceedings of the 8th European Conference on Machine Learning*, volume 912 of *Lecture Notes in Artificial Intelligence*, pages 122–137. Springer-Verlag, 1995.
- [30] P. Doherty and J. Kvarnstrom. Tackling the qualification problem using fluent dependency constraints: Preliminary report. In *International Workshop on Temporal Representation and Reasoning*, pages 97–104, 1998.
- [31] B. Duval and P. Nicolas. Learning default theories. In Anthony Hunter and Simon Parsons, editors, *Proceedings of the 5th European Conference on Symbolic and Quantitative Approaches to Reasoning and Uncertainty (ECSQARU-99)*, volume 1638 of *Lecture Notes in Artificial Intelligence*, pages 148–159, Berlin, 1999. Springer.
- [32] A. Esposito, D. Malerba, and F. A. Lisi. Learning recursive theories with ATRE. In H. Prade (Ed.), *Proceedings of the 13th European Conference on Artificial Intelligence*, pages 435–439. John Wiley & Sons, Chichester, England, 1998.
- [33] F. Esposito, D. Malerba, and F. Lisi. Induction of recursive theories in the normal ILP setting: Issues and solutions. In J. Cussens and A. Frisch, editors, *Proceedings of the 10th International Conference on Inductive Logic Programming*, volume 1866 of *Lecture Notes in Artificial Intelligence*, pages 93–111. Springer-Verlag, 2000.
- [34] T. A. Estlin and R. J. Mooney. Learning to improve both efficiency and quality of planning. In *Proceedings of the 15th International Joint Conference on Artificial Intelligence (IJCAI-97)*, pages 1227–1233, San Francisco, 1997. Morgan Kaufmann Publishers.
- [35] C. Feng. Inducing temporal fault diagnostic rules from a qualitative model. In S. Mugleton, editor, *International Workshop on Inductive Logic Programming*, pages 243–258, 1991.
- [36] R. E. Fikes and N. J. Nilsson. STRIPS: A new approach to the application of theorem proving to problem solving. In D. C. Cooper, editor, *Proceedings of the 2nd International Joint Conference on Artificial Intelligence*, pages 608–620, London, UK, 1971. William Kaufmann.
- [37] E. Fink and Q. Yang. Formalizing plan justifications. In *Proceedings of the Ninth Conference of the Canadian Society for Computational Studies of Intelligence (CSCSI92)*, pages 9–14, 1992.
- [38] P. A. Flach. The use of functional and logic languages in machine learning. In Maria Alpuente, editor, *Ninth International Workshop on Functional and Logic Programming (WFLP2000)*, pages 225–237, 2000.
- [39] P. A. Flach and N. Lavrac. The role of feature construction in inductive rule learning. In Luc De Raedt and Stefan Kramer, editors, *Proceedings of the ICML2000 workshop on Attribute-Value and Relational Learning: crossing the boundaries*, pages 1–11, Stanford, USA, 2000. 17th International Conference on Machine Learning.

- [40] M. Fox and D. Long. The detection and exploitation of symmetry in planning problems. In Dean Thomas, editor, *Proceedings of the 16th International Joint Conference on Artificial Intelligence (IJCAI-99-Vol2)*, pages 956–961, S.F., 1999. Morgan Kaufmann Publishers.
- [41] Y. Freund, M. J. Kearns, D. Ron, R. Rubinfeld, R. E. Schapire, and L. Sellie. Efficient learning of typical finite automata from random walks. In *Proceedings of the 25th Annual ACM Symposium on Theory of Computing*, pages 315–324, San-Diego, CA, 1993. ACM press.
- [42] K. Furukawa, T. Murakami, K. Ueno, T. Ozaki, and K. Shimazu. On a sufficient condition for the existence of most specific hypothesis in Progol. In Nada Lavrač and Sašo Džeroski, editors, *Proceedings of the 7th International Workshop on Inductive Logic Programming*, volume 1297 of *LNAI*, pages 157–164, Berlin, 1997. Springer.
