An investigation into Theory Completion techniques in Inductive Logic Programming

Stephen Moyle

Worcester College



Programming Research Group
Oxford University Computing Laboratory
Wolfson Building
Parks Road
Oxford OX1 3QD

Thesis submitted in partial fulfilment
of the requirements for the degree of
Doctor of Philosophy.

Trinity, 2000.
(Resubmitted Trinity, 2003)

An investigation into Theory Completion techniques in Inductive Logic Programming

Stephen Moyle

Worcester College

Programming Research Group
Oxford University Computing Laboratory
Wolfson Building
Parks Road
Oxford OX1 3QD

Thesis submitted for the degree of Doctor of Philosophy.

University of Oxford, Trinity 2003.

Abstract: Traditional Inductive Logic Programming (ILP) focuses on the setting where the target theory is a generalisation of the observations. This is known as Observational Predicate Learning (OPL). In the Theory Completion setting the target theory is not in the same predicate as the observations (non-OPL). This thesis investigates two alternative simple extensions to traditional ILP to perform non-OPL or Theory Completion. Both techniques perform extraction-case abduction from an existing background theory and one seed observation. The first technique – Logical Back-propagation – modifies the existing background theory so that abductions can be achieved by a form of constructive negation using a standard SLD-resolution theorem prover. The second technique – SOLD-resolution – modifies the theorem prover, and leaves the existing background theory unchanged. It is shown that all abductions produced by Logical Back-propagation can also be generated by SOLD-resolution; but the reverse does not hold. The implementation using the SOLD-resolution technique – the `ALECTO` system – was applied to the problems of completing context free and context dependant grammars; and learning Event Calculus programs. It was successfully able to learn an Event Calculus program to control the navigation of a real-life robot. The Event Calculus is a formalism to represent common-sense knowledge. It follows that the discovery of some common-sense knowledge was produced with the assistance of a machine.

Keywords: Machine Learning, Inductive Logic Programming, Theory revision, Theory Completion, Logical Back-propagation, SOLD-resolution, Abduction, Event Calculus.

Longer abstract

This thesis is concerned with machine assisted theory formation in scientific discovery. Theory Completion is considered as the augmenting of an existing theory with new hypotheses so that new observations can be explained. The generalisation of observations to form a theory is common to many forms of Machine Learning, however Inductive Logic Programming (ILP) is one of the few techniques that is able to utilise existing theories. Theories in ILP are represented in the form of Logic Programs. Hypotheses produced by ILP are sets of Horn clauses which form the description of the new scientific discoveries. A further advantage possessed by ILP is that these hypotheses can be communicated to humans so that the new discoveries can be understood. Discoveries in science can benefit from two–way interactions between a human and a machine assistant – in which the human provides the assistant with the existing theory, surprising unpredicted observations, and some hunch about the nature of the missing theory components; while the machine assistant communicates its plausible Theory Completions that explain the observations to the human.

Inductive Logic Programming provides theory and methods for proposing hypotheses based on an incomplete existing background theory and some seed observation. The main technique investigated here for this purpose is Inverse Entailment. Another ILP technique used is the enumeration of plausible hypotheses. These hypotheses can then be ranked according to varying criteria. ILP systems have contributed to many scientific discoveries, however they do require a substantial amount of human input. Human input is required to encode the existing theory, the observations, and a raft of settings to encode the scientist's hunches about the "missing theory".

One limitation of the standard ILP setting is that the description of the theories generated are often restricted to being in the same language – or predicate – of the observations. This is called Observational Predicate Learning (OPL). There are, however, circumstances in scientific theory formation where the observations provide evidence for an alternative description that is desired – this is the non-OPL setting. To perform Theory Completion in the non-OPL setting requires extensions to the implementation techniques for performing Inverse Entailment. Two such extensions are investigated: Logical Back-propagation and SOLD-resolution. Both of

these techniques have the similar objective of producing a negative explanation for the non explanation of the seed example in the presence of the existing background theory. In logic programming terms this equates to the derivation of negated facts (or goal clauses) from a definite program and a goal clause.

In Logical Back-propagation, to test for the derivation of negated facts, first the background theory is required to be translated and additional clauses for the negated facts added. The derivation of any true negated facts can then proceed using standard SLD-resolution (for example by using a PROLOG system). Logical Back-propagation is a form of constructive negation. When using the second technique, SOLD-resolution, to derive negated facts, the background theory clauses remain unchanged, but the mechanism for deriving them is altered. To derive the true negated atoms from the background theory and the negated observation a SOLD-resolution system must be used. This, however, is straightforward to implement in PROLOG as a *meta interpreter*. Each of these methods for extending Inverse Entailment were investigated. Both techniques are related to extraction-case abduction which "extracts" facts to produce an explanation for an observation in the presence of an existing theory.

The techniques of Logical Back-propagation have previously been implemented in the Theory Completion by Inverse Entailment system `CProgol5.0`. An implementation based on the SOLD-resolution method of extraction-case abduction is developed in this work. The resulting system, `ALECTO`, has a similar philosophy to `CProgol5.0`. This is the philosophy of providing simple extensions to the Inverse Entailment approach, to cater for non observation predicate learning. `ALECTO` first uses SOLD-resolution and other techniques proposed by Yamamoto to translate the observations into ground abductive explanations. It is these abductive explanations that are generalised in the usual ILP manner. When evaluating proposed hypotheses `ALECTO` uses a positive-only data scoring function. While searching for a theory, `ALECTO` assumes that all yet to be processed abductions are true. This strategy allows for multiple-predicate learning to be achieved in a serial process, and it is similar to one used in the Abductive Concept Learning system `MACL`.

Three application domains have been used to investigate the performance of the Theory Completion systems `CProgol5.0` and `ALECTO`. The following three application domains were utilised: completing a grammar, completing a semantic intrusion detection system, and learning Event Calculus programs. In these experiments the Theory Completion setting was restricted so that the existing background theories prior to completion contained **no** definitions of the desired predicates. The first set of experiments study which combinations of deleted definitions can be regenerated for a simple context free grammar. Although simple, the grammar contains enough complexity and subtleties for the purpose of investigating Theory Completion. In these experiments `ALECTO` performed considerably better than `CProgol5.0`,

particularly when two definitions were required to be completed "simultaneously".

Experiments were performed to recover clauses from the context dependent grammar for detecting intrusions. The first set of experiments studied single, top-level, deleted definitions and their regeneration from the context sensitive grammar. In the second experiment two deleted top-level definitions were regenerated. The third experiment attempted the recovery of a deleted definition from a lower level.

Two different experiments to complete Event Calculus programs were performed. Event Calculus programs have a well defined theory structure which include a set of immutable core axioms, a narrative of events, an initial state, and a set of domain specific axioms. There are strict requirements when writing Event Calculus programs that the *domain specific axioms* are the components to be constructed. When finished, the domain specific axioms can be used, in conjunction with the other components, to generate the desired (hopefully) dynamic behaviour of the Event Calculus program. The structure of an Event Calculus program is particularly well suited to Theory Completion. The experiments performed were designed to determine whether Theory Completion techniques could be used to produce the domain specific axioms from the core axioms – or *empty theory* – of the Event Calculus and observations of desired dynamic behaviour.

The work on learning Event Calculus programs was, in part, motivated from earlier work in which `CProgol5.0` was used to generate domain specific axioms for the simple Blocks World Event Calculus program. Limitations in using `CProgol5.0` included the need to pre-process the data to produce forms of negative observations. The most serious concern, however, was the need to re-code the two Event Calculus transition predicates *initiates* and *terminates* into a single *flips* predicate. `ALECTO` was applied to a previously used Blocks World data set and achieved similar results to the earlier work, but without the need for re-coding, nor the need for the construction of negative information.

`ALECTO` was applied to learning a more substantial Event Calculus program – a program to control the navigation of a real-life robot. The original control program is part of a suite of robot programs incorporating sensing, planning, and acting – all written in the Event Calculus. The robot navigation control program consists of three sub-components: 1) the domain specific axioms that describe the direct effects of actions in specific situations; 2) definitions for ramifications – or indirect effects of actions; and 3) background knowledge consisting, in part, of the layout of a model office (the robot's environment). The original program was manually adapted to a slightly different formulation of the Event Calculus, and also to account for a wider class of dynamic behaviour. With the alterations the robot navigation control program consisted of 13 pairs of initiates/terminates domain specific axioms.

The data for the experiments in learning the robot navigation control program was generated by executing all of the components together to produce time trace data of the dynamic behaviour of the robot. Theory Completion was then used to take the "existing" Event Calculus program (emptied of the domain specific axioms), and the time trace data to induce replacement domain specific axioms. ALECTO was applied to 30 different runs where each run consisted of observing the (simulation of the) robot navigate through the model office from a random start point to a random finishing point. The domain specific axioms produced by ALECTO from the data for each run were evaluated on the remaining 29 data sets. The estimated average accuracy of the ALECTO generated domain specific axioms over the 30 runs was 70.2%, with a standard deviation of 6.8%. The average of the standard deviations within each run was 13.1% The average number of clauses in the 30 domain specific axiom sets generated was 14.

The development of the application of ALECTO to the learning of Event Calculus programs has produced a framework which should be applicable to other Event Calculus settings. The experimental results achieved by ALECTO in learning Event Calculus program suggest the possibility that discoveries of dynamic behaviour for some natural phenomenon may, in the future, be assisted by a machine.

The Theory Completion techniques investigated in this thesis fall well short of the techniques that would be required to build a truly usable machine assistant for formulating scientific discoveries. Such an assistant would also be required to revise theories, as well as have better abilities to communicate with a human. It should also be able to suggest that the existing theory be completely abandoned, and propose a plausible alternative.

Preface

The motivation for this work came from my time working in the U.K. offshore gas industry when I was installing computerised supervisory control systems for the control and monitoring of two - then new - natural gas producing platforms. One of the tasks that I performed was to configure a "real-time" database with historical archiving. This was relatively uncomplicated and nothing that could be called "frontier technology". Such archiving was capable of recording a large number of sensor readings for long periods of time, which could be replayed as time trend charts when desired. The system was typically used by operators and production engineers, who could use the trend charts, allied with their experience, production knowledge, and common sense, to "better operate" the process. One thing that struck me was that although large amounts of process data were being archived, the human resources to glean knowledge from the trends were not abundant.

What was needed was an automated system that could combine some attributes of the human operators - knowledge of the process and common sense - with the recorded observations of the dynamic process behaviour, and could *glean* knowledge for later use by humans.

With this as my motivation, I started my research program with the Machine Learning Group at the Oxford University Computing Laboratory. Stephen Muggleton introduced me to the *Frame Problem* and suggested that I began by investigating the Situation and Event Calculi. Fortunately at that time John McCarthy was visiting the Laboratory and in a brief meeting inspired me towards the field of *Common sense reasoning*. From that time I began to pursue the idea of applying Inductive Logic Programming to learning Event Calculus programs. Shortly thereafter, while visiting Oxford, Murray Shanahan strongly encouraged this course of research.

Acknowledgments

My research was aided by many individuals. Stephen Muggleton played an important role as my early supervisor - introducing me to many facets of AI, machine learning, and especially Inductive Logic Programming. He also provided the necessary enhancements to his ILP system, `CProgol5.0`, that

enabled early experimental work to be performed. Other members of the then Oxford University Computing Laboratory Machine Learning Group provided support - C. David Page, James Cussens, Rupert Parson, and Donald Michie - a visiting member of the Group. Jouni Kuha and David Firth, of Nuffield College, suggested the statistical techniques to analyse the experimental results. Tony Hoare provided wisdom and supervisory support while I was between supervisors. Vítor Santos Costa's improvements to his `Yap` PROLOG system made implementation easier. Murray Shanahan kindly provided parts of his robot control program and simulator which were used in some of the experiments. Khalid Khan much improved my understanding of different notions of generality, and more importantly, improved the dissertation by careful proof reading.

In particular, I am indebted to Ashwin Srinivasan who formally supervised the latter stages of this work. His supervision was only a fragment of what I gleaned from his sizeable store of knowledge. Ashwin instilled in me a valuable sense of pragmatism, and helped me to maintain a sense of perspective. He also provided many enhancements to his ILP system `ALEPH` so that it could be a foundation for my implementation.

I am grateful to the ESPRIT IV Long Term Research Project Number 20237 Inductive Logic Programming II for contributing travel funds. The final experiments were performed on equipment provided by European Union project IST-1999-11495:SolEuNet. Closer to home, the Nuffield College IT team were very supportive, providing comfortable office accommodation and computing resources.

Many friends have influenced my thinking. In particular I mention Jurgen Doornik whose discussions were stimulating. Finally, my wife Penny provided me with her unreserved support.

S.A.M.
September 2000

Contents

Abstract **i**

Abstract (long) **ii**

Preface **vi**

I Theory Completion: theory and implementation **1**

1 Introduction **2**
1.1 Theory Completion 3
1.2 Automated assistants for theory formation 3
1.2.1 Symbolic Machine Learning 3
1.2.2 Inductive Logic Programming 4
1.3 Theory Completion in Inductive Logic Programming 6
1.3.1 Theory Completion in ILP and the relation to Scientific Method 9
1.4 Related work 13
1.5 Thesis outline 15

2 Machine Induction of First Order Theories by Inductive Logic Programming **17**
2.1 The general setting for ILP 17
2.1.1 The logical conditions for ILP 19
2.1.2 Different ILP settings 21
2.2 Hypothesis generation 22
2.2.1 Inverse Entailment 22
2.2.2 Implementing Inverse Entailment 25
2.2.3 A definite mode language for Inverse Entailment . . . 30
2.3 Alternative hypotheses 31
2.3.1 Generality orderings and ILP 32
2.3.2 The generality ordering of θ-subsumption 33
2.4 Hypothesis testing 36
2.4.1 Selecting from a set of competing hypotheses 36

2.4.2 Meta-logical search control . . . 37
2.4.3 An introduction to Positive-only scoring . . . 39
2.5 A basic ILP algorithm . . . 41
2.6 Summary . . . 44

3 Theory Completion by Inverse Entailment 45
3.1 An introduction to Theory Revision and Theory Completion 46
3.1.1 Theory Revision . . . 46
3.1.2 Theory Completion . . . 47
3.2 Inverse Entailment Revisited . . . 48
3.2.1 The role of Abduction in Theory Completion . . . 48
3.3 Extraction-case Abduction using Logical Back-propagation . 50
3.3.1 Transforming the Definite Logic Program for LBP . . 51
3.4 Extraction-case Abduction using SOLD-Resolution . . . 55
3.5 Comparisons of SOLD-resolution and Logical Back-propagation 59
3.6 Implementing SOLD-resolution . . . 61
3.6.1 Further steps towards Inverse Entailment . . . 63
3.7 Discussion . . . 69
3.8 Summary . . . 70

4 ALECTO: An ILP system for Theory Completion 72
4.1 Introduction . . . 72
4.1.1 Extending the ILP system ALEPH . . . 73
4.2 Transformations of observations by generating Abductive explanations . . . 74
4.3 Transforming observations . . . 75
4.3.1 Justifications for the transformation of observations into crosslists . . . 79
4.4 Controlling the search for theories . . . 80
4.4.1 Coping with incomplete hypotheses . . . 80
4.4.2 An abductive Positive-Data Only Scoring function . . 81
4.5 The ALECTO algorithm . . . 82
4.5.1 Complexity issues . . . 86
4.6 Discussion . . . 88
4.7 Summary . . . 90

II Applications of ALECTO 92

5 Application 1: Learning components of a Natural Language Grammar 93
5.1 Introduction . . . 93
5.2 General Experimental Methodology . . . 94
5.3 The toy example grammar domain . . . 94

5.4 Objective . . . 96
5.5 Method . . . 96
5.6 Background theory specification . . . 97
5.7 Results . . . 98
5.8 Discussion . . . 99
5.9 Summary . . . 100

6 Application 2: Learning components of an Intrusion Detection System 103
6.1 Introduction . . . 103
6.1.1 The buffer overflow attack . . . 104
6.1.2 An Intrusion Detection System for recognising buffer overflow attacks . . . 106
6.2 Experiments for Theory Completion . . . 109
6.2.1 Objective . . . 109
6.3 Method . . . 110
6.3.1 Input components for the IDS Theory Completion . . . 111
6.4 Results . . . 112
6.5 Discussion . . . 113
6.6 Summary . . . 115

7 Application 3: Learning Event Calculus programs by Theory Completion 116
7.1 Introduction . . . 117
7.2 Introduction to the Event Calculus . . . 118
7.2.1 Components of Event Calculus programs . . . 121
7.2.2 The Event Calculus as a Logic Program . . . 123
7.2.3 Knowledge representation in the Event Calculus . . . 125
7.2.4 Programming in the Event Calculus . . . 125
7.3 Learning Event Calculus programs . . . 126
7.3.1 Background and Motivation . . . 126
7.3.2 Objective . . . 127
7.3.3 Method . . . 127
7.3.4 Common background knowledge specification . . . 128
7.3.5 An Event Calculus program to describe the blocks world 131
7.3.6 An Event Calculus program to control a robot's navigation . . . 133
7.4 Learning Domain Specific Axioms for the Blocks World . . . 141
7.4.1 Input components specific to the Blocks World . . . 141
7.4.2 Results . . . 142
7.5 Learning Domain Specific Axioms for the Robot Navigation Control program . . . 143
7.5.1 Input components specific to the Robot Navigation Control program . . . 143