- [43] A. Van Gelder, Kenneth A. Ross, and J. S. Schlipf. The well-founded semantics for general logic programs. *Journal of the ACM*, 38(3):620–650, 1991.
- [44] M. Gelfond and V. Lifschitz. The stable semantics for logic programs. In R. Kowalski and K. Bowen, editors, *Proceedings of the 5th international symposium on logic programming*, pages 1070–1080, Cambridge, MA., 1988. MIT Press.
- [45] M. Gelfond and V. Lifschitz. Classical negation in logic programs and disjunctive databases. *New Generation Computing*, 9(3–4):365–385, 1991.
- [46] M. Gelfond and V. Lifschitz. Representing action and change by logic programs. *Journal of Logic Programming*, 17:301–321, 1993.
- [47] M. Gelfond and V. Lifschitz. Action languages. *Electronic Transactions on AI*, 3(16), 1998.
- [48] M. Gelfond, V. Lifschitz, and A. Rabinov. What are the limitations of the situation calculus? In *Automated Reasoning: Essays in Honor of Woody Bledsoe*, pages 167–179. Kluwer Academic Publishers, Dordrecht, 1986.
- [49] M. Ginsberg. Reasoning about action I: A possible worlds approach. *Artificial Intelligence*, 35:165–195, 1988.
- [50] J. Gustafsson and P. Doherty. Embracing occlusion in specifying the indirect effects of actions. In *Principles of Knowledge Representation and Reasoning*, pages 87–98, 1996.
- [51] Y. Huang, B. Selman, and H. Kautz. Control knowledge in planning: Benefits and trade-offs. In *Proceedings of the 6th National Conference on Artificial Intelligence (AAAI-99); Proceedings of the 11th Conference on Innovative Applications of Artificial Intelligence*, pages 511–517, Menlo Park, Cal., 1999. AAAI/MIT Press.
- [52] Y. Huang, B. Selman, and H. Kautz. Learning declarative control rules for constraint-based planning. In *Proc. 17th International Conf. on Machine Learning*, pages 415–422. Morgan Kaufmann, San Francisco, CA, 2000.
- [53] D. Hume and C. Sammut. Applying inductive logic programming in reactive environments. In S. Muggleton, editor, *Proceedings of the International Workshop on Inductive Logic Programming*, pages 279–290, 1991.

- [54] K. Inoue and Y. Kudoh. Learning extended logic programs. In *Proc. of the Intl. Joint Conf. on Artificial Intelligence (IJCAI)*, 1997.
- [55] A. Kakas, E. Lamma, and F. Riguzzi. Learning multiple predicates. *Lecture Notes in Computer Science*, 1480:303–316, 1998.
- [56] A. Kakas and F. Riguzzi. Learning with abduction. In Nada Lavrač and Sašo Džeroski, editors, *Proceedings of the 7th International Workshop on Inductive Logic Programming*, volume 1297 of *Lecture Notes in Artificial Intelligence*, pages 181–188, Berlin, 1997. Springer.
- [57] L. Karlsson, J. Gustafsson, and P. Doherty. Delayed effects of actions. In *European Conference on Artificial Intelligence*, pages 542–546, 1998.
- [58] H. Kautz. The logic of persistence. In *Proceedings of the 5th National Conference of Artificial Intelligence*, pages 401–405, 1986.
- [59] H. Kautz and B. Selman. Planning as satisfiability. In J. Lloyd, editor, *Proceedings of the Tenth European Conference on Artificial Intelligence*, pages 359–379, 1992.
- [60] R. Khardon. Learning action strategies for planning domains. *Artificial Intelligence*, 113(1–2):125–148, 1999.
- [61] J. Koehler and J. Hoffmann. On reasonable and forced goal orderings and their use in an agenda-driven planning algorithm. *Journal of Artificial Intelligence Research*, 12:338–386, 2000.
- [62] B. J. Kuipers. *Qualitative Reasoning: Modeling and Simulation with Incomplete Knowledge*. MIT Press, Cambridge, MA, 1994.
- [63] E. Lamma, F. Riguzzi, and L. Moniz Pereira. Strategies in combined learning via logic programs. *Machine Learning*, 1/2(38):63–87, 2000.