7.5.2 Experimental Method 144
7.5.3 Results 146
7.6 Discussion 149
7.6.1 Related Work 150
7.7 Summary 151

III Discussion and Concluding Remarks 152

8 Discussion and Concluding Remarks 153
8.1 Summary of the Thesis 153
8.2 Discussion 156
8.2.1 Theory Completion Techniques in ILP 156
8.2.2 Applications requiring Theory Completion 159
8.2.3 Logical Styles of reasoning and uses of the Event Calculus 162
8.2.4 Learning common sense knowledge 162
8.3 Limitations 164
8.4 Future Directions 166
8.4.1 Theory and methods 166
8.4.2 Application domains and settings 167
8.4.3 Other systems 169
8.4.4 Evaluating the robot navigation control program . . . 169
8.4.5 Other 169
8.5 Concluding Remarks 170

A List of symbols 171

B Appendix to Chapter 5 174
B.1 Observations available for completing the grammar 174
B.2 Background knowledge for completing the grammar 175
B.3 Experimental results from completing the grammar 179
B.3.1 Theories generated by `CProgol5.0` for the grammar . 179
B.3.2 Theories generated by `ALECTO` for the grammar 181

C Appendix to Chapter 6 184
C.1 Observations available for completing the grammar 184
C.2 Background knowledge 184
C.3 Mode Declarations 184
C.4 Experimental Results 189
C.4.1 Learning Curves 189
C.4.2 Sample Theories 192

D Appendix to Chapter 7 **194**
D.1 Supporting material, common to both Event Calculus problems 194
D.2 Supporting material for learning Domain Specific Axioms to describe the Blocks World 197
D.3 Supporting material for learning Domain Specific Axioms for a Robot Navigation Control program 201

List of Figures

1.1 The partially complete grammar theory for parsing sentences, $B_{grammar}$, and the observations $E_{grammar}$ that the completed grammar should be able to explain as *parsable*. 8
1.2 The completed definitions for *verb_phrase* and *noun_phrase*. The complete grammar parser consisting of the original unfinished program with the additional definitions $B_{grammar} \cup H_{grammar}$, will explain – or parse – all observations, $E_{grammar}$. 8

2.1 Derivation of $\overline{verb_phrase([has, a, cat], [])}$ by linear resolution from $B_{grammar_0}$ and $\overline{e_1^+}$. 24
2.2 An ILP algorithm based on Inverse Entailment. 42
2.3 An algorithm for generating a most specific (bottom) clause (adapted from [Muggleton, 1995]). 43

3.1 The derivation of $\neg verb_phrase([has, a, cat], [])$ by linear resolution from $B'_{grammar_0} \cup \neg e_1^+$ and $\neg sentence([the, president, has, a, cat], [])$. Notice that the dashed arrows indicate some resolution steps are not shown. 53
3.2 An algorithm to generate most specific clauses from a single observation e^+ and background knowledge B, using Inverse Entailment extended with Logical Back-propagation. 54
3.3 The derivation of $\overline{verb_phrase([has, a, cat], [])}$ by SOLD-resolution from $B_{grammar_0}$ and $\overline{e_1^+}$. 58
3.4 The derivation by SOLD-resolution from $B_{grammar_0}$ and $\overline{e_1^+}$ of $\overline{verb_phrase(T, [])} \vee \overline{noun_phrase([the, president, has, a, cat], T)}$. 60
3.5 An algorithm to generate most specific clauses from a single observation e^+ and background knowledge B, using Inverse Entailment extended with SOLD-resolution. 67

4.1 The transformation from observations to *h-lists*. 76
4.2 An algorithm for generating *crosslists* from background knowledge B, and observations E^+. 77
4.3 An algorithm for abductively generalising *crosslists*. 83

4.4 A simple best-first search algorithm for abductively *searching* for the best clause. The score of each clause can be obtained by the *score* algorithm presented in figure 4.5. 84
4.5 An algorithm for abductively *scoring* a clause with respect to the partially complete theory (from definition 75). 85
4.6 The `ALECTO` algorithm for Theory Completion. 86

5.1 The calling diagram for the simple grammar. 95

6.1 Components of a malicious transaction containing a buffer overflow attack. 105
6.2 The calling diagram for the complete Intrusion Detection System [Moyle and Heasman, 2002]. 108
6.3 The experimental procedure for the recovery of IDS target clauses by Theory Completion. Note that for the experiments $n = 10$, $M = 824$. 112
6.4 Learning curves for regenerating `consume_params/5` and `consume_null_operation/5` (separately). The estimates of accuracy are averaged from 10 fold cross-validation, while error bars are 1 population standard deviation. 113
6.5 Run times for regenerating `consume_params/5` and `consume_null_operation/5` (separately). The run times are averaged from 10 fold cross-validation, while error bars are 1 population standard deviation. 114
6.6 The argument input–output map for the clause `consume_load/4`. 115

7.1 The Blocks World Narrative, illustrating the sequence of events and the subsequent evolution of the state of the world. 119
7.2 Depiction of experimental method. A: Generate time trace data; B: Induce Domain Specific Axioms using Theory Completion. 128
7.3 A photograph of the Khepera® Miniature Mobile Robot (reproduced with permission of K-Team SA, Switzerland). . . . 133
7.4 A schematic of the robot and its "model office" environment. The robot is represented by a circle with a triangle inside. The right angle of the triangle represents the direction the robot is facing. Shaded robots indicate that the robot is in transit. The unshaded robot is stationary. The robot's initial position and destination are shown. The robot at $*$ is enroute to the final destination. The office consists of *rooms* - *R*, *doors* - *D*, and *corners* - *C*. 134
7.5 The experimental procedure for the generation of domain specific axioms by Theory Completion. Note that for the experiments $n = 30$. Step 4 can be refined as shown in figure 7.6. . 145

7.6 A procedure for measuring the variance of the DSA recovery procedure. Note that for the experiments $k = 29$. 145
7.7 A learning curve generated from a single route through the model office. Examples are the number of contiguous time points from which the observed fluents were drawn. The estimates of accuracy are averaged from 30 test sets, while error bars are 1 population standard deviation. 148
7.8 Accuracy versus the number of abducible observations (related to the number of events that act on the robot's trace). Note that two data items appear overlaid at the abscissa value of 16 – 67.81 ± 17.14% and 67.90 ± 16.57%. 149
7.9 The run time of both the extraction-case abduction and the induction versus the number of abducible observations. The experiments were performed on hardware using Intel Pentium III 550MHz processors, and 512 MB RAM. 150

B.1 The theories generated using `CProgol5.0` for the Group I Grammar problems. 179
B.2 The theories generated using `CProgol5.0` for the Group II Grammar problems. 180
B.3 The theories generated using `CProgol5.0` for the Group III Grammar problems. 181
B.4 The theories generated using `ALECTO` for the Group I Grammar problems. 181
B.5 The theories generated using `ALECTO` for the Group II Grammar problems (part 1 of 2). 182
B.6 The theories generated using `ALECTO` for the Group II Grammar problems (part 2 of 2). 183
B.7 The theories generated using `ALECTO` for the Group III Grammar problems. 183

List of Tables

1.1 A comparison of the philosophical modes of reasoning as outlined by Peirce in his *syllogistic theory*, to the mechanistic techniques studied later in the 20th century: *Resolution*, *Absorption*, and *Identification*. 11

2.1 The relationship between different generality orders in ILP. . 32

3.1 A comparison of some of the features of the two techniques for producing negated atoms: Logical Back-propagation, and SOLD-resolution. 70

4.1 A comparison of some of the features of three implementations which can be used for Theory Completion: `ALECTO`, `CProgol5.0`, and `ACL`. 90

5.1 The schedule of grammar experiments, detailing the clauses to be removed from the original theory. 101
5.2 Summary of results of regenerating deleted clauses from the grammar using `ALECTO` and using `CProgol5.0`. 102

6.1 The IDS target clauses for each Theory Completion experiment. 111

7.1 The primitive actions of the robot. These actions depend on the situation the robot is in. 137

8.1 A comparison of the modes of reasoning, as outlined by Peirce, to the different ways in which an Event Calculus program can be used. The Event Calculus components are labelled as follows: Core Axioms – T_{Core}, Narrative of events – $T_{Narrative}$, state of the system at time point i – T_i, domain specific axioms – T_{DSAs}, and the symbol $\Rightarrow$ is read as "producing". Other miscellaneous background theory is not shown. 163

C.1 Learning curve information for generating `consume_load/4` using `ALECTO`. 189

C.2 Learning curve information for generating `consume_place_addr/4` using `ALECTO`. 190
C.3 Learning curve information for generating `consume_null_long/4` using `ALECTO`. 190
C.4 Learning curve information for generating `consume_null_operation/5` using `ALECTO`. 190
C.5 Learning curve information for generating `consume_params/5` using `ALECTO`. 191
C.6 Learning curve information for generating `consume_place_null/5` using `ALECTO`. 191
C.7 Learning curve information for generating `consume_place_addr/4` and `consume_null_long/4` using `ALECTO`. 191

D.1 Accuracies for the 30 induced sets of DSAs for the Robot Navigation Control program using `ALECTO`. The average accuracy across the 30 trials is 70.2%. 205
D.2 Miscellaneous statistics for generating the 30 induced sets of DSAs for the Robot Navigation Control program using `ALECTO`. 206

List of Program Code Snippets

3.1 A SOLD-resolution interpreter to implement Yamamoto's SOLD-resolution in PROLOG. 61
3.2 Multiple *skiplists* produced by the SOLD-interpreter with the query $SOLD(B_{grammar_1}, \overline{sentence([the, president, has, a, cat], [])})$. 62
3.3 Trivial PROLOG code for generating a ground *h-list* from a (potentially) non-ground *skiplist*. 64
3.4 An example of acceptability constraints for `noun_phrase/2` and `verb_phrase/2`. Variables `X` and `Y` are grounded as a side effect of unification. 65
3.5 A bottom clause with head `verb_phrase/2` derived using SOLD-resolution, and the observation $sentence([the, president, has, a, cat], [])$. Note that `[repeated literals]` refers to literals that appear in the bottom clause more than once, and are then removed by `ALEPH` when finalising the bottom clause. 68
3.6 A bottom clause with head `noun_phrase/2` derived using SOLD-resolution, and the observation $sentence([the, president, has, a, cat], [])$. Note that `[repeated literals]` refers to literals that appear in the bottom clause more than once, and are then removed by `ALEPH` when finalising the bottom clause. 68
5.1 The simple context free grammar. 94
5.2 The legal sentences generated by the simple context free grammar. 95
6.1 The *top level* intrusion detection clause for one particular buffer overflow attack strategy. 109
6.2 Two *examples* of malicious byte streams that conform to the intrusion attack strategy in code snippet 6.1. Note that the final few bytes of each example have been written as ASCII characters to convey the intuition of the attacker's intention. 109
6.3 *Target* clauses from the IDS for Theory Completion. 110
7.1 The Event Calculus core axioms (as a PROLOG program). . . 130

7.2 The domain specific axioms to describe the Blocks World in the Event Calculus. . . . 131
7.3 The initial situation and the narrative of events to describe the Blocks World in the Event Calculus. . . . 132
7.4 The Event Calculus domain specific axioms to describe the Robot Navigation Control program. . . . 136
7.5 The robot's initial situation (Taken from Shanahan [Shanahan, 2000]). . . . 137
7.6 The robot's narrative of events – a plan of execution. . . . 137
7.7 The PROLOG encoding of the layout of the "model office" (Taken from [Shanahan, 2000]). . . . 138
7.8 Miscellaneous predicates for the robot (Taken from [Shanahan, 2000]). . . . 138
7.9 Fluent values at different time points as derived from the Robot Navigation Control program. . . . 140
7.10 Completed domain specific axioms produced by ALECTO for the Blocks World. . . . 142
7.11 Sample DSAs generated by ALECTO for the robot navigation control program. . . . 147
B.1 The legal sentences generated by the simple context free grammar. . . . 174
B.2 A sample of random sentences of maximum length 5. . . . 175
B.3 The settings used by CProgol5.0 for the grammar problem. . 175
B.4 The settings used by ALECTO for the grammar problem 176
B.5 Typical CProgol5.0 specific language bias for the grammar problem. . . . 176
B.6 Typical ALECTO specific language bias for the grammar problem. 176
B.7 Common CProgol5.0 and ALECTO *body modes* for the grammar problem. . . . 177
B.8 Common CProgol5.0 and ALECTO typing information for the grammar problem. . . . 177
B.9 CProgol5.0 abductive explanation acceptability constraints for the grammar problem. . . . 177
B.10 ALECTO abductive explanation acceptability constraints for the grammar problem. . . . 178
B.11 Common background clauses for the grammar problem. . . . 178
C.1 Ten *examples* of malicious byte streams that conform to the intrusion attack strategy in code snippet 6.1. . . . 185
C.2 Two *random examples*. . . . 185
C.3 Sample grammar code from the consumer clauses. . . . 186
C.4 Sample code represent a dis-assembler for Intel x86 processors. 187
C.5 Mode declarations and other settings for recovering consume_load/4. 188
C.6 A sample recovered theory recovered clause for consume_load/4. 192
C.7 A sample recovered theory recovered clause for consume_place_addr/4. 192

C.8 A sample recovered theory recovered clause for consume_null_long/4.192
C.9 A sample recovered theory recovered clause for consume_null_operation/5.193
C.10 A sample recovered theory recovered clauses for consume_params/5.193
C.11 A sample recovered theory recovered clause for consume_place_null/5.193
C.12 A sample recovered theory for consume_place_addr/4 and consume_null_long/4. 193
D.1 The Event Calculus core axioms. 194
D.2 The specification of abducible predicates for the Event Calculus in ALECTO. 194
D.3 The specification of acceptability constraints for the Event Calculus abducibles in ALECTO. 195
D.4 The specification of hypothesis structure constraints for the Event Calculus abducibles in ALECTO. 196
D.5 The mode declarations used for learning domain specific axioms to describe the Blocks World. 197
D.6 Initial situation and narrative of events for the Blocks World. 197
D.7 Other Background knowledge for the Blocks World. 198
D.8 Example negative fluent trace data for the Blocks World. . . 198
D.9 Observations of dynamic behaviour for the Blocks World: Positive fluent traces 199
D.10 Domain specific axioms for describing the Blocks World induced using CProgol5.0. 200
D.11 Domain specific axioms for describing the Blocks World induced by Wellner. 200
D.12 The specification of a scoring function for the Event Calculus abducibles in ALECTO. 201
D.13 Initial situation and narrative of events for the Robot Navigation Control program. 201
D.14 Mode declarations for the Robot Navigation Control program. 202
D.15 Example negative fluent trace data for the Robot Navigation Control program. 203
D.16 Observations of dynamic behaviour for the Robot Navigation Control program: Positive fluent traces 204
D.17 Modified Robot Navigation Control DSAs used to generate fluent traces for the experiments. 207

Part I

Theory Completion: theory and implementation

Chapter 1

Introduction

This thesis is concerned with machine assisted theory formation and revision in scientific discovery. The Oxford English Dictionary defines the word **theory** as "*[a] supposition or system of ideas explaining something, especially one based on general principles independent of the particular things to be explained*". P.B. Medawar ([Medawar, 1979] page 1) interprets **science** "*to broadly refer to all exploratory activities of which the purpose is to come to a better understanding of the natural world*". The notion of *scientific discovery* pre-supposes that natural laws exist, and that they are waiting to somehow be discovered. Theories can therefore be considered as *general principles* to *explain particular things* that occur in *the natural world*. How then, are such theories formed? One approach to theory formation is to *generalise* observations of particular things that occur – or by the process of **induction**.

Rarely are scientific theories correct for all particular things that occur, especially for those things not yet seen. Having formed a scientific theory how can it modified to explain new observations? The formation of additional theory based upon new observations is termed **Theory Completion**. A further task relating to scientific discovery is the *revision* of theories. The revision of a theory becomes necessary whenever new observations contradict the predictions of the existing theory. In this case some part of the existing theory must be *corrected*, and possibly additional theory may also need to be added. There is an interplay between the process of completion and correction in the development of scientific theories. In the most extreme case, however, it may be necessary to disband the original theory altogether[1].

[1]This is what Kuhn [Kuhn, 1962] describes as a change in *paradigm*.

1.1 Theory Completion

This work focuses on *Theory Completion*, or the process of augmenting an existing theory to account for new observations. The process requires an existing theory which will often be referred to as the existing background theory, or **background knowledge**. The motivation for performing Theory Completion exists whenever new observations – also referred to as **examples** – occur that are not explained by the existing theory. The output of the Theory Completion process is additional theory, referred to as **hypotheses**, that, when added to the existing theory, explain the new observations. The hypotheses considered in this work are general rules, and not simply specific explanations of specific observations. Theory Completion, however, is only a small part of the general process of theory formation.

1.2 Automated assistants for theory formation

The need to develop automated assistants for theory formation increases as the amount of data recorded electronically continues to grow. In both science and commerce the requirements are to *analyse* and *understand* patterns in data. However, increasingly human analysts are overwhelmed by the quantities of data involved. Techniques that "automatically" *analyse* patterns in data are classed as forms of **Machine Learning**. Machine learning techniques that can also communicate their *understanding* to humans are known as **symbolic** methods.

1.2.1 Symbolic Machine Learning

Definitions for *Machine Learning* vary. This is illustrated by the following examples ([Mitchell, 1997],[Michie, 1988], and [Simon, 1984]).

Definitions of Machine Learning

1. H. Simon, 1984 (page 28)

   > Learning is any change in a system that allows it to perform better the second time on repetition of the same task or on another task drawn from the same population.

2. D. Michie, 1988 (page 107)

   > [A] system [that] uses sample data (training set) to generate an up-dated basis for improved performance on subsequent data[.]

3. T. Mitchell, 1997 (page 2)

> A computer program that improves its performance at some task through experience.

Michie considers that his definition refers to the *Weak Criterion of Machine Learning*. He later defines the *Strong criterion of Machine Learning* as

> [A] system [that] satisfies [the] weak criterion and also can *communicate* its internal updates in explicit **symbolic**[2] form. (Michie 1988 page 108)

Although it is not explicit from the quotation taken from Michie, it is clearly intended that *communicate* refers to one agent being human. In this situation the symbolic machine learning assistant would not only *analyse patterns* in the data but also communicate an *understanding* of what it has learned from the data to a human. Such a situation would be a successful form of machine assisted theory formation in a scientific discovery setting.

1.2.2 Inductive Logic Programming

Inductive Logic Programming (ILP) is a sub-field of symbolic machine learning that aims to generate *Horn clauses*[3] from pre-classified examples. ILP passes Michie's *strong criterion* as it is capable of using sample data to update its basis for performance, and it can communicate its new basis in the *symbolic* form of Horn clauses. Often an ILP system will be presented with some existing set of Horn clauses known as "background knowledge" or "background theory". The background theory is regarded as correct and complete until new examples are observed that are not derivable[4] from the background theory, indicating the need for further learning.

The task of the ILP system is to find a *suitable* set of Horn clauses – known as the *hypothesis* – that when added to the existing background theory explains the new examples. For any given input set – background knowledge and examples – there may be an infinite set of *plausible* hypotheses. In general, there are two main steps in finding a suitable hypothesis; (1) the generation of plausible hypotheses, and (2) the selection of the *best* hypothesis. Approaches vary on how to decide what the best hypothesis is. The best hypothesis may be the one deemed most comprehensible by humans; it may simply be the shortest that predicts the most observations. Chapter 2 presents the theory and methods of inductive logic programming in more detail.

[2] Typefaces have been altered for emphasis.

[3] Horn clauses are the fundamental building blocks of Logic Programs (see chapter 2 for more details).

[4] In ILP a theory is a Logic Program. A logic program that can *derive* an example is taken to be a theory that *explains* or *predicts* the example.

Inductive Logic Programming and Two Way Scientific discovery

ILP techniques have been applied in diverse areas. In numerous applications, (see for example [Muggleton, 1999], [Finn *et al.*, 1998], and [Cussens, 1997]) the Horn clauses – or theories – produced from the observations have been considered as *scientific discoveries*. This statement is corroborated by the number of publications of these discoveries in the journals of the scientific fields that represent the discovery domain (see for example [Cussens *et al.*, 1997; King *et al.*, 1992; King and Srinivasan, 1997; King *et al.*, 1996; King and Srinivasan, 1996; Muggleton *et al.*, 1992; Saith *et al.*, 1998; Srinivasan and King, 1999; Sternberg *et al.*, 1992; Sternberg *et al.*, 1994]).

Although some researchers aim to apply ILP techniques to produce scientific discoveries without the need for human intervention (see [Bryant *et al.*, 1999]), the position of this thesis is that humans (or the scientists) are a key component in the knowledge discovery process. Humans provide the following essential pieces to the discovery process[5].

- The Problem. Without a problem there can be no requirement for learning.
- A way of describing the problem. This includes a level of abstraction – or a language bias – that is considered appropriate to describing the problem, and the observations.
- Some form of partially formed theory, or perhaps old theories that no longer work in light of "surprising" new observations.
- The "surprise" or a set of unexpected observations, which the human desires to be explained.
- Finally, some hunch[6] about what the missing theory might be like.

That a tradesman can be more productive with the correct tools is taken for granted. Why then, should not a scientist be able to use a tool for making himself more productive? Given such a tool, the knowledge discovery process can be seen to be iterative. The scientist provides the tool[7] with the materials discussed above. The machine will then (after some time) produce some "plausible" hypotheses. The scientist can then select from the hypotheses, or perhaps review the input data and ask the machine to try again. Notice how mandatory the ability of the scientist to *comprehend* the machine's hypothesis is – in accordance with Michie's definition of *strong*

[5] A similar point of view is expressed in [Page, 2000].

[6] The term *hunch* is deliberately intended to be vague.

[7] As the tool is likely to be a computer program or the like, it will be referred to as a *machine*.

machine learning (see page 4). This process of human-guided machine-assisted learning will be termed **Two-Way Scientific Discovery**[8] – and is the foundation of the thesis.

1.3 Theory Completion in Inductive Logic Programming

The definitions of observations in a logic program are captured in *predicates*. Predicates are the names of the relationships that make statements about individual things. It is the description of these relationships that make up the collection of knowledge. This collection can be used to make statements about other individuals by providing predictions. Existing ILP systems often assume that the background theory is both correct and complete in all but the definition – or predicate – of the observations. This further assumes that the only completions necessary to the background theory are additional hypotheses in the predicate of the examples.

The form of Theory Completion where the additional theory that is learnt is described in the same predicate as the observation is termed **Observation Predicate Learning** (OPL) in [Muggleton and Bryant, 2000]. This form of Theory Completion is common in both machine learning and ILP. When the definition of the observations differs from the theory to be created then **non-OPL** is required.

Non-OPL Theory Completion[9] requires extensions to the standard OPL techniques. Some of the extensions require a form of **abductive** reasoning[10]. Theory Completion using a technique called **Logical Back-propagation** was introduced in [Moyle and Muggleton, 1997]. More recently it has been described afresh in [Muggleton and Bryant, 2000] where they refer to the implementation, `CProgol5.0` as a *Theory Completion by Inverse Entailment* (TCIE) system.

Theory Completion encompasses a subset of the more general setting of *Theory Revision*. Theory Revision [De Raedt, 1992] attempts to address both incompleteness and incorrectness in a logical theory. De Raedt ([De Raedt, 1992] page 10) describes it as a process of updating a theory T which comprises two sets of clauses T_{fix} and T_{mod}. T_{fix} represents the

[8] The term *Two-way Learning* was first brought to my attention when helping to prepare an ILP Summer School with Stephen Muggleton and Ashwin Srinivasan in 1997. Other people have used alternative terms. Stephen Muggleton in [Muggleton, 1999] uses the term "semi-automated scientific discovery". This does not, however, emphasize the necessary cyclical nature of the human-computer interaction when making discoveries. Mike Sternberg in [Sternberg *et al.*, 1994] uses "joint human-computer collaboration in scientific discoveries" to describe the same process.

[9] In the sequel whenever the term *Theory Completion* is used alone it refers to *Non-OPL Theory Completion*.

[10] *Abductive reasoning* is discussed in more detail in section 1.3.1.

fixed components of the theory which may not be changed by the revising system, whilst attempting to revise T_{mod} to form T'_{mod} such that the resulting theory $T' = T_{fix} \cup T'_{mod}$ does not violate any integrity constraints I, and the evidence E that was previously not derivable from T is derivable from T'. The outcome of Theory Revision is that both: 1) new clauses may be added to a theory, and 2) that "bad" clauses may be removed. In Theory Completion only the former may occur, that is the completeness problem is tackled. MIS [Shapiro, 1983] and CLINT [De Raedt, 1992] are two examples of systems that contain both these operations.

This thesis is motivated by earlier work on learning Event Calculus programs. In this initial work (reported in [Moyle and Muggleton, 1997] and [Moyle, 1999]) limitations in both the theory of Logical Back-propagation and the implementation were encountered. Principally, difficulties existed in 1) learning two predicate definitions simultaneously – known as **multiple predicate learning** (MPL) [De Raedt *et al.*, 1993], and 2) controlling the abductive reasoning provided by Logical Back-propagation. The theory behind Logical Back-propagation, and some of its weaknesses are presented in chapter 3.

This work aims to investigate a more general setting of non-OPL Theory Completion which overcomes some of the previous limitations. In particular the use of **SOLD-resolution** [Yamamoto, 1997] is examined as an alternative to Logical Back-propagation for the basis of a Theory Completion system.

To illustrate the Theory Completion setting consider the following example.

Example 1 (Introductory problem) *Say that one wishes to* discover *a natural language (English) grammar that is capable of parsing simple english* sentences*. An existing theory that describes a* sentence *is: "*A legal sentence is a list of words, where a prefix of the list is a noun phrase, and the remaining suffix is a verb phrase*". Suppose also that there exist definitions for some of the lowest level parsing functions (e.g. those functions for* verb*,* noun*, and* determiner*). Examples of legal sentences have been observed. The partial theory $B_{grammar}$, and the observed sentences $E_{grammar}$, are presented as logic programs in figure 1.1.*

Using the existing partial theory it is hoped that the yet to be defined definitions for noun phrase and verb phrase can be completed by utilising information provided by the observations. With suitably discovered definitions for noun phrase and verb phrase the final theory (i.e. the existing theory augmented with the new definitions) should be capable of parsing the example sentences – and also providing the knowledge contained in these definitions.

$$B_{grammar} = \left\{ \begin{array}{lll} sentence(S,R) & \leftarrow & noun_phrase(S,T), verb_phrase(T,R). \\ determiner([the|R],R) & \leftarrow & \\ determiner([a|R],R) & \leftarrow & \\ noun([president|R],R) & \leftarrow & \\ noun([cat|R],R) & \leftarrow & \\ verb([has|R],R) & \leftarrow & \end{array} \right.$$

$$E_{grammar} = \left\{ \begin{array}{l} sentence([the, president, has, a, cat], []) \leftarrow \\ sentence([the, president, has, the, president], []) \leftarrow \\ sentence([the, president, has, the, cat], []) \leftarrow \\ sentence([the, president, has, a, president], []) \leftarrow \\ sentence([the, cat, has, the, president], []) \leftarrow \\ sentence([the, cat, has, the, cat], []) \leftarrow \\ sentence([the, cat, has, a, president], []) \leftarrow \\ sentence([the, cat, has, a, cat], []) \leftarrow \\ sentence([a, president, has, the, president], []) \leftarrow \\ sentence([a, president, has, the, cat], []) \leftarrow \\ sentence([a, president, has, a, president], []) \leftarrow \\ sentence([a, president, has, a, cat], []) \leftarrow \\ sentence([a, cat, has, the, president], []) \leftarrow \\ sentence([a, cat, has, the, cat], []) \leftarrow \\ sentence([a, cat, has, a, president], []) \leftarrow \\ sentence([a, cat, has, a, cat], []) \leftarrow \end{array} \right.$$

Figure 1.1: The partially complete grammar theory for parsing sentences, $B_{grammar}$, and the observations $E_{grammar}$ that the completed grammar should be able to explain as *parsable*.

$$H_{grammar} = \left\{ \begin{array}{lll} noun_phrase(S,T) & \leftarrow & determiner(S,N), noun(N,T) \\ verb_phrase(T,R) & \leftarrow & verb(T,N), noun_phrase(N,R) \end{array} \right.$$

Figure 1.2: The completed definitions for *verb_phrase* and *noun_phrase*. The complete grammar parser consisting of the original unfinished program with the additional definitions $B_{grammar} \cup H_{grammar}$, will explain – or parse – all observations, $E_{grammar}$.

The Theory Completion techniques developed in this thesis are tested on three application domains. First the Theory Completion of the simple context free grammar above is explored, then it is followed by a context dependent grammar used for intrusion detection. The third test domain is learning Event Calculus programs. The most complicated Theory Completion task considered is a problem arising from robot navigation control. Here the description of a program to control the navigation of a real-life robot is completed automatically from the dynamic observations of a simulated robot.

1.3.1 Theory Completion in ILP and the relation to Scientific Method

Although it is generally acknowledged[11] that there is no single "method of doing science" there are parallels between Theory Completion techniques used in ILP with those purported in the use of scientific method and reasoning.

Scientists often concern themselves with the task of discovering novel *laws of nature* or *hypotheses*. Although the *reasoning* with such discovered hypotheses has been placed on the rigorous footings of *logical deduction*, the reasoning applied to the formation of *hypotheses* in the first-place has been the concern of philosophers of science. The form of reasoning with hypotheses – *deduction* – is non-defeasible (or sound), while the reasoning required to form hypotheses is defeasible. There has been some debate [Flach and Kakas, 2000] whether the defeasible reasoning – or non-deductive logic – for the formation of hypotheses is a single cogent category (known as *induction*); or whether it is further subdivided into two forms: *induction* and *abduction*. For this work, in common with abduction in logic programming and artificial intelligence, the second view, that of both *induction* and *abduction*, is taken.

Induction

Theory Completion in ILP utilises forms of induction. Induction has also received attention from philosophers of science. Bacon, Whewell ([Whewell, 1847] via [Medawar, 1982]), Mill, Jevons, Peirce, and others wrote about induction[12]. Bacon, an early advocate of induction from observations said ([Bacon, 1994] page 20) "... in regard to the nature of things, I use induction throughout ...". Mill [Mill, 1872] suggests that "The business of inductive

[11] Peter Medawar in [Medawar, 1982] (page 116) says: "There is no such thing as The Scientific Method".

[12] More recently Gillies has promoted machine-aided scientific discoveries as *evidence* for the success of inductivist methods [Gillies, 1996].

logic is to provide rules and models ...". Jevons writes [Jevons, 1877] (page 123)

> Induction ... [is] ... the inference of general [laws] from particular truths [or facts].

More importantly for Theory Completion in ILP, he goes on to write

> Induction is ... the inverse operation of deduction

For this work *induction* is taken to be *inductive generalisation* ([Plotkin, 1969], [Flach and Kakas, 2000]) and seen as the process of hypothesis *generation*, as opposed to the process of hypothesis *evaluation*.

Abduction

The introduction of the term Abduction is widely attributed to the 19th/20th century philosopher Charles S. Peirce. There is some confusion (see [Aliseda, 2000]) – including by Peirce himself – as to what he originally meant by the term abduction[13]. For Peirce, eventually, abduction was the process of forming *new* hypotheses – motivated by the observation of a *surprise* result [Kruijff, 1997]. This process is described by Peirce as "human intuition", and "magic". Similarly, Whewell called good hypotheses "happy guesses". Abduction encompasses the fact that humans typically propose a few hypotheses that are "most likely to contain" a correct one – even though the total number of possible hypotheses is very large – or even infinite.

Peirce described two alternative notions of abductive reasoning. In his earlier work he provided his *syllogistic theory*, while later he produced his *inferential theory* in which abduction represents the hypothesis generation part of explanatory reasoning. This work utilises the former, syllogistic approach, which has been interpreted by some researchers as a form of inverted *modus ponens* (see [Aliseda, 2000]). This seems to be the view taken more recently by researchers in Artificial Intelligence (AI) and abductive logic programming [Flach and Kakas, 2000]. The AI researchers, have settled on the distinctions between three particular forms of reasoning: Deduction, Induction, and Abduction.

Peirce's *syllogistic theory*: three forms of reasoning

Peirce's *syllogistic theory* ([Peirce, 1931], [Kakas *et al.*, 1993], [Flach and Kakas, 2000]) distinguishes the following three forms of reasoning.

Deduction an analytical[14] process based on the application of general rules to particular cases, with the inference of a result.

[13] Peirce himself claims to have confused the terms induction and abduction (and had pseudonyms for abduction of hypotheses and retroduction)

[14] Peirce differentiates between "analytical" and "synthetic" forms of reasoning, which would now be described as "sound" and "unsound" reasoning.

Table 1.1: A comparison of the philosophical modes of reasoning as outlined by Peirce in his *syllogistic theory*, to the mechanistic techniques studied later in the 20th century: *Resolution*, *Absorption*, and *Identification*.

Philosophical Modes of Reasoning		*Mechanistic reasoning techniques*	
Deduction	$rule(s)$ $+case$ $\overline{result}$	$case \quad rule$ $\searrow \quad \swarrow$ $result$	Resolution
Induction	$case(s)$ $+result(s)$ $\overline{rule(s)}$	$case \rightarrow rule$ $\nearrow$ $result$	Absorption
Abduction	$rule(s)$ $+result(s)$ $\overline{case(s)}$	$case \leftarrow rule$ $\nwarrow$ $result$	Identification

Induction synthetic reasoning which infers the rule from the case and the result.