- [64] C. Leckie and I. Zukerman. Inductive learning of search control rules for planning. *Artificial Intelligence*, 101(1–2):63–98, 1998.
- [65] H. J. Levesque, R. Reiter, Y. Lesperance, F. Lin, and R. B. Scherl. GOLOG: A logic programming language for dynamic domains. *Journal of Logic Programming*, 31(1-3):59–83, 1997.
- [66] V. Lifschitz. Answer set planning. In M. Gelfond, N. Leone, and G. Pfeifer, editors, *Proceedings of the 5th International Conference on Logic Programming and Nonmonotonic Reasoning (LPNMR-99)*, volume 1730 of *Lecture Notes in Artificial Intelligence*, pages 373–374, Berlin, 1999. Springer.
- [67] F. Lin. Embracing causality in specifying the indirect effects of actions. In C. S. Mellish, editor, *Proceedings of the International Joint Conference on Artificial Intelligence (IJCAI)*. Morgan Kaufmann, 1995.
- [68] F. Lin. On measuring plan quality (A preliminary report). In Anthony G. Cohn, Lenhart Schubert, and Stuart C. Shapiro, editors, *Proceedings of the 6th International Conference on Principles of Knowledge Representation and Reasoning (KR-98)*, pages 224–233, San Francisco, 1998. Morgan Kaufmann Publishers.

- [69] F. Lin and R. Reiter. How to progress a database (and why) I: Logical foundations. In Jon Doyle, Erik Sandewall, and Pietro Torasso, editors, *KR'94: Principles of Knowledge Representation and Reasoning*, pages 425–436. Morgan Kaufmann, San Francisco, California, 1994.
- [70] F. Lin and Y. Shoham. Concurrent actions in the situation calculus. In Paul Rosenbloom and Peter Szolovits, editors, *Proceedings of the Tenth National Conference on Artificial Intelligence*, Menlo Park, California, 1992. American Association for Artificial Intelligence, AAAI Press.
- [71] D. Lorenzo and R. P. Otero. Learning action theories with causality. In P. Flach and S. Džeroski, editors, *Proceedings of the Ninth Inductive Logic Programming Workshop (ILP99), Late Breaking Papers Volume*, 1999.
- [72] D. Lorenzo and R. P. Otero. Using an ILP algorithm to learn logic programs to reason about actions. In J. Cussens and A. Frisch, editors, *Proceedings of the Tenth Inductive Logic Programming Workshop (ILP00), Work in Progress Track*, 2000.
- [73] D. Lorenzo and R.P. Otero. Learning action theories as logic programs. In *International Joint Conference on Declarative Programming*, pages 383–396, 1999.
- [74] D. Lorenzo and R.P. Otero. Learning to reason about actions. In *W. Horn (Ed.), Proceedings of the 14th European Conference on Artificial Intelligence*, pages 435–439. IOS Press, Amsterdam, 2000.
- [75] L. Martin and C. Vrain. MULT_ICN: An empirical multiple predicate learner. In L. De Raedt, editor, *Proceedings of the 5th International Workshop on Inductive Logic Programming*, pages 129–144, 1995.
- [76] M. Martin and H. Geffner. Learning generalized policies in planning using concept languages. In Anthony G. Cohn, Fausto Giunchiglia, and Bart Selman, editors, *KR2000: Principles of Knowledge Representation and Reasoning*, pages 667–677, San Francisco, 2000. Morgan Kaufmann.
- [77] N. McCain and H. Turner. Causal theories of action and change. In Howard Shrobe and Ted Senator, editors, *Proceedings of the Thirteenth National Conference on Artificial Intelligence and the Eighth Innovative Applications of Artificial Intelligence Conference*, pages 460–465, Menlo Park, California, 1997. AAAI Press.
- [78] J. McCarthy. Programs with common sense. *Machine Intelligence*, 4:463–502, 1958.
- [79] J. McCarthy. Epistemological problems of artificial intelligence. In Tom Kehler and Stan Rosenschein, editors, *Proceedings of the Fifth International Joint Conference on Artificial Intelligence*, pages 1038–1044, Los Altos, California, 1977. Morgan Kaufmann.