Abduction another form of synthetic inference, but of the case from a rule and a result.

Table 1.1 presents similarities between research efforts in "inverting" deductive procedures and those reasoning techniques outlined by Peirce. Some early efforts to formalise induction by Plotkin [Plotkin, 1969] focused on inverting deduction, which coincidentally had been suggested by Jevons (see previous page). The deductive method used was that of *resolution* [Robinson, 1965]. The *absorption* operator with similarities to induction was suggested by [Sammut and Banerji, 1984]. Muggleton [Muggleton, 1987] proposed the *identification* operator in a propositional setting that is similar to abduction (in the AI sense).

Example 2 (Deduction, Induction, and Abduction illustrated) *Consider the following components taken from the earlier grammar example.*

$$Rule = \{ \quad sentence(S,R) \quad \leftarrow \quad noun_phrase(S,T), verb_phrase(T,R).$$

$$Result(s) = \left\{ \begin{array}{l} e_1 = sentence([the, president, has, a, cat], []) \leftarrow \\ e_2 = sentence([a, cat, has, the, president], []) \leftarrow \end{array} \right.$$

Case(s), Δ_1 *and* Δ_2.

$$\Delta_1 = \left\{ \begin{array}{l} noun_phrase([the, president, has, a, cat], [has, a, cat]) \leftarrow \\ verb_phrase([has, a, cat], []) \leftarrow \end{array} \right.$$

$$\Delta_2 = \left\{ \begin{array}{l} noun_phrase([a, cat, has, the, president], [has, the, president]) \leftarrow \\ verb_phrase([has, the, president], []) \leftarrow \end{array} \right.$$

Deduction *can be used to derive the Result(s) from the Rule and the Case(s). That is*

$$\frac{\begin{array}{r} sentence(S, R) \leftarrow noun_phrase(S, T), verb_phrase(T, R). \\ +noun_phrase([the, president, has, a, cat], [has, a, cat]) \leftarrow \\ +verb_phrase([has, a, cat], []) \leftarrow \end{array}}{sentence([the, president, has, a, cat], []) \leftarrow}$$

and

$$\frac{\begin{array}{r} sentence(S, R) \leftarrow noun_phrase(S, T), verb_phrase(T, R). \\ +noun_phrase([a, cat, has, the, president], [has, the, president]) \leftarrow \\ +verb_phrase([has, the, president], []) \leftarrow \end{array}}{sentence([a, cat, has, the, president], []) \leftarrow}$$

The process of **Induction** *can be used to propose a Rule from the Result(s) and the Case(s)*[15]*. That is*

$$\frac{\begin{array}{r} sentence([the, president, has, a, cat], []) \leftarrow \\ sentence([a, cat, has, the, president], []) \leftarrow \\ +noun_phrase([the, president, has, a, cat], [has, a, cat]) \leftarrow \\ +verb_phrase([has, a, cat], []) \leftarrow \\ +noun_phrase([a, cat, has, the, president], [has, the, president]) \leftarrow \\ +verb_phrase([has, the, president], []) \leftarrow \end{array}}{sentence(S, R) \leftarrow noun_phrase(S, T), verb_phrase(T, R).}$$

Finally, **Abduction** *can be used to suggest possible Case(s) from Rule(s) and Result(s). That is*

$$\frac{\begin{array}{r} sentence(S, R) \leftarrow noun_phrase(S, T), verb_phrase(T, R). \\ +sentence([the, president, has, a, cat], []) \leftarrow \end{array}}{\begin{array}{r} noun_phrase([the, president, has, a, cat], [has, a, cat]) \leftarrow \\ verb_phrase([has, a, cat], []) \leftarrow \end{array}}$$

and

$$\frac{\begin{array}{r} sentence(S, R) \leftarrow noun_phrase(S, T), verb_phrase(T, R). \\ +sentence([a, cat, has, the, president], []) \leftarrow \end{array}}{\begin{array}{r} +noun_phrase([a, cat, has, the, president], [has, the, president]) \leftarrow \\ +verb_phrase([has, the, president], []) \leftarrow \end{array}}$$

The methods used in Theory Completion utilise all three of the above forms of reasoning. Deduction is built into the logic programming [Lloyd, 1993] language PROLOG [Clocksin and Mellish, 1987]. Methods for performing induction are detailed in chapter 2, while elementary abductive reasoning mechanisms are described in chapter 3.

[15] At least two $\langle result, case \rangle$ pairs are required to generate a *generalised* rule.

1.4 Related work

Theory Completion in ILP develops theoretical components common to both Theory Recovery and *Abductive Logic Programming*. The most complicated application of Theory Completion considered in this thesis is the learning of Event Calculus programs. Work relating to learning programs that describe action change are therefore related, as are applications of ILP to temporal problems.

Theory Revision and Theory Recovery

The general setting of Theory Completion and its relationship to Theory Revision has already been mentioned in section 1.3. The setting of Theory Recovery is described as "*the process of adjusting, or completing background knowledge*" ([Parson *et al.*, 1999] page 257). This description is similar to that of Theory Completion. In [Parson *et al.*, 1999] the system `CProgol5.0` is used to recover definitions of predicates deleted from a synthetic example. In these experiments only the definitions of a *single* predicate required recovery in any one situation.

Recently Muggleton and Bryant have applied `CProgol5.0` in two domains [Muggleton and Bryant, 2000]: 1) learning components of a grammar; and 2) learning gene–enzyme code descriptions deleted from a functional genomics model. In these experiments the number of deleted definitions from the original programs were varied and the recovery performance measured.

This thesis also uses a grammar problem to investigate alternative Theory Completion techniques. The setting for these experiments differs from those in both [Parson *et al.*, 1999] and [Muggleton and Bryant, 2000]. The experiments in Theory Recovery only consider the setting where a *single* predicate description is required, while the work here considers multiple-predicate learning. The Theory Completion experiments in [Muggleton and Bryant, 2000] consider the setting where *some* definitions are present in the existing theory, while the work here focuses on the case where *no* definitions are in the existing theory.

Abductive Logic Programming

A large body of research on abductive techniques in logic programming is available (see for example [Denecker and Kakas, 2000]). Some mechanisms used in Theory Revision systems are based on abductive operators. There is a close relationship between Inductive Logic Programming and Abductive Logic Programming as has been noted by Dimopoulos and Kakas in [Dimopoulos and Kakas, 1996] where they describe generating hypotheses H from a background theory T, and observations O such that the combination of hypotheses and background theory explain the observations. They

distinguish the main difference between abduction and induction as follows ([Dimopoulos and Kakas, 1996] page 28).

> Whilst in *abduction* the addition H to T just adds missing specific facts related to the particular observations in *induction* H introduces new general relations in T[16].

The Theory Completion techniques developed in this thesis are similar to a particular incorporation of abduction in learning called the **extraction-case** (see [Dimopoulos and Kakas, 1996] page 160). The techniques also share methods common to those used in the system `ACL`, developed by Kakas and Raguzzi [Kakas and Riguzzi, 2000]. More detail about these Theory Completion techniques can be found in chapters 3 and 4.

Learning Programs about Action and Change

The Event Calculus is a logic programming framework for reasoning about actions and change. Its simple axioms encode the common sense law of inertia. It consists of the following components: the core axioms (which provide the law of inertia), the domain specific axioms (which are provided by a programmer), an initial state of the system, and a sequence of events. From these it is possible to reason about the state of the system at different times.

The Event Calculus, as a reasoning framework, can be utilised in three alternative ways. Event calculus programs can be used *deductively* to predict how the state of the program evolves over time. Also work has shown (e.g. [Shanahan, 2000]) how an event calculus program can be used *abductively* to plan the sequence of events necessary to change the system from an initial state into a goal state. The work reported in this paper focuses on a third, *inductive* way of using and event calculus program. Here the objective is to induce the domain specific axioms – the situation-action rules – from observations of the evolving states of a system.

One of the application domains investigated in this thesis is the learning of Event Calculus programs. This expands on the work reported in [Moyle and Muggleton, 1997]. Ben Wellner [Wellner, 1999] in his work on *Logic based agents* also uses the Event Calculus to learn action theories on a similar problem to [Moyle and Muggleton, 1997] and obtains similar results. He too uses techniques from Abductive Logic Programming (see chapter 4).

Lorenzo and Otero [Lorenzo and Otero, 1999] [Lorenzo and Otero, 2000] use the Situation Calculus, which is an alternative action formalism to the Event Calculus, to induce logical theories about action and change. Their work focuses on learning *indirect effects* of actions. The work here focuses on learning rules for *direct effects* of actions (see chapter 7). They propose

[16] Italics added for emphasis.

the development of their own learning system, based on ILP and Inverse Entailment [Muggleton, 1995] notions.

Research in applying ILP to learning *precondition rules* in the Event Calculus has been performed by [Sablon, 1995]. This work uses ILP in a planning system that performs some learning. The ILP approach taken in the work is based on *Iterative Version-spaces* [De Raedt, 1992; Mitchell, 1982] rather than Inverse Entailment, which is used here. Inverse Entailment, one basis for ILP, is described in chapter 2.

Applications of Inductive Logic Programming in temporal domains

Previous researchers have studied ILP in temporal domains. Bratko and colleagues [Bratko *et al.*, 1992], [Bratko, 1994] describe the application of inducing qualitative models [Kuipers, 1986] of dynamic systems from example behaviours. The dynamic system studied was that of a U-Tube containing fluid. A qualitative model was also used as background knowledge to learn temporal diagnostic rules for diagnosing satellite faults [Feng, 1992].

Sammut and Hume have studied the learning of action sequences in a simulated robot world [Hume and Sammut, 1992]. An alternative approach has focused in the area of *behavioural cloning*, referring to the capturing of a human skill from examples of their behaviour. Propositional learning techniques have been applied in the study of behavioural cloning (e.g. [Bain and Sammut, 1999] have learned flight controllers). ILP systems have been applied to the problem of learning the rules for turning an aircraft from simulated F16 flight data [Srinivasan and Camacho, 1999], and also to learning traffic flow rules [Eastmond, 1996].

1.5 Thesis outline

The contribution of this thesis is the study of Theory Completion techniques in Inductive Logic Programming. General extensions to existing ILP techniques are provided for the use in applications where the concepts (or theories) to be learned are not in the same predicate (or language) as the available observations. A comparison of two techniques is performed. An implementation based on SOLD-resolution, called `ALECTO`, is developed, which is then applied in two application domains. In the first application, clauses for a simple context free grammar are generated using a partial grammar and examples of legal sentences. The second application demonstrates how the complete navigation control program of an actual robot may be generated from observations of the robot's route as it negotiates its way through a multi-room office.

The thesis is presented in three parts: Part I – *Theory Completion: theory and implementation*; Part II – *Applications of* `ALECTO`; and Part III

– *Discussion and concluding remarks.*

Part I is organised as follows. The machine induction of first order theories by ILP is introduced in chapter 2. Chapter 3 develops the techniques of theory completion as an extension of ILP. This includes the theory of two methods for theory completion: 1) Logical Back-propagation; and 2) SOLD-resolution. The implementation of SOLD-resolution theory completion techniques in the system `ALECTO` are then outlined in chapter 4.

Part II contains three applications of `ALECTO` in a theory completion setting. The first is that of completing a simple natural language grammar (chapter 5, which is also used in the sequel to illuminate the theoretical development. The second is that of completing a semantic intrusion detection system (chapter 6). The final application of `ALECTO` is that of completing Event Calculus programs (chapter 7).

Part III contains the final chapter which presents a discussion and some concluding remarks.

Chapter 2

Machine Induction of First Order Theories by Inductive Logic Programming

This chapter introduces the theory and techniques of Inductive Logic Programming (ILP), and Theory Completion. The main ILP procedures involve: (1) the generation of hypotheses from examples relative to an existing background theory, (2) the refinement and search for improved hypotheses, and (3) the testing of proposed hypotheses. Many differing ILP algorithms and systems have been developed, but the techniques promoted here are those from the Progol family of algorithms [Muggleton, 1995]. The Progol algorithm is based on the theory of Inverse entailment. To illustrate the techniques a common example based on a PROLOG program for a simple context free grammar is used throughout the chapter.

2.1 The general setting for ILP

This section presents the general setting for ILP. In the definitions that follow the term *clause* refers to any well-formed logical formula. It is common, however, within ILP to use only Horn clauses.

Definition 3 (Background Theory) *B is defined as the union of (possibly zero) clauses. $B = C_1 \cup \ldots \cup C_n$. The background theory represents the collection of prior knowledge, before any induction has been performed.*

In ILP the background theory is commonly *believed* to be correct, but as the examples are not explained by it, it is believed to be incomplete.

Example 4 (Grammar (part 1)) *The following definite clauses*[1] *represent relationships between words in a simple Natural Language Processing*

[1] All variables in the formulae are implicitly universally quantified. All predicate, func-

(NLP[2]) grammar (taken from [Van Le, 1993] p479)[3].

$$B_{grammar_0} = \begin{cases} sentence(S,R) & \leftarrow & noun_phrase(S,T), verb_phrase(T,R). \\ noun_phrase(S,T) & \leftarrow & determiner(S,N), noun(N,T). \\ determiner([the|R],R) & \leftarrow & \\ determiner([a|R],R) & \leftarrow & \\ noun([president|R],R) & \leftarrow & \\ noun([cat|R],R) & \leftarrow & \\ verb([has|R],R) & \leftarrow & \end{cases}$$

Definition 5 (Examples for induction) *The observations, or examples E to be used for induction, are also a conjunction of clauses. The examples can be partitioned into two sets $\langle E^+, E^- \rangle$, where E^+ represents the positive examples, and E^- the negative examples. The examples typically satisfy the following equations.*

$$E = E^+ \cup E^- \tag{2.1}$$

$$E^+ \cap E^- = \emptyset \tag{2.2}$$

In this work the terms *examples* and *observations* will be used interchangeably[4].

Example 6 (Grammar (part 2)) *Suppose that the beliefs defined by $B_{grammar_0}$ are assumed correct, and that* "the president has a cat" *and* "a cat has the president" *are both examples of legal*[5] sentences.

$$E^+_{grammar_0} = \begin{cases} sentence([the,\ president,\ has,\ a,\ cat]\ ,[]) & \leftarrow \\ sentence([a,\ cat,\ has,\ the,\ president]\ ,[]) & \leftarrow \end{cases}$$

The above facts can be considered as positive examples for the predicate sentence/2 [6]*. Notice that neither of the positive examples logically follow*

tion, and constant symbols begin with a lower case character. Variables begin with an upper case character.

[2]Natural Language Processing is concerned with techniques for computer understanding, analysis, manipulation, and/or generation of natural language.

[3]The procedural meaning of the predicates in the grammar is that the first argument is a list of words, and the second argument is a suffix of the first list of words, the prefix part having been *consumed* by the predicate. If the second argument is the empty list of words (`[]`), then the first argument has the relation of the predicate (e.g. `noun([president],[])` means that the word *president* is a *noun*).

[4]Strictly, *examples* are appropriate in the situation where the clauses to be induced have the same predicate symbol as the examples (this is termed Observational Predicate Learning – see definition 8); while *observations* refer to the setting where the target clauses have *different* predicate symbols to those of the observations.

[5]Here *legal* disregards the normal notions of punctuation and capitalisation.

[6]The number associated with each predicate indicates its arity.

from the beliefs held in $B_{grammar_0}$. *Furthermore, the following facts come to light.* "the the president" *is not a sentence, nor are,* "the the a president president" *or* "the has has the president".

These negative examples can be expressed as the following clauses[7]*:*

$$E^-_{grammar_0} = \begin{cases} \textit{sentence([the, the, president],[]).} \\ \textit{sentence([the, the, a, president, president],[]).} \\ \textit{sentence([the, has, has, the, president],[]).} \end{cases}$$

Notice that equations 2.1 and 2.2 hold for $E_{grammar_0}$.

Definition 7 (Hypothesis) *H is defined as the union of (possibly zero) clauses. $H = H_1 \cup \ldots \cup H_n$. The hypothesis is simply a set of clauses.*

In ILP it is usual that the hypotheses considered are a direct generalisation of the examples. This is the case where predicate symbol of the hypotheses is the same as that of the examples. The following definition formalises this notion.

Definition 8 (Observational Predicate Learning) *(from [Muggleton and Bryant, 2000]) The setting where Horn clauses to be induced H have the same predicate symbol as the examples E is termed* Observational Predicate Learning *(OPL). That is $\forall H_i \in H \exists E_j \in E \, s.t. \, predicate(H_i) = predicate(E_j)$ where $predicate(C)$ returns the predicate symbol of the positive literal in the Horn clause C, and E_j is considered as a unit (Horn) clause.*

2.1.1 The logical conditions for ILP

The general setting for ILP is *given*: prior background knowledge B, and examples E consisting of positive and negative examples; *find* an hypothesis H such that the four conditions below hold (from [Muggleton and de Raedt, 1994]).

ILP condition 1 - Prior Satisfiability $B \cup E^- \not\models \square$. This condition states that the negative examples are not inconsistent with the background knowledge. In the case that B and E are restricted to definite clauses, this condition becomes $E^- \cap \mathcal{M}^+(B) = \emptyset$, where $\mathcal{M}^+(B)$ is the minimum Herbrand model[8] of B.

[7] Some ILP systems (e.g. `CProgol5.0`) encode such negative examples in the form `:-sentence([the, the, president],[]).` which is to be read as "it is false if this negative example can be proved".

[8] For details of the semantics of logic programs see for example [Lloyd, 1993].

Example 9 (Grammar (part 3)) *None of the three negative examples are inconsistent with the background knowledge, $B_{grammar_0}$. The minimum Herbrand model for $B_{grammar_0}$ is*[9]*:*

$$\mathcal{M}^+(B_{grammar_0}) = \left\{ \begin{array}{ll} \textit{determiner([the],[])} & \textit{determiner([a],[])} \\ \textit{noun([president],[])} & \textit{noun([cat],[])} \\ \textit{verb([has],[])} & \\ & \\ \textit{noun_phrase([a,cat],[])} & \textit{noun_phrase([the,president],[])} \\ \textit{noun_phrase([the,cat],[])} & \textit{noun_phrase([a,president],[])} \\ & \\ \multicolumn{2}{l}{\textit{determiner([the,president,has,a,cat],[president,has,a,cat])}} \\ \multicolumn{2}{l}{\textit{determiner([a,cat,has,the,president],[has,the,president])}} \\ \multicolumn{2}{l}{\textit{determiner([the,president],[president])}} \\ \multicolumn{2}{l}{\textit{determiner([a,cat],[cat])}} \\ \multicolumn{2}{l}{\vdots} \\ & \\ \multicolumn{2}{l}{\textit{noun_phrase([the, president, has, a, cat] ,[has, a, cat])}} \\ \multicolumn{2}{l}{\textit{noun_phrase([a, cat, has, the, president] ,[has, the, president])}} \end{array} \right\}$$

All negative examples are false in $\mathcal{M}^+(B_{grammar_0})$.

ILP condition 2 - Posterior Satisfiability $B \cup H \cup E^- \not\models \square$. This condition ensures that any hypothesis that is finally adopted is *consistent* with the negative examples. When restricted to definite clauses this condition becomes $E^- \cap \mathcal{M}^+(B \cup H) = \emptyset$.

ILP condition 3 - Prior Necessity $B \not\models E^+$. This condition is the main *motivation* for induction. It states that some positive examples are false with respect to the background theory. In the case of definite clauses this condition is $E^+ \not\subseteq \mathcal{M}^+(B)$.

Example 10 (Grammar (part 4)) *Neither of the following two positive examples*

$$E^+_{grammar_0} = \left\{ \begin{array}{l} sentence([the, president, has, a, cat], []), \\ sentence([a, cat, has, the, president], []) \end{array} \right\}$$

are in the minimum model of the background knowledge and are therefore false *in $\mathcal{M}^+(B_{grammar_0})$.*

ILP condition 4 - Posterior Sufficiency $B \cup H \models E^+$. This is the *objective* of the inductive process, to produce hypotheses that are *complete* with respect to the positive examples. When restricted to definite clauses this condition equates to $E^+ \subseteq \mathcal{M}^+(B \cup H)$.

[9]For simplicity only lists of words of length six or fewer are considered. Also, it is assumed that the empty list `[]` is in the Herbrand Universe of $B_{grammar_0}$, as is the list constructor functor.

Example 11 *[Grammar (part 5)] Provided with the four conditions above the following definite clause hypothesis may be proposed.*

$$H_{sentence} = sentence(S, R) \leftarrow noun_phrase(S, V), verb(V, N), noun_phrase(N, R).$$

Hypothesis $H_{sentence}$ satisfies ILP conditions 2 and 4. A second plausible hypothesis is H_{verb_phrase}.

$$H_{verb_phrase} = verb_phrase(T, R) \leftarrow verb(T, N), noun_phrase(N, R).$$

A definition for sentence/2 already exists in $B_{grammar_0}$, whereas there is no pre-existing definition for verb_phrase/2. Notice also that the hypothesis $H_{sentence}$ is a definition of the concept sentence*, and that the starting example was an instance of* sentence *– this is the* OPL *setting. Compare this to H_{verb_phrase} which is a definition for a concept that was not part of the example set – or the* non-OPL *setting (see definition 8 on page 19). This is an important distinction in the discussion of Theory Completion techniques for ILP (more details are presented in chapter 3).*

2.1.2 Different ILP settings

The ILP setting that has been presented so far is that known as *explanatory ILP*. An alternative style of ILP is that of *descriptive ILP*. This section provides a brief introduction and comparison of both styles.

Explanatory ILP

In the preceding text, whenever the term *ILP* has been used it has, in fact, referred to *Explanatory ILP*[10]. The core feature of Explanatory ILP is the fundamental relationship described in ILP Condition 4 (on page 20) which requires that the set of hypotheses, together with the existing background theory B, entail each and every positive example E^+,(and none of the negative examples E^-). That is $\forall e \in E^+ : B \cup H \models e$, and $\forall e \in E^- : B \cup H \not\models e$. In this setting it is said that H (combined with B) *explains* the examples E^+.

Descriptive ILP

Descriptive ILP (from [Blockeel, 1998] page 84) is the ILP setting where there is a set of hypotheses H for which each positive example makes each hypothesis in the set true, while none of the negative examples make any of the hypotheses true. Each hypothesis is said to *describe* each of the positive examples. The following conditions formalise this notion.

[10]This is also known as *normal* ILP (see page 168 of [Nienhuys-Cheng and de Wolf, 1997]).

Descriptive ILP condition 1

$$\{\forall e \in E^+ \wedge \forall h \in H : h \text{ is } \textit{true} \text{ in } \mathcal{M}^+(e \cup B)\}$$

Here each positive example makes each hypothesis in the set of hypotheses true.

Descriptive ILP condition 2

$$\{\forall e \in E^- \wedge \forall h \in H : h \text{ is } \textit{false} \text{ in } \mathcal{M}^+(e \cup B)\}$$

It is required that none of the negative examples make any of the hypotheses true.

Descriptive ILP is also known as *non-monotonic learning* and *learning from interpretations*. In this setting, ILP has similar objectives to those in the field of *Data Mining* (see for example [Fayyad *et al.*, 1996]). Descriptive ILP has been shown to be efficient, and in some circumstances, it is possible to PAC-learn a clausal theory (see [De Raedt and Dzeroski, 1994]). It is, however, limited to learning non-recursive theories.

Notice, however, that descriptive ILP does not provide any "clues" how such sets of hypotheses may be generated. Contrast this with ILP Condition 4 (shown on page 20) which is the basis for hypothesis generation using Inverse Entailment (as described in the next section). For this thesis, the *Explanatory ILP* setting will be used, particularly as recursive theories are important in Theory Completion.

2.2 Hypothesis generation

It is now possible to consider the generation of hypotheses from examples relative to an existing background theory. Early forms of hypothesis generation in ILP studied the process of inverting the deductive rule of inference, *resolution* [Muggleton, 1992; Plotkin, 1971; Robinson, 1965], thus treating induction as a form of *reverse* deduction. An alternative approach called Inverse Entailment [Muggleton, 1995], introduces techniques for reversing the logical consequence relation, based on a model theoretic approach. The method for hypothesis generation that will be in this work is Inverse Entailment.

2.2.1 Inverse Entailment

The condition of Posterior Sufficiency (ILP condition 4) provides the basis for generating an initial, single clause[11] hypothesis H, from a single positive

[11]Here H refers to a *single clause* hypothesis and not a set of clauses as mentioned previously.

example $e_i^+ \in E^+$, and background knowledge B. The negation of such an hypothesis can be derived by applying the deduction theorem [Hamilton, 1988] twice

(2.3)	$B \cup H \models e^+$	Posterior Sufficiency
(2.4)	$B \models H \rightarrow e^+$	by Deduction theorem
(2.5)	$B \models \overline{e^+} \rightarrow \overline{H}$	by Contra-position
(2.6)	$B \cup \overline{e^+} \models \overline{H}$	by Deduction theorem

If B and e_i^+ are restricted to definite clauses, then $\overline{H}$ is some (possibly infinite) conjunction of logically entailed (ground) literals. Such literals may be derived by linear resolution [Lloyd, 1993].

Example 12 (Grammar (part 6)) *Given the first positive example from* $E^+_{grammar_0}$ *(i.e.* $e_1^+ = sentence([the, president, has, a, cat], []) \leftarrow$*), then the conjunction of ground literals derivable from linear resolution is*

$$\begin{aligned}\overline{H_{(e_1^+, B_{grammar_0})}} \quad = \quad & noun_phrase([the, president], []) \wedge determiner([the], []) \wedge \\ & noun([president], []) \wedge \overline{verb_phrase([has, a, cat], [])} \wedge \\ & determiner([the, president], [president]) \wedge \ldots \wedge \\ & verb([has], []) \wedge noun_phrase([a, cat], []) \wedge \\ & determiner([a], []) \wedge noun([cat], []) \wedge \\ & \overline{sentence([the, president, has, a, cat], [])}\end{aligned}$$

The positive literals are ground facts contained in the minimum model of the background knowledge. The last literal is simply the negation of the positive example as derived in equation 2.6. The derivation of the other negative literal, $\overline{verb_phrase([has, a, cat], [])}$*, is illustrated in figure 2.1. Notice that this literal cannot be derived by SLD-resolution which is employed by* Prolog *systems [Stickel, 1992] (see chapter 3 for further details).*

By negating the above conjunction of literals, the following (non definite) ground clause is obtained.

$$\begin{aligned}& H_{(e_1^+, B_{grammar_0})} = \\ & verb_phrase([has, a, cat], []) \vee sentence([the, president, has, a, cat], []) \leftarrow \\ & \quad noun_phrase([the, president], []), verb([has], []), noun([president], []), \\ & \quad determiner([the], []), noun_phrase([a, cat], []), determiner([a], []), \\ & \quad noun([cat], []).\end{aligned}$$

The continuation of the Grammar example below considers the relationship between $H_{(e_1^+, B_{grammar_0})}$ and the previously proposed hypotheses, $H_{sentence}$ and H_{verb_phrase}.

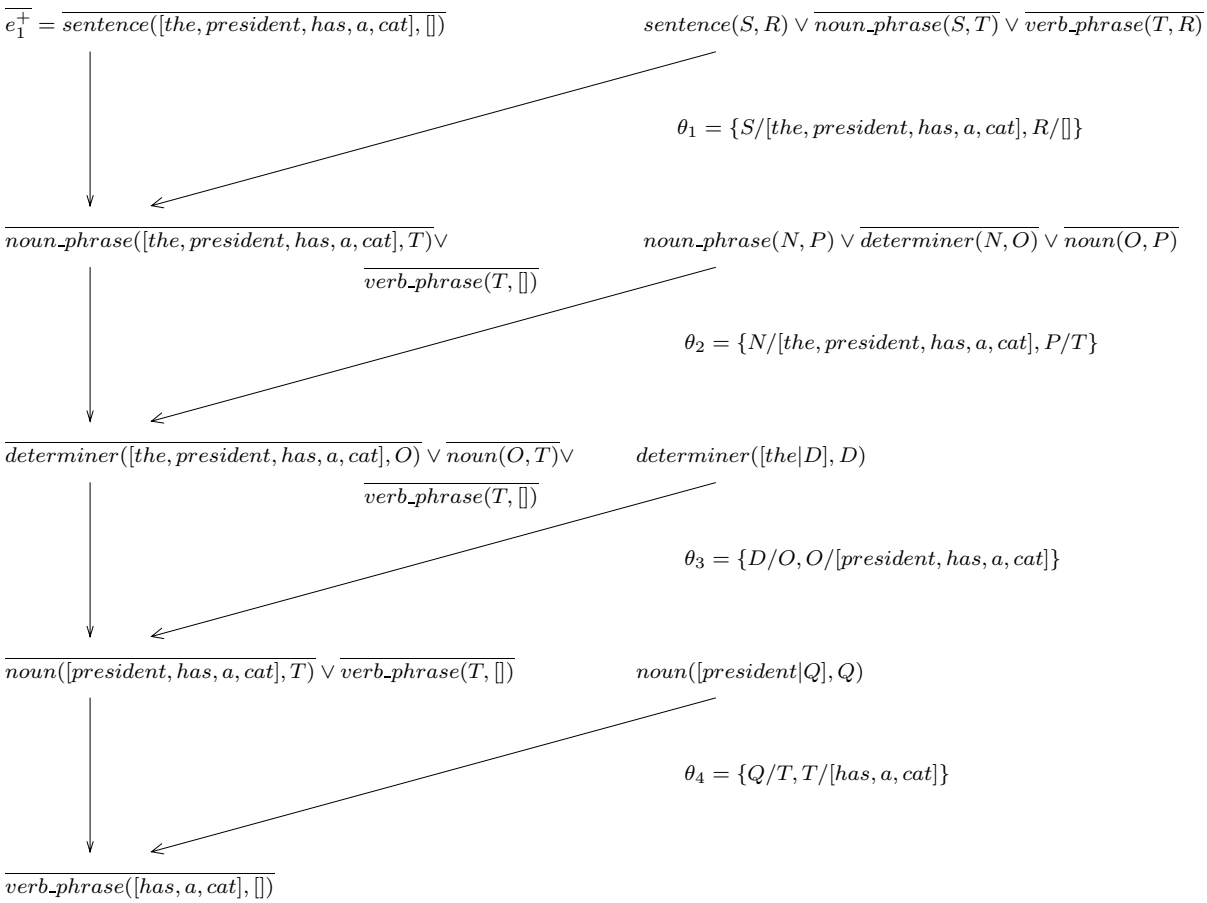


Figure 2.1: Derivation of $\overline{verb_phrase([has, a, cat], [])}$ by linear resolution from $B_{grammar_0}$ and $\overline{e_1^+}$.

Example 13 (Grammar (part 7)) *Together with the background knowledge, each of the proposed hypotheses logically entail the example.*

$$\begin{aligned} B_{grammar_0} \cup H_{sentence} &\models e_1^+ \\ B_{grammar_0} \cup H_{verb_phrase} &\models e_1^+ \end{aligned}$$

Notice also that both hypotheses logically entail the second example – that is the hypotheses are complete*; and do not entail either of the negative examples – that is the hypotheses are* correct*.*

It is also the case that both hypotheses logically entail $H_{(e_1^+, B_{grammar_0})}$*. That is:*

$$H_{sentence} \models H_{(e_1^+, B_{grammar_0})} \tag{2.7}$$

$$H_{verb_phrase} \models H_{(e_1^+, B_{grammar_0})} \tag{2.8}$$

In the example above $\overline{H_{(e_1^+, B_{grammar_0})}}$ consisted of *all* ground literals that were logically entailed by the background knowledge and the negated example. Such a conjunction - denoted $\overline{\perp}$ - in general, may contain infinitely many ground literals [Muggleton, 1995]. Given that $\overline{\perp}$ contains all ground literals which are true in all models of $B \cup \overline{e_i^+}$, and that $\overline{H}$ must also be true in every model of $B \cup \overline{e_i^+}$ then:

$$B \cup \overline{e_i^+} \models \overline{\perp} \models \overline{H} \tag{2.9}$$

From which

$$H \models \perp \tag{2.10}$$

In ILP $\perp$ is known as the **most specific hypothesis**, the **bottom clause**, and sometimes simply as **bottom**.

Notice that for the Grammar example above this relationship accords with the observed relationships presented in equations 2.7 and 2.8. By applying the deduction theorem it is possible to restate the relationship between hypotheses and $\perp$ as being logical implication:

$$\models H \rightarrow \perp \tag{2.11}$$

It has been shown, however, that implication between general clauses is undecidable [Schmidt-Schauss, 1988]. For Horn clauses, any implication between non trivial clauses[12], and sets of clauses is undecidable. An alternative ordering on the hypothesis space is that of θ-subsumption, which is presented in section 2.3.2. Put simply, the bottom clause $\perp$ *anchors* a lattice of Horn clauses $\mathcal{H}$ ordered by the θ-subsumption relationship. For any hypothesis $H_i \in \mathcal{H}$ the posterior sufficiency relationship (see page 20) for the chosen example e^+ holds: $B \cup H_i \models e^+$.

2.2.2 Implementing Inverse Entailment

In general $\overline{\perp}$ may contain an infinite number of ground literals. This is to be expected as minimal Herbrand models may contain infinitely many atoms. The process of generating bottom is referred to in [Rouveirol, 1992] as *saturating* the bottom clause. Practical techniques to overcome infinite bottom clauses constrain the atoms that can be considered when generating $\perp$. Three main techniques are employed by ILP systems which restrict the cardinality of $\perp$: 1) applying language restrictions; 2) applying restrictions on the non-determinism permitted in hypotheses; and 3) placing limits on proof depth. Such restrictions enable a finite subset of the minimum Herbrand model $\mathcal{M}^+(B \cup \overline{e_i^+})$ to be generated when producing $\perp$. The combination of all these techniques is known as *Mode Directed Inverse Entailment* (MDIE) [Muggleton, 1995]. This section details the main mechanisms involved in generating $\perp$ using MDIE.