- [80] J. McCarthy. Circumscription—A form of nonmonotonic reasoning. In V. Lifschitz, editor, *Formalizing Common Sense: Papers by John McCarthy*, pages 142–157. Ablex Publishing Corporation, Norwood, New Jersey, 1990.
- [81] J. McCarthy and P.J. Hayes. Some philosophical problems from the standpoint of artificial intelligence. *Machine Intelligence*, 4:463–502, 1969.

- [82] E. McCreath. *Induction of First Order Logic from Noisy training examples and fixed example set sizes*. PhD thesis, School of Computer Science and Engineering, 1999.
- [83] E. McCreath and A. Sharma. Extraction of meta-knowledge to restrict the hypothesis space for ilp systems. In *Proceedings of Eighth Australian Joint Conference on Artificial Intelligence*, pages 75–82, 1995.
- [84] A. McGovern and A. G. Barto. Automatic discovery of subgoals in reinforcement learning using diverse density. In *International Conference on Machine Learning*, pages 361–368, 2001.
- [85] R. Miller and M. Shanahan. Narratives in the situation calculus. *Journal of Logic and Computation*, 4(5):513–530, 1994.
- [86] R. Miller and M. Shanahan. Reasoning about discontinuities in the event calculus. In Luigia Carlucci Aiello, Jon Doyle, and Stuart Shapiro, editors, *KR'96: Principles of Knowledge Representation and Reasoning*, pages 63–74. Morgan Kaufmann, San Francisco, California, 1996.
- [87] T. M. Mitchell, P. E. Utgoff, and R. Banerji. Learning by experimentation: acquiring and refining problem-solving heuristics. In R. S. Michalski, J. G. Carbonell, and T. M. Mitchell, editors, *Machine Learning: An Artificial Intelligence Approach*, pages 163–190. Morgan Kaufmann, San Mateo, California, 1983.
- [88] S. Moyle and S. Muggleton. Learning programs in the event calculus. In N. Lavrač and S. Džeroski, editors, *Proceedings of the Seventh Inductive Logic Programming Workshop (ILP97)*, LNAI 1297, pages 205–212, Berlin, 1997. Springer-Verlag.
- [89] S. Muggleton. Inverse entailment and Progol. *New Generation Computing, Special issue on Inductive Logic Programming*, 13(3-4):245–286, 1995.
- [90] S. Muggleton. Completing inverse entailment. In *Proceedings of the International Workshop on Inductive Logic Programming*, pages 245–249, 1998.
- [91] S. Muggleton and C.H. Bryant. Theory completion using inverse entailment. In *Proc. of the 10th International Workshop on Inductive Logic Programming (ILP-00)*, Berlin, 2000. Springer-Verlag.
- [92] S. Muggleton and W. Buntine. Machine invention of first-order predicates by inverting resolution. In S. Muggleton, editor, *Inductive Logic Programming*, pages 261–280. Academic Press, 1992.
- [93] S. Muggleton and L. De Raedt. Inductive logic programming: Theory and methods. *Journal of Logic Programming*, 19/20:629–679, 1994.
- [94] S. Muggleton and C. Feng. Efficient induction of logic programs. In *Proceedings of the 1st Conference on Algorithmic Learning Theory*, pages 368–381. Ohmsma, Tokyo, Japan, 1990.
- [95] S. Muggleton and J. Firth. Cprogol 4.4 tutorial introduction, 1999.

- [96] I. Niemelä and P. Simons. Efficient implementation of the well-founded and stable model semantics. In Michael Maher, editor, *Proceedings of the 1996 Joint International Conference and Symposium on Logic Programming*, pages 289–303, Cambridge, 1996. MIT Press.
- [97] I. Niemelä and P. Simons. Smodels: An implementation of the stable model and well-founded semantics for normal LP. In Jürgen Dix, Ulrich Furbach, and Anil Nerode, editors, *Proceedings of the 4th International Conference on Logic Programming and Nonmonotonic Reasoning*, volume 1265 of *LNAI*, pages 420–429, Berlin, 1997. Springer.