[12]Non trivial in this case refers to a minimum of three Horn clauses.

Language restrictions

Let the definite clause language for a clausal theory T, $\mathcal{L}_T$ consist of the tuple $\langle \mathcal{P}_T, \mathcal{F}_T, \mathcal{V}_T, \mathcal{C}_T \rangle$ where $\mathcal{P}_T$ is the set of predicate symbols in T, $\mathcal{F}_T$ is the set of function symbols in T, $\mathcal{V}_T$ is the set of variable symbols in T, and $\mathcal{C}_T$ is the set of constant symbols in T.

The restrictions on the clausal language when performing MDIE are known collectively as the *Definite Mode Language* (DML) (see [Muggleton, 1995] page 261).

Example 14 (Grammar (part 8)) *Let the following set of clauses B', which when added to $B_{grammar_0}$, make the new background theory for the Grammar be $B'_{grammar_0}$*[13]*:*

$$B' = \left\{ \begin{array}{ll} listofwords([W]) \leftarrow word(W). & \\ listofwords(L_{Out}) \leftarrow listofwords(L_{In}), & \\ \quad word(W), insert(W, L_{In}, L_{Out}). & \\ & \\ word(the) \leftarrow & word(president) \leftarrow \\ word(has) \leftarrow & word(a) \leftarrow \\ word(cat) \leftarrow & \end{array} \right\}$$

The definite clause language for $B'_{grammar_0}$ is:

$$\mathcal{P}_{B'_{grammar_0}} = \left\{ \begin{array}{l} sentence/2, noun_phrase/2, verb_phrase/2, \\ determiner/2, noun/2, verb/2, \\ listofwords/1, word/1, insert/3 \end{array} \right\}$$

$$\mathcal{F}_{B'_{grammar_0}} = \{.()\}$$

$$\mathcal{V}_{B'_{grammar_0}} = \{R, T, S, O, P, W, L_{In}, L_{Out}\}$$

$$\mathcal{C}_{B'_{grammar_0}} = \{a, cat, has, president, the\}$$

The first restriction on selecting from the minimum Herbrand model when generating $\overline{\perp}$ is to allow only a finite subset of predicate symbols from the background theory $\mathcal{P}_{DML}$, to be used in $\overline{\perp}$ (i.e. $\mathcal{P}_{DML} \subseteq \mathcal{P}_T$).

Example 15 (Grammar (part 9)) *Let the restrictions on predicates for generating $H_{(e_1^+, B'_{grammar_0})}$ be $\mathcal{P}_{DML_{B'_{grammar_0}}} = \{noun_phrase/2, verb/2\}$.*

The second restriction relates to the Herbrand universe of ground terms $\mathcal{U}(T)$, that may be used in constructing atoms of the minimum model. These are restricted in two ways, first by restricting the functions that may appear

[13]The predicate *insert*/3 non-deterministically inserts an element (the first argument) into a list (second argument) to form another list (the third argument).

in $\mathcal{U}(T)$, and second by restricting the variables and function symbols that may appear (e.g. $\mathcal{F}_{DML} \subseteq \mathcal{F}_T$). In the example above there is a single function symbol .(), the list constructor.

Two ways in which variables can be restricted are: (1) *typing* and (2) *input-output modes.*

Definition 16 (Type) *A* type τ *is a set of (typically ground) atoms from a clausal theory* T *, from which a variable or constant may take its value. That is* $\tau_{typename} = \{x \mid typename(x) \in \mathcal{M}^+(T)\}$

Types define the set of values (or ground terms) that certain variables in the mode language are permitted to take.

Example 17 (Grammar (part 10)) *The type* τ_{word} *may be considered as a type for* $B'_{grammar_0}$. $\tau_{word} = \{a,\ cat,\ has,\ president,\ the\}$.

For this example there are only two basic types words *and* listsofwords. *However, one could imagine two extra types* shortwords *and* longerwords. *These types could be defined such that* shortwords *have three or fewer characters, and* longerwords *have four or more characters. This would lead to the following types:* $\tau_{shortwords} = \{a,\ cat,\ has,\ the\}$ *and* $\tau_{longerwords} = \{president\}$.

The second variable restriction is that of input-output usage. In logic programming, it is common [Deransart *et al.*, 1996] to specify the *mode* in which a variable is used within a predicate definition. Although modes do not effect the meaning of a predicate, they can be important in the procedural execution of the predicate.

Example 18 (Grammar (part 11)) *Consider the predicate* sentence(S,R)*, which is true if* S *is a legal sentence (with a remaining list of words of* R*). One* **mode** *of applying the predicate is to provide a ground term for* S *as* **input** *and for the predicate to return possible instantiations of* R *as* **output** *- and return all the lists of words that are suffixes of the ground sentence* S*. Alternatively, the predicate* sentence(S,R) *could be used to determine the sentences of a particular instantiation*[14] *of* R*. For this case* R *would be provided as* **input** *and instantiations of sentences,* S *would be returned as* **output**. *A third* **mode** *for the* sentence/2 *predicate would be to provide both variables,* S *and* R*, as* **inputs**. *In this* **mode** *the predicate will return the truth or falsity of the relationship between two ground terms. For instance, consider the query "Is it true that* $[a, cat, has, a, president]$ *is a legal sentence with an empty suffix,* $[]$*?".*

[14]This would be require some effort to program in PROLOG.

Definition 19 (Variable modes) *An* **input** *variable X, to a predicate is a variable that must be bound before the predicate is used in a resolution step (i.e.* called *in a* PROLOG *refutation). An input variable is denoted $+X$. An* **output** *variable Y, from a predicate is a variable that need not be bound before use. An output variable is denoted $-Y$. Typically an output variable will be bound by the execution of the predicate.*

Example 20 (Grammar (part 12)) *Three possible input-output definitions for sentence/2 may be represented as:*

$$\begin{aligned} sentence/2_a &= sentence(+X, -Y). \\ sentence/2_b &= sentence(-X, +Y). \\ sentence/2_c &= sentence(+X, +Y). \end{aligned}$$

Notice that the first two specifications of sentence/2 are non-deterministic, with more than one possible set of outputs for any given ground input. The third specification for sentence/2 may only return yes *or* no *to a* PROLOG *query to a set of given inputs.*

The final restriction on variables is that the variable must be instantiated to a **constant**[15]. Let the variable to be used as a constant input to a predicate be represented as $\#Z$.

Example 21 (Grammar (part 13)) *Consider the predicate noun_phrase/3, where the third argument represents the number of words in the first argument. The predicate may be defined with the following input-output modes.*

$$noun_phrase/3 = noun_phrase(-X, +Y, \#N)$$

The following are legitimate ground instantiations for the predicate from $B'_{grammar_0}$:

$$noun_phrase([the, cat, has, the, president], [has, the, president], 5).$$
$$noun_phrase([a, cat, has, the, president], [has, the, president], 5).$$
$$noun_phrase([the, president], [], 2).$$

When generating the bottom clause using mode directed Inverse Entailment, all atoms that appear in bottom require that a mode for every variable be specified. An atom that has the mode for all variables specified is called a completely mode specified atom.

Example 22 (Grammar (part 14)) *Consider again the predicate noun_phrase/3. One complete mode specification for* noun_phrase *is*[16]*:*

$$noun_phrase/3_a = noun_phrase(-listofwords, +listofwords, \#nat)$$

[15] In general, this restriction can be a *ground term.*

[16] The unary predicate *nat* defines natural numbers.

Such a specification can be considered as a definite clause where each variable appearing in the clause head appears in a unary body literal whose predicate symbol is the defined type for the variable[17]*. For the above predicate this clause can be written as:*

$$\begin{aligned} noun_phrase(X,Y,Z) \quad \leftarrow \quad & not\ var(Y), listofwords(Y), \\ & not\ var(Z), nat(Z), \\ & listofwords(X). \end{aligned}$$

Here not var(V) *specifies that the variable V is ground. The clause can be read as follows: "To generate an atom of noun_phrase/3 with the input-output specification as above, find an instance of type* listofwords Y_i *which is ground, and an instance of type* nat Z_j *which is also ground. Accept the atom $noun_phrase(X_i, Y_k, Z_j)$ if it succeeds with a ground instance for X_k and X_k is also of type* listofwords*, else reject the atom". The following are ground instantiations for the predicate from $B'_{grammar_0}$ using the above mode specification.*

$$noun_phrase([the, cat, has, the, president], [has, the, president], 5).$$
$$noun_phrase([a, cat, has, the, president], [has, the, president], 5).$$

The restrictions presented above do not guarantee a finite Herbrand model for $\overline{\perp}$. It is still possible that recursive predicate definitions will lead to an infinite set of ground atoms that satisfy the language restrictions that include predicate and function restrictions, and variable and mode restrictions. A further restriction, that places an upper bound on the number of atoms in bottom is *finite recall*.

Finite recall

One method for limiting a potentially infinite number of atoms to be returned with a single predicate symbol is to restrict the number of times the predicate can be *recalled* with the same input bindings. The recall restricts the cardinality of the set of the predicate's image under those bindings.

Example 23 (Grammar (part 15)) *Consider the binary predicate* noun *with the following input-output mode relationship* $noun(-listofwords, +listofwords)$*. Then with a recall number $r = 1$ and an input of $listofwords([])$ the sets of size 1 or less are*[18]*:*

$$|Atoms_{noun}(\!|\{[]\}|\!)|_{r=1} = \{\{noun([cat], [])\}, \{noun([president], [])\}\}$$

[17] Similar techniques are suggested in [Frisch, 1999].

[18] $R(\!|S|\!)$ is the *relational image* for relation R under set S (see [Potter *et al.*, 1991] page 92).

For a recall number of 2 there is one set of size 2 (and two sets of size 1).

$$|Atoms_{noun}(\{[]\})|_{r=2} = \left\{ \begin{array}{l} \{noun([cat],[]),\{noun([president],[])\}, \\ \{noun([cat],[]),noun([president],[])\} \end{array} \right\}$$

Contrast this with verb with the same input-output mode relationship. There is only a single atom that is returned when the second argument is bound to []. *That is* $verb([has],[])$.

Notice that the recall number only bounds the number of ground instances of an atom for any one set of input-constant variable bindings, and does not guarantee that there will be at least that number of atoms. Consider a recall number of 3 and the $noun/2$ example above. There does not exist a set of atoms of size 3 for the predicate. For the $verb/2$ example there are no sets of atoms of size 2.

Restricted proof depth

The size of the Herbrand model of a logic program can be restricted to its *h-easy* [Blum and Blum, 1975; Shapiro, 1983] subset.

Definition 24 (h-easy models) *(From [Muggleton and Feng, 1992] page 287) Given a logic program T, an atom α is* h-easy *with respect to T, if and only if there exists a derivation of α from T involving at most h binary resolutions. The Herbrand h-easy model of T, $\mathcal{M}_h(T)$, is the set of all Herbrand instantiations of* h-easy *atoms of T.*

A consequence of this definition is that for any finite h, the number of *h-easy* atoms of T is finite.

2.2.3 A definite mode language for Inverse Entailment

In the preceding sections a number of restrictions were placed on the theory T to restrict the cardinality of the minimum Herbrand model $\mathcal{M}^+(T)$. These restrictions ensure that the bottom clause generated by Inverse Entailment remains finite. Such restrictions are communicated to the ILP system Progol [Muggleton, 1995] by a *definite mode language.*

Definition 25 (Mode declaration) *A mode declaration (from [Muggleton, 1995] page 261) is either a head mode declaration or a body mode declaration. A head mode declaration has the form $modeh(r, Atom)$, while a body mode declaration has the form $modeb(r, Atom)$. The* recall number *r is either an integer ≥ 1 or a '*'. Atom is either a ground atom, or an atom where the variables are restricted to* typed place holders. *A typed place holder is the name of a type preceded by one of the following input-output usage symbols: $+$, $-$, or $\#$. The sign of a modeh atom is positive, and the sign of a modeb atom is negative. Mode declarations are illustrated below.*

Example 26 (Grammar (part 16)) *The following are valid mode declarations for the atoms from* $B'_{grammar_0}$.

```
:- modeh(1,verb_phrase(+listofwords,-listofwords)).
:- modeb(*,verb(+listofwords,-listofwords)).
:- modeb(*,noun(+listofwords,-listofwords)).
:- modeb(*,determiner(+listofwords,-listofwords)).
:- modeb(1,noun_phrase(+listofwords,-listofwords)).
```

The first mode declaration indicates that the head of the bottom clause to be constructed by Inverse Entailment should have predicate symbol $verb_phrase/2$, *whose arguments are instantiated to constants of type* $listofwords$. *The procedural mode in which the predicate* $verb_phrase/2$ *is to be used is that the first argument will be provided as input, and the second should be output by the predicate.*

The last mode declaration, restricts the body literals of bottom to contain only one atom of $noun_phrase/2$ *provided that the first argument is of type* $listofwords$, *and the second argument is of type* $listofwords$.

The following clause is legal within the mode declarations above.

$$\overline{H_{(e_1^+, B_{grammar_0})}} = noun_phrase([the, president], []) \wedge determiner([the], []) \wedge noun([president], []) \wedge \overline{verb_phrase([has, a, cat], [])} \wedge verb([has], []) \wedge noun_phrase([a, cat], []) \wedge determiner([a], []) \wedge noun([cat], []) \wedge \overline{sentence([the, president, has, a, cat], [])} \quad (2.12)$$

To summarise, the generation of a most specific clause, or bottom, using mode directed Inverse Entailment is performed by: 1) restricting the predicate and function symbols that can be present in bottom; 2) restricting the constants that may appear in bottom by providing variables in the remaining predicates and functions with typing and mode information; 3) restricting the number of ground atoms of such predicates returned from any one set of inputs from the definition. An algorithm for constructing a most specific (bottom) clause is presented in figure 2.3 on page 43.

2.3 Alternative hypotheses

In section 2.2.1 (page 24) it was demonstrated that there may be alternative hypotheses clauses H_i which imply $\bot$, and also satisfy ILP condition 4: Posterior Sufficiency. This section considers how alternative hypotheses may be ordered. Given that implication between such an hypothesis H_i and $\bot$ is undecidable for non-trivial Horn clause theories, the weaker alternative, θ-subsumption is often used. θ-Subsumption imposes a lattice over equivalence

classes of clauses, with a ..." bottom element $\perp$, and a top element $\square$ (the empty clause). Such a lattice represents a generality ordering over hypotheses from most general ($\top = \square$) to most specific ($\perp$).

This section introduces the notion of generality orderings over hypotheses, and presents the generality ordering induced by θ-subsumption. Much of this material is adapted from [van der Laag, 1995].

2.3.1 Generality orderings and ILP

Consider two hypotheses H_{i_1} and H_{i_2} that satisfy ILP condition 4: Posterior Sufficiency with respect to some particular positive example e_i^+ that is $B \cup H_{i_1} \models e_i^+$, and $B \cup H_{i_2} \models e_i^+$. Then

- H_{i_1} is too weak or overly *specific*, iff $B \cup H_{i_1} \not\models e_j^+$ for some positive example ($i \neq j$).
- H_{i_2} is too strong or overly *general*, iff $B \cup H_{i_2} \models e_k^-$ for some negative example.

Definition 27 (Generality ordering) *A* generality ordering *is a quasi-ordering*[19] $\geq$ *on clauses C and D, such that $C \geq D$ implies $C \models D$. Given a generality ordering $\geq$ and two clauses C and D, such that $C \geq D$, C is said to be* more general *than D, and D is said to be* more specific *than C. In such a case C is also called a* generalisation *of D, and D a* specialisation *of C.*

Different generality orderings have been proposed for ILP. The definition above is concerned simply with two clauses without any relationship to any existing background theory (should it be present). *Relative Subsumption*, *Relative Implication* [Plotkin, 1971], and *Generalised Subsumption* [Buntine, 1988] all define notions of generality relative to an existing background theory. The strongest generality relationship between two clauses is logical entailment ($C \models D$), while the weaker ordering is that of *θ-Subsumption* . The relationship between four generality notions is shown in the following table.

Table 2.1: The relationship between different generality orders in ILP.

WEAKER			STRONGER
θ-subsumption	gen-subsumption	rel-subsumption	rel-implication
$\succeq$	$\geq_B$	$\succeq_B$	$\models_B$

For various types of generality ordering, researchers have proposed operators that compute more specific or more general clauses from initial sets of clauses. These operators are known as *refinement operators.*

[19] A quasi-ordering is a reflexive and transitive binary relation. Where C and D are limited to equivalence classes then the generality ordering is a *partial-order.*

2.3.2 The generality ordering of θ-subsumption

θ-Subsumption is an important generality ordering for ILP as it is often used as an alternative to logical implication [van der Laag, 1995]. This section introduces θ -subsumption.

Definition 28 (Substitution) *A* substitution θ *is a finite set of the form* $\{X_1/t_1, \ldots, X_n/t_n\}$ $(n > 0)$*, where each* X_i *is a distinct variable and each* t_i *is a term distinct from* X_i*.*

An element X_i/t_i *in a substitution is called a binding for* X_i*.*

θ *is called a* ground substitution *if each* t_i *is a ground term.*

The substitution given by the empty set is called the empty substitution *and is denoted by* ε*.*

Definition 29 (θ-Subsumption) *Let* C *and* D *be two clauses and* $[C]$ *be a set of clauses. Then*

1. C θ-subsumes D*, denoted by* $C \succeq D$*, iff there exists a substitution* θ *such that* $C\theta \subseteq D$ *.*
2. C properly θ-subsumes D*, denoted by* $C \succ D$*, iff* $C \succeq D$ *and* $D \not\succeq C$*.*
3. C *and* D *are called* subsume-equivalent*, denoted by* $C \sim D$*, iff* $C \succeq D$ *and* $D \succeq C$*.*
4. *The set of all clauses that are subsume-equivalent to* C *,* $[C]$ *is called a (*θ-subsumption) equivalence class *or simply an* equivalence class*.*

θ-Subsumption is a weaker relation than logical entailment. In general, θ-subsumption is not equivalent to logical entailment. That is if $C \succeq D$ then $C \models D$, but the reverse is not necessarily true. θ-Subsumption is a quasi-order over the language of first order clauses $\mathcal{C}$, and in particular the language of Horn clauses $\mathcal{H}$.

Definition 30 (most general instance/least general generalisation) *Let* l *and* l' *be two compatible literals (having the same predicate symbol and sign). If* $l\theta = l'\theta$ *for some substitution* θ *(a unifier) there is a* most general unifier (mgu) μ *with the properties*

1. $l\mu = l'\mu$
2. *For any* σ*, if* $l\sigma = l'\sigma$ *then there is a* δ *such that* $l\sigma = l\mu\delta = l'\mu\delta$*.*

The most general instance (mgi) *of two literals* l *and* l' *is* $l\mu = l'\mu$*.*
The least general generalisation (lgg) *of a set of (compatible) literals* L *is* k *iff*

1. *For every* $l \in L$ *then* $k\theta = l$

2. If for every $l \in L$ there exists $k'\theta = l$, then $k' = k\theta$

The definition of clause mgi and lgg *is as follows. Let C and D be two* clauses *(sets of literals) which share no variables.*

The least general generalisation (l.g.g.) *of C and D is*
$lgg(C, D) = \{lgg(l, m) : l \in C$ *and* $m \in D$ *and* l *and* m *are compatible*$\}$
The most general instance (m.g.i.) *of C and D is*
$mgi(C, D) = C \cup D$

Definition 31 (Lattice) *(From [Davey and Priestley, 1990]) Let P be a non-empty ordered set. If for all $x, y \in P$ the* greatest lower bound (glb) *and the* least upper bound (lub) *exist, then P is called a* lattice.

With the addition of top and bottom elements ($\top = \square$ and $\bot = \bot_{(B,e^+)}$) a least upper bound and greatest lower bound exists for each pair of clauses $C, D \in \mathcal{H}$, ordered by θ-subsumption. In ILP the least upper bound is known as the *least general generalisation (l.g.g.)* [Plotkin, 1969], and the greatest lower bound as the *most general instance (m.g.i.)* [Robinson, 1965].

Definition 32 (θ-Subsumption Lattice) *Let $\top = \square$ and $\bot = \bot_{(B,e^+)}$.*

$$\mathcal{H}_1^+(B, e^+) = \{C : \top \succeq C \succeq \bot\}$$

is a lattice ordered by θ-subsumption as follows:

lub $[C] \sqcup [D] = [lgg(C, D)]$

glb $[C] \sqcap [D] = [mgi(C, D)]$

Notice that the lattice $\mathcal{H}_1^+(B, e^+)$ is determined by the following:

- The choice of example e^+;
- The background knowledge B; and
- The declarative bias (including the mode language $\mathcal{M}$.

A change in any of these above items will potentially result in a different bottom clause, which will "anchor" the θ-subsumption lattice. In particular, consider the situation when *extraction case abduction* (see section 3.4 on page 55) from a single example e^+ produces alternative atoms that can be used to produce the head of the bottom clause. In such a situation one example provides a set of θ-subsumption lattices.

The lattice structure that θ-subsumption induces over the set of single clause hypotheses, $\square \succeq H_i \succeq \bot_{(B,e^+)}$ provides a straightforward method of enumerating all such hypotheses that satisfy ILP condition 4: Posterior Sufficiency. One can either traverse the lattice from top down, or bottom up. Top down search typically starts from the empty clause from which

a set of specialisations are generated. In a similar manner the search can start from the most specific clause from which a set of generalisations are computed.

Generality orderings that relate clauses independently from other clauses or programs, with operators that take a single clause as argument and return a set of more specific (more general respectively) are known as *refinement operators*. Refinement operators which take as input a single clause, and generate the set of all specialisations (or generalisations respectively) are known as downward refinement operators (upward refinement operators respectively), and are given the symbol ρ. The set of all clauses produced from a refinement operator and a starting clause is ρ^*. Properties of refinement operators are studied in [van der Laag, 1995]. For the case of θ-subsumption based refinement in the subsumption lattice $\rho^*_{\succeq}(H) = \left\{H_i |\ H \succeq H_i \succeq \bot_{(B,e^+)}\right\}$.

Practically, a clause H can be specialised (or downward refined) to the clause H_i (such that $H_i \succeq \bot_{(B,e^+)}$) by:

- adding to H an extra literal from $\bot_{(B,e^+)}$ to form H_i;
- or by making a substitution θ, where $H\theta = H_i$, and the domain of θ is the terms in $\bot_{(B,e^+)}$;
- or both

Example 33 (Grammar (part 17)) *Below are some hypothesis clauses for the grammar which are produced by refinements starting from the clause* $verb_phrase(S,R) \leftarrow$.

$$\rho_{\succeq}(verb_phrase(S,R)) =$$
$$\left\{ \begin{array}{lcl} verb_phrase(S,R) & \leftarrow & verb(S,U) \\ verb_phrase(S,R) & \leftarrow & determiner(U,V) \\ & \vdots & \end{array} \right\}$$

$$\rho_{\succeq}(verb_phrase(S,R) \leftarrow verb(X,Y)) =$$
$$\left\{ \begin{array}{lcl} verb_phrase(S,R) & \leftarrow & verb(S,U), determiner(U,V) \\ verb_phrase(S,R) & \leftarrow & verb(S,U), noun_phrase(U,R) \\ & \vdots & \end{array} \right\}$$

$$\rho_{\succeq}(verb_phrase(S,R) \leftarrow verb(S,U), determiner(U,V)) =$$
$$\left\{ \begin{array}{lcl} verb_phrase(S,R) & \leftarrow & verb(S,U), determiner(U,V), noun(V,R) \\ & \vdots & \end{array} \right\}$$

Notice that in each case, the hypothesis H_i *θ-subsumes* $H_{(e_1^+, B_{grammar_0})}$. *That is* $H_i \succeq H_{(e_1^+, B_{grammar_0})}$ *(refer to equation 2.12) which ensures that* $B_{grammar_0} \cup H_i \models e_1^+$.

The next section introduces techniques for selecting between competing hypotheses in the subsumption lattice.

2.4 Hypothesis testing

A blind search of the complete lattice over the hypothesis space is undesirable. A more sophisticated approach is required to navigate the lattice towards a *satisfactory* hypothesis. It is therefore important to consider what attributes a satisfactory hypothesis might have, and some implementation techniques for measuring them. This section introduces some methods by which ILP systems test provisional hypotheses, and perhaps reject them.

Definition 34 (Covers) *An example e is said to be* covered *by a hypothesis H_i and background knowledge B if*

$$B \cup H_i \models e.$$

Cover testing can be implemented using PROLOG's resolution theorem prover. The set of examples $E'_{(B,H_i)}$ covered by an hypothesis H_i, is $E' = \{e : e \in E \text{ and } B \cup H_i \cup \overline{e} \vdash_h \square\}$, where E is the complete set of examples, and h is the maximum proof depth. E' can be partitioned into four sets[20] - correctly covered positive examples $E^+_{\blacksquare}$, correctly not covered negative examples $E^-_{\blacksquare}$, incorrectly not covered positive examples $E^+_{\square}$, and incorrectly covered negative examples $E^-_{\square}$,

ILP conditions 2 and 4 provide some attributes that a candidate hypothesis clause should have. ILP condition 2 - Posterior Satisfiability, requires that no negative examples be covered, that is $E^-_{\square} = \emptyset$. This suggests that any candidate hypothesis which covers any negative example be rejected. ILP condition 4: Posterior Sufficiency only requires that an hypothesis need cover one positive example to be a candidate. Naturally, an hypothesis that covers more positive examples than another hypothesis should be preferred (providing neither covers any negatives).

In some applications the requirement that an hypothesis clause not cover any negative examples is too strong. In particular when the example set – and possibly the existing background theory – are suspected of containing miss-classified elements, or *noise*[21]. One simple strategy for coping with noise is to entertain hypotheses that cover no more than a pre-determined amount of negative examples. More sophisticated noise handling methods involving information encoding techniques are discussed in [Lavrac and Dzeroski, 1994]. These methods provide the theoretical basis for compression measures in ILP and are discussed below.

2.4.1 Selecting from a set of competing hypotheses

When two or more provisional hypotheses have the same coverage one may employ Ockham's razor and select the shortest. Care must be taken when

[20] $\square$ represents falsity, and $\blacksquare$ truth.

[21] In such circumstances ILP Condition 2 - *Posterior Satisfiability* is violated.

defining a length measure for competing hypotheses, that as well as their size, the size of derivations of the examples using the background theory is also considered. A model for the justification of first order theories based on data compression is developed in [Srinivasan *et al.*, 1994]. The model accounts for not only compression of the examples, but the proof lengths required to regenerate the examples from the background theory and the hypothesis.

One simple scheme for ranking competing hypothesis is the *pseudo compression* measure f, which is defined as (see [Muggleton, 1995] page 284):

$$f = p - n - l - h \tag{2.13}$$

where p is the number of positive examples covered, n is the number of negative examples covered, l is the number of literals in the hypothesis, and h is an optimistic estimate of the number of additional literals required to complete the input-output of the hypothesis. By maximising f there is a preference for shorter hypotheses which cover more positive examples, and less negative examples.

2.4.2 Meta-logical search control

Sometimes domain knowledge is available that can be used to reduce the amount of the θ-subsumption lattice that need be searched. Such domain knowledge to control the search is called *search bias*. It may be known, for example, what the syntactic structure is that all acceptable[22] hypotheses must have. Conversely, it may be known the structures that they must *not* have. Two particular types of search bias are *integrity constraints*[23] and *prune statements* [Srinivasan and Camacho, 1999].

An integrity constraint, in database terminology, is a condition that must not be broken to ensure integrity of the database.

Example 35 (Grammar (part 18)) *An* integrity constraint *for the grammar database is:* no list of words can be both a verb phrase and a noun phrase. *This statement may be translated to the Horn clause*

$$\square \leftarrow verb_phrase(S, R), noun_phrase(S, R).$$

which states that it is false *if a list of words is both a legal verb phrase* and *a legal noun phrase.*

Each candidate hypothesis is tested for constraint violation. A test for constraint violation can be performed simply by testing whether the distinguished predicate $false$ (to represent $\square$) can be derived. It is assumed that

[22] Acceptability may be a subjective notion, provided by the inductive logic programmer.

[23] Not to be confused with the constraints as used in *Constraint Logic Programming* as they can also refer to permissible structures of hypotheses. See example 36.

no constraints are violated by the initial background theory before the addition of the candidate hypothesis. All hypotheses that violate any constraints are rejected.

A further use for constraints is to specify components that an acceptable hypothesis *must* have. This will be referred to as an **hypothesis structure constraint**.

Example 36 (Grammar (part 19)) *An* hypothesis structure constraint *for the grammar database is:* the hypotheses that define sentence must contain at least one reference to verb phrase*. This statement may be translated to the Horn clause*

$$\square \leftarrow hypothesis((Head :- Body)),\ not\ in(Body, verb_phrase(S, R)).$$

which states that it is false if an hypothesis does not contain the literal $verb_phrase(S, R)$ *in its body*[24]*.*

Prune statements (Taken from [Srinivasan and Camacho, 1999] page 191.) are meta-logical search control rules which restrict the hypothesis nodes which are searched. Such statements specify what structure candidate clauses may not have, for example a particular set of literals that may not appear in the hypothesis. Should a clause have the undesirable structure, then all further refinements of such a clause will also contain the structure. This allows the entire search tree to be pruned beneath[25] the first clause that matches the structure. Judicious prune statements can produce large reductions in the search space.

Example 37 (Grammar (part 20)) *A* prune statement *for the grammar database is:* hypotheses that define a sentence must **not** contain more than one reference to noun phrase*. This statement may be translated to the Horn clause*

$$\begin{aligned} prune((Head :- Body)) \quad \leftarrow \quad & in(Body, noun_phrase(U, V)), \\ & in(Body, noun_phrase(W, X)), \\ & U \neq W,\ V \neq X. \end{aligned}$$

The use of user defined search bias can improve the efficiency of the hypothesis searching. Naturally, care must be taken as the use of such bias introduces incompleteness into the search process.

[24]The predicate $in(A, B)$ is true when the atom B is *in* a comma separated list of atoms A, and the predicate *hypothesis* contains the clause under consideration.

[25]This presumes that search is being performed in a top-down or *general-to-specific* manner.

2.4.3 An introduction to Positive-only[26] scoring

In section 2.4.1 scoring was introduced as a method for ranking potential hypotheses from those in the refinement lattice[27]. Some ILP algorithms (e.g. [Muggleton, 1995]) have the ability to score competing clauses based on some pseudo compression function similar to equation 2.13.

But what happens when there are no negative examples? If $n = 0$ in equation 2.13, then the shortest hypothesis will be selected so that f is maximised. This will result in the generic rule "*everything is in the target concept*".

Example 38 (Grammar (part 21)) *In the context of the grammar example the shortest rule that allows all of the positive examples to be derived (or* explained*) with the background knowledge is* $verb_phrase(S, R) \leftarrow$. *This translates to "*every sentence is a verb phrase*". This is clearly an* over general *hypothesis.*

At the other end of the hypothesis spectrum is that of a very *large* hypothesis. Consider the *bottom clause* as an hypothesis of this type (for example see the clause in equation 2.12)[28]. Normally in machine learning, smaller hypotheses explaining all the examples are preferred to larger hypotheses.

Learning from positive-data only requires solving the hypothesis-generality versus hypothesis-size tradeoff. The description that follows is adapted from [Muggleton, 1996].

A Bayesian scoring function

In [Muggleton, 1996] the hypothesis generality–size trade-off is presented as the optimisation of the *Bayes' posterior*. The Bayesian setting is:

$$p(H|E) = \frac{p(H)p(E|H)}{p(E)} \tag{2.14}$$

where H represents an hypothesis under consideration, E is the examples (or evidence); $p(H|E)$ is the *posterior* probability of choosing H *given* E; $p(H)$ is the *prior* probability of selecting H from the set of all possible legal hypotheses ($\mathcal{H}$); $p(E|H)$ is the *likelihood* of the examples appearing in the presence of the hypothesis; and $p(E)$ is the probability of the examples being selected from all possible examples.

[26] Short for "positive data only".

[27] Recall (page 34) that *all* hypotheses in the lattice satisfy ILP Condition 4 in some way.

[28] Recall that an *unrestricted* bottom clause can easily contain infinitely many literals.

The size[29] of an hypothesis is

$$sz(H) = -\log_2 D_{\mathcal{H}}(H) \tag{2.15}$$

while the generality of an hypothesis is:

$$g(H) = D_X(H) \tag{2.16}$$

Where $D_{\mathcal{H}}$ is the probability distribution associated with all possible hypotheses, and $D_X(H)$ is the probability distribution of all instances (or possible observations) with respect to a particular hypothesis H (for more information on the U-Learnability framework see [Muggleton and Page, 1994]).

From equation 2.14 it is possible to derive the following:

$$p(H|E) = D_{\mathcal{H}}\left(\frac{1}{D_X(H)}\right)^m c_m \tag{2.17}$$

Where m is the number of (positive) examples, and c_m is a constant. Taking logs of both sides of 2.17 and substituting for $D_{\mathcal{H}}$ and $D_X(H)$ from equations 2.15 and 2.16 leads to

$$\log p(H|E) = m \log\left(\frac{1}{g(H)}\right) - sz(H) + d_m \tag{2.18}$$

Equation 2.18 can be approximated by the function

$$f_m(H) = m(1 - g(H)) - sz(H) \tag{2.19}$$

Implementing the Bayesian scoring function

In the development above the outcome was to maximise the function in equation 2.19. In an implementation of this function it is necessary to calculate $g(H_i)$ and $sz(H_i)$ for the H_is considered in the search of the lattice of hypotheses. To recap, $g(H_i)$ is the proportion of *all* instances[30] that are *classified as positive* by the particular hypothesis H_i.