- [98] S. Nienhuys-Cheng and R. de Wolf. A complete method for program specialization based on unfolding. In W. Wahlster, editor, *Proceedings of the 12th European Conference on Artificial Intelligence*, pages 438–442. John Wiley, 1996.
- [99] T. Oates and P. R. Cohen. Learning planning operators with conditional and probabilistic effects. In *Planning with Incomplete Information for Robot Problems: Papers from the 1996 AAAI Spring Symposium*, pages 86–94. AAAI Press, Menlo Park, California, 1996.
- [100] M. Otero and R. P. Otero. Using causality for diagnosis. In *Eleventh International workshop of Principles of Diagnosis (DX'00)*, Morelia (Mexico), 2000.
- [101] R. P. Otero and P. Cabalar. Pertinence and causality. In *Proceedings of the 3rd Workshop on Nonmonotonic Reasoning, Action, and Change (NRAC), IJCAI'99*, 1999.
- [102] R. P. Otero and O. G. Trinidad. Action formalisms in language understanding. In Christof Monz and Maarten de Rijke, editors, *First Workshop on Inference in Computational Semantics (ICoS-1)*, Amsterdam, 1999. Institute for Logic, Language and Computation.
- [103] D. J. Pearson. Learning procedural planning knowledge in complex environments. In *Proceedings of the Thirteenth National Conference on Artificial Intelligence and the Eighth Innovative Applications of Artificial Intelligence Conference*, pages 1401–1401, Menlo Park, 1996. AAAI Press / MIT Press.
- [104] J. Pinto and R. Reiter. Temporal reasoning in logic programming: A case for the situation calculus. In David S. Warren, editor, *Proceedings of the Tenth International Conference on Logic Programming*, pages 203–221, Budapest, Hungary, 1993. The MIT Press.
- [105] G. D. Plotkin. A note on inductive generalisation. In B. Meltzer and D. Michie, editors, *Machine Intelligence 5*, pages 153–163. Elsevier North Holland, New York, 1970.
- [106] J.R. Quinlan. Learning logical definitions from relations. *Machine Learning*, 5:239–266, 1990.
- [107] J.R. Quinlan. *C4.5: Programs for Machine Learning*. Morgan Kaufmann, San Francisco CA, 1993.
- [108] L. Raedt and N. Lavrac. Multiple predicate learning in two ILP settings. *Logic J. of the IGPL*, 4(2):227–254, 1996.
- [109] S. Ramachandran, R. Money, and B. J. Kuipers. Learning qualitative models for systems with multiple operating regions. Technical report, Department of Computer Science. University of Texas, 1994.

- [110] R. Reiter. Natural actions, concurrency and continuous time in the situation calculus. In Luigia Carlucci Aiello, Jon Doyle, and Stuart Shapiro, editors, *KR'96: Principles of Knowledge Representation and Reasoning*, pages 2–13. Morgan Kaufmann, San Francisco, California, 1996.
- [111] R. Reiter. Knowledge in action: logical foundation for describing and implementing dynamical systems, 1998. Manuscript.
- [112] B. L. Richards, I. Kraan, and B. J. Kuipers. Automatic abduction of qualitative models. *Proceedings of the Fifth International Workshop on Qualitative Reasoning about Physical Systems*, pages 295–301, 1991.
- [113] J. Rintanen. Incorporation of temporal logic control into plan operators. In Werner Horn, editor, *Proceedings of the 14th European Conference on Artificial Intelligence*, pages 526–530, Amsterdam, 2000. IOS Press.
- [114] J. Rintanen. An iterative algorithm for synthesizing invariants. In *Proceedings of the 17th National Conference on Artificial Intelligence*, pages 806–811. The AAAI Press, 2000.
- [115] R. L. Rivest and R. E. Schapire. Diversity-based inference of finite automata. In Ashok K. Chandra, editor, *Proceedings of the 28th Annual Symposium on Foundations of Computer Science*, pages 78–88. IEEE Computer Society Press, 1987.
- [116] C. Rouveirol. Flattening and saturation: Two representation changes for generalization. *Machine Learning*, 14(2):219–232, 1994.
- [117] S. J. Russell and P. Norvig. *Artificial Intelligence. A Modern Approach*. Prentice-Hall, Englewood Cliffs, 1995.
- [118] C. Sakama. Some properties of inverse resolution in normal logic programs. In *Proceedings of the Ninth International Conference on Inductive Logic Programming*, volume 1634 of *Lecture Notes in Artificial Intelligence*, pages 279–290. Springer-Verlag, 1999.
- [119] C. Sakama. Inverse entailment in nonmonotonic logic programs. In J. Cussens and A. Frisch, editors, *Proceedings of the 10th International Conference on Inductive Logic Programming*, volume 1866 of *Lecture Notes in Artificial Intelligence*, pages 209–224. Springer-Verlag, 2000.
- [120] E. Sandewall. Combining logic and differential equations for describing real-world systems. In Hector J. Levesque Ronald J. Brachman and Raymond Reiter, editors, *Proceedings of the 1st International Conference on Principles of Knowledge Representation and Reasoning*, pages 412–420, Toronto, Canada, 1989. Morgan Kaufmann.
- [121] M. Shanahan. *Solving the Frame Problem. A Mathematical Investigation of the Common Sense Law of Inertia*. The MIT Press, 1997.
- [122] M. Shanahan. A logical account of the common sense informatic situation for a mobile robot. *Linköping Electronic Articles in Computer Science*, 1999.
- [123] M. Shanahan. The ramification problem in the event calculus. In Dean Thomas, editor, *Proceedings of the 16th International Joint Conference on Artificial Intelligence (IJCAI-99-Vol1)*, pages 140–146, S.F., 1999. Morgan Kaufmann Publishers.

- [124] W. M. Shen. *Autonomous Learning from the Environment*. Computer Science Press, 1994.
- [125] A. Srinivasan, S. Muggleton, and M. Bain. Distinguishing exceptions from noise in non-monotonic learning. In *2nd Inductive Logic Programming Workshop*, 1992.
- [126] R. S. Sutton. Learning to predict by methods of temporal differences. *Machine Learning*, 3:9–44, 1988.
- [127] M. Thielscher. Causality and the qualification problem. In Luigia Carlucci Aiello, Jon Doyle, and Stuart Shapiro, editors, *Proceedings of the Fifth International Conference on Principles of Knowledge Representation and Reasoning*, pages 51–62, San Francisco, 1996. Morgan Kaufmann.
- [128] M. Thielscher. Ramification and causality. *Artificial Intelligence Journal*, 1-2(89):317–364, 1997.
- [129] L. Todorovski, S. Džeroski, A. Srinivasan, J. Whiteley, and D. Gavaghan. Discovering the structure of partial differential equations from example behavior. In *Proc. 17th International Conf. on Machine Learning*, pages 991–998. Morgan Kaufmann, San Francisco, CA, 2000.
- [130] M. Veloso. *Learning by Analogical Reasoning in General Problem Solving*. PhD thesis, Department of Computer Science, Carnegie Mellon University, 1992.
- [131] M. Veloso, J. Carbonell, A. Pérez, D. Borrajo, E. Fink, and J. Blythe. Integrating planning and learning. *Journal of Experimental and Theoretical Artificial Intelligence*, 7(1), 1995.
- [132] X. Wang. Learning by observation and practice: an incremental approach for planning operator acquisition. In *Proc. 12th International Conference on Machine Learning*, pages 549–557. Morgan Kaufmann, 1995.
- [133] D. S. Weld. Recent advances in ai planning. *AI Magazine*, 20(2):93–123, 1999.
- [134] A. Yamamoto. Which hypotheses can be found with inverse entailment? In Nada Lavrač and Sašo Džeroski, editors, *Proceedings of the 7th International Workshop on Inductive Logic Programming*, volume 1297 of *Lecture Notes in Artificial Intelligence*, pages 296–308, Berlin, 1997. Springer.

UNIVERSIDADE DA CORUÑA
Servicio de Bibliotecas



1700757442