As the size of the instance space can be infinite, it is common to only consider a *random* sample of the instances for use in scoring. The relative *coverage* of a *random* sample of size r (without replacement[31]) with respect to the hypothesis is used as the measure of generality.

$$g(H_i) = \frac{\sum_{j=1}^{r} c}{r} \quad \text{where} \quad c = \begin{cases} 1 & \text{if } B \cup H_i \models e_j \\ 0 & \text{otherwise} \end{cases}$$

[29]Measured in *bits* required for encoding the hypothesis in some universal Turing machine.

[30]Both positive examples, and unclassified instances.

[31]Strictly the sample should be taken *with* replacement. However, as the size of the number of atoms in the Herbrand base approaches infinity this matters less.

Example 39 (Grammar (part 22)) *The instance space for the grammar of sentences with five words or fewer contains 3905 unique sentences*[32] *or "examples". Of these, only sixteen are correctly classified as* positive.

The size of any hypothesis $sz(H_i)$ can be crudely measured by the number of literals in H_i.

In figure 2.2 a basic ILP algorithm is presented which outlines the iterative nature of how an hypothesis $\mathcal{H}$ containing n clauses is constructed, one hypothesis at a time. That is $\mathcal{H} = \{H_1, H_2, \ldots, H_n\}$. In such a setting the aim is to maximise $f_m(\mathcal{H})$. However, within each iteration only one clause H_i is being selected from the subsumption lattice.

2.5 A basic ILP algorithm

Figure 2.2 presents the outline of an ILP algorithm based on mode directed Inverse Entailment, and summarises a more detailed version first shown in [Muggleton, 1995]. The algorithm implements the techniques presented in this chapter by repeatedly generalising single examples, each time producing a single hypothesis. When complete, the algorithm returns a set of hypotheses $\mathcal{H}$. Should no general hypothesis be compressive, the example itself e_i^+ will be added to $\mathcal{H}$, which guarantees that the algorithm will terminate, even in the case where no generalisations can be found. This algorithm is designed to process the observations in a single *batch* process, with no interaction with a human. This form of ILP system is known as *non-interactive*. Other ILP systems enable more interaction with a human, and take a single example at a time (for example CLINT in [De Raedt, 1992]). Such a system is known as *incremental*.

[32] $3905 = 5^1 + 5^2 \cdots + 5^5$

$ILP(B, E^+, \mathcal{L}, I_\mathcal{L}, E^r)$: Given background theory B, observations E^+, definite mode language $\mathcal{L}$, hypothesis constraints $I_\mathcal{L}$, a finite set of random observations E^r, returns an hypothesis $\mathcal{H}$ (a set of clauses) from definite mode language $\mathcal{L}$ such that $\mathcal{H}$ satisfies the following: $B \cup \mathcal{H} \models E^+$, $B \cup \mathcal{H} \not\models \square$, and $B \cup \mathcal{H} \cup I_\mathcal{L} \not\models \square$.

1. Set $\mathcal{H} = \emptyset$;
2. if $E_i^+ = \emptyset$ return $\mathcal{H}$;
3. Increment i;
4. Select $e_i^+ \in E_i^+$ (e.g. randomly or in sequence);
5. Generate $\bot_{(B \cup \mathcal{H}, e_i^+)}$ with respect to $\mathcal{L}$ using *saturate*() (see figure 2.3);
6. Search for the *best* scoring H_i with respect to the observations E^+ and E^r, subject to $I_\mathcal{L}$, where $\square \succeq H_i \succeq \bot_{(B \cup \mathcal{H}, e_i^+)}$;
7. $\mathcal{H}_i = \mathcal{H}_{(i-1)} \cup H_i$;
8. $E_i^+ = E_{i-1}^+ \setminus \{e_k^+\}$ where $e_k^+ \in E_{i-1}^+$ s.t. $B \cup \mathcal{H}_i \models e_k^+$;
9. Go to Step 2.

Figure 2.2: An ILP algorithm based on Inverse Entailment.

$saturate(B, a^+, \mathcal{L}, h, i)$:
Given background theory B (Horn clauses), a single positive atom a^+, a mode language $\mathcal{L} = \mathcal{M}_{head} \cup \mathcal{M}_{body}$, a depth bound h, a determinacy bound i; return a most specific clause $\bot_{(B,a^+)}$ such that $B \cup \overline{e^+} \models \overline{\bot_{(B,a^+)}}$

1. Let $k = 0$, $hash : Terms \rightarrow N$ be a hash function which uniquely maps terms to natural numbers, $\bot_i = \langle\rangle$ and $Terms_{Input} = \emptyset$, $atom(m_i)$ is the atom formed by the predicate/arity symbol in mode definition m_i.

2. If there is no modeh m in $\mathcal{M}_{head}$ such that $atom(m) \preceq a$ then return $\Box$. Otherwise let m be the first modeh declaration in $\mathcal{M}_{head}$ such that $atom(m) \preceq a$ with substitution θ_h. Let a_h be a copy of $atom(m)$ and for each v/t in θ_h if v corresponds to a #type in m then replace v in a_h by t otherwise replace v in a_h by v_k where $k = hash(t)$ (i.e. *variablise* the ground term) and add v to $Terms_{Input}$ if v corresponds to +type. Add a_h to $\bot_{(B,a^+)}$.

3. If $k = i$ return $\bot_{(B,a^+)}$ else $k = k + 1$.

4. For each modeb m in $\mathcal{M}_{body}$ with recall number r, let $\{v_1, .., v_n\}$ be the variables of +type in $atom(m)$ and $T(m) = T_1 \times .. \times T_n$ be a set of n-tuples of terms such that each T_i corresponds to the set of all terms of the type associated with v_i in m (term t is tested to be of a particular type by calling PROLOG with type(t) as goal). For each $\langle t_1, .., t_n \rangle$ in $T(m)$ let a_b be a copy of $atom(m)$ and $\theta = \{v_1/t_1, .., v_n/t_n\}$. If PROLOG with depth-bound h succeeds on goal $a_b\theta$ with the set of answer substitutions Θ_b of cardinality no more than r then for each θ_b in Θ_b and for each v/t in θ_b if v corresponds to a #type in m then replace v in a_b by t otherwise replace v in a_b by v_k where $k = hash(t)$ and add v to $Terms_{Input}$ if v corresponds to -type. Add $\overline{a_b}$ to $\bot_{(B,a^+)}$.

5. Goto step 3.

Figure 2.3: An algorithm for generating a most specific (bottom) clause (adapted from [Muggleton, 1995]).

2.6 Summary

Inductive logic programming aims to obtain general laws from specific examples in the presence of an existing background theory. Page ([Page, 2000] page 16) points out:

> ILP has attracted great interest within the machine learning and AI communities because of its *logical foundation*, its ability to utilise *background knowledge* and *structured data representations*, and its *comprehensible* results. But most of all, the interest has come from ILP's application successes[33].

The key techniques used in ILP include:

- Generation of a most specific hypothesis from a single positive example relative to background knowledge;
- A lattice over the hypothesis space of single Horn clauses is induced by θ-subsumption;
- The lattice can be traversed either top down, or bottom up;
- The suitability of each candidate hypothesis can be measured, and a most suitable hypothesis selected.

The techniques presented in this chapter emphasise that there is much work for the Inductive Logic Programmer to do. These tasks include the collection, verification, and encoding of background knowledge and examples; and the provision of the appropriate definite mode language. The definite mode language defines the type of hypotheses that the programmer is interested in. In a sense the ILP programmer has to encode the data and the settings to follow his hunch about likely hypotheses. This then conforms to a Popperian notion[34] of having to provide some form of *a priori* theory, or a notion of 'creative intuition'. However, in common with a Baconian or an inductivist approach, the induction proceeds from the examples and a background theory, to automatically construct hypotheses within the definite mode language. It is in no way the case that the ILP algorithm does all the work.

[33] Italics added for emphasis.
[34] As expressed in [Gillies, 1996]

Chapter 3

Theory Completion by Inverse Entailment

Theory Completion [Moyle and Muggleton, 1997] is a special case of the logic programming setting of Theory Revision [De Raedt, 1992]. Many techniques similar to those required for Theory Completion are available in Theory Revision and other fields (for example Abductive Logic Programming [Kakas *et al.*, 1993]). This chapter, however, investigates simple extensions to Inverse Entailment as a means towards developing a Theory Completion system.

Muggleton's Inverse Entailment [Muggleton, 1995] provides a basis for the theory of logical induction from positive examples. In Observational Predicate Learning a positive example provides the hypothesis predicate for the induction process. In Theory Completion predicates other than those of the examples may be used as the hypothesis predicate. Although the theory of Inverse Entailment encompasses such a setting, implementations are often restricted to producing a *bottom clause* with head predicates the same as the examples. This, in part, is because such implementations base their theorem proving on PROLOG. As such, standard PROLOG systems are insufficient for deriving negated atoms that would form the head of the bottom clause in predicates other than those of the observations. The generation of atoms that will form the body of a bottom clause is less problematic, simply requiring techniques for collecting a finite subset of the atoms in the Herbrand Model of the background theory in the presence of the negated observation ($\mathcal{M}(B \cup \overline{e^+})$).

In this chapter, two techniques are investigated which extend *SLD resolution* for definite logic programs so that negated atoms may be derived for use in the head of the bottom clause. These techniques are: 1) *Logical Backpropagation* (LBP) [Moyle and Muggleton, 1997]; and 2) *SOLD-resolution* [Yamamoto, 1997]. These two methods are compared, and it is shown that atoms derivable by LBP can be derived by SOLD-resolution. The methods for deriving negated atoms from examples and background knowledge

are forms of *extraction-case abduction* [Kakas *et al.*, 1993], [Dimopoulos and Kakas, 1996].

3.1 An introduction to Theory Revision and Theory Completion

This section introduces Theory Revision, and it is then shown how the "normal" semantics of ILP is simply a special case of Theory Revision. Theory Completion is presented as both an extension of the normal semantics of ILP, and a special case of Theory Revision.

3.1.1 Theory Revision

Theory revision is a general form of ILP. De Raedt [De Raedt, 1992] (page 10) outlines the problem of Theory Revision as:

- *Given*: integrity constraints I; evidence E (a set of well-formed formulae in the predicate calculus); and a theory $T = T_{fix} \cup T_{mod}$ (a set of well-formed formulae in the predicate calculus) satisfying I.
- *Find*: a theory $T' = T_{fix} \cup T'_{mod}$ obtained by applying transformations on T_{mod} (known as *revising* T_{mod}) such that: T' explains E, and T' satisfies I.

Compare the above description to the problem statement for ILP given in chapter 2 (on page 22) which can be paraphrased as:

- *Given*: integrity constraints I; evidence E ; and a background theory B satisfying I, and $B \nvDash E$.
- *Find*: an hypotheses H such that: $B \cup H \models E$, and $B \cup H$ satisfies I.

The specification for ILP is a subset of that for Theory Revision where $T_{fix} = B$, $T_{mod} = \emptyset$, $T'_{mod} = H$, and logical entailment is used for the notion of satisfaction.

Remark 40 (Acceptable revisions) *Notice that in theory revision, changes to T_{mod} can include all or any of the following*[1]*: (1) removing clauses from T_{mod}, (2) adding clauses to T_{mod}, and (3) no changes to T_{mod}.*

Many learning systems can be classified as Theory Revision systems. For example: MIS [Shapiro, 1983], MARVIN [Sammut and Banerji, 1984], CIGOL [Muggleton, 1988], CLINT [De Raedt, 1992], MINERVA [Taylor, 1996]. Such systems have the ability to identify false clauses for removal, and then suggest possible fixes.

[1] Altering any clause in T_{mod} can be achieved by step (1) followed by step (2).

3.1.2 Theory Completion

Theory Completion is simply a restricted form of Theory Revision, limited to only the **addition** of clauses to an existing theory.

Definition 41 (Theory Completion) *Let I,E, and T be sets of well-formed formulae in the predicate calculus. Then* Theory Completion *is defined as:*

- Given*: integrity constraints I; evidence E; and a theory $T = T_{fix} \cup T_{mod}$ satisfying I.*
- Find*: a theory $T' = T_{fix} \cup T'_{mod}$ obtained by* **adding clauses** *to T_{mod} (known as* completing T_{mod} *) such that: T' explains E, and T' satisfies*[2] *I.*

Example 42 (Theory Revision compared with Theory Completion) *The clause for noun_phrase/2 from example 11 in the previous chapter (on page 21) is $T_{mod} = noun_phrase(S,T) \leftarrow determiner(S,N), noun(N,T)$. An acceptable*[3] revision *for the clause would be to replace it by the two clauses in $T'_{mod_{TR}}$.*

$$T'_{mod_{TR}} = \left\{ \begin{array}{lcl} noun_phrase(S,T) & \leftarrow & noun(S,T), S = [a|T] \\ noun_phrase(S,T) & \leftarrow & noun(S,T), S = [the|T] \end{array} \right.$$

In the Theory Completion *setting $T'_{mod_{TR}}$ would* **not** *be acceptable because it involves replacing existing clauses by alternative clauses. However $T'_{mod_{TC}} = T_{mod} \cup T'_{mod_{TR}}$ would be acceptable*[4]*, as would the Completion below.*

$$T''_{mod_{TC}} = \left\{ \begin{array}{lcl} verb_phrase(S,T) & \leftarrow & verb(S,R), noun_phrase(R,T) \end{array} \right\}.$$

Remark 43 (Acceptable completions) *Compare the acceptable changes in Theory Completion to those in Theory Revision (remark 40). The changes to T_{mod} in* **Theory Completion** *are limited to: (1) adding clauses to T_{mod}, and (2) no changes to T_{mod}.*

`CProgol5.0` is a Theory Completion system and is described afresh in [Muggleton and Bryant, 2000]. The methods used in `CProgol5.0` are investigated later in this chapter.

[2] Here *satisfies* is taken to mean *Consistency* rather than the *Theoremhood* view.

[3] In practice, one would be unlikely to replace a general theory (e.g. T_{mod}), with a more specific theory (e.g. $T'_{mod_{TR}}$).

[4] $T'_{mod_{TC}}$ would not be a very useful Completion, as the clauses in $T'_{mod_{TR}}$ are redundant in the presence of T_{mod}.

3.2 Inverse Entailment Revisited

The condition of Posterior Sufficiency (ILP condition 4) (see page 20, chapter 2) provides the basis for generating an initial hypothesis H, from a single positive example $e_i^+ \in E^+$, and background knowledge B. The Inverse Entailment relationship is described by the following equation.

$$B \cup \overline{e_i^+} \models \overline{H} \tag{3.1}$$

Often, however, $\overline{e_i^+}$ is not necessarily the only negated atom, being a logical consequence of $B \cup \overline{e_i^+}$, that is of interest to the Inductive Logic Programmer. Indeed, a negated atom that *does not* have the same predicate symbol as the example is useful when completing theories. Derivation of negated atoms using SLD-resolution (as implemented in PROLOG) is problematic. The techniques of both *SOLD-resolution* and *Logical Back-propagation* aim to extend Inverse Entailment by providing such alternative negated atoms, $\overline{h}$. Both these techniques are investigated in the following sections of this chapter.

3.2.1 The role of Abduction in Theory Completion

Abduction as a form of (unsound) reasoning was introduced in chapter 1 (page 10). Abduction in logic programming – or Abductive Logic Programming (ALP) – is closely related to Induction in Logic programming [Dimopoulos and Kakas, 1996]. Some methods used in ALP have similar aims to those in Theory Completion.

To recap, abduction is the temporary adoption of *premises*[5] Δ without evidence, in the presence of some *existing theory* B, so that unexplained – but observed – *outcomes* E^+ can be satisfactorily explained.

Definition 44 (Abductive Logic Programming) *Let I,E^+, and B be sets of well-formed formulae in the predicate calculus. Then* Abduction *is defined as:*

- Given: *integrity constraints I; evidence E^+; and a theory B satisfying I; where $B \not\models E^+$.*
- Find: *a set of (ground)* abducibles[6] Δ *obtained such that: $B \cup \Delta$ explains E^+, and $B \cup \Delta$ satisfies I.*

[5] In chapter 1 *premises* were introduced as *cases*, while *outcomes* were introduced as *results*.

[6] *Abducibles* are a special, domain-specific, class of clauses ([Toni, 1995] page 32). Often, abducible predicates are a subset of those predicate symbols in the background theory B.

Compare the above definition for ALP with the definition of normal ILP (see page 46). The main difference between the Abductive and Inductive forms is that ALP is primarily concerned with generating *ground explanations* – or "missing facts" [Dimopoulos and Kakas, 1996] – from examples, while ILP aims to produces non-ground generalisations of the examples. This form of abduction that generates ground facts is termed **extraction-case abduction** (see also the introductory chapter in [Flach and Kakas, 2000]).

In extraction-case abduction, the set of abducibles Δ are considered to represent a more *fundamental* cause – than that of the observed fact – for the failure of B to predict E. The process of abduction converts the observation into something more fundamental, and that may be used as the basis of further reasoning – perhaps that of inductive reasoning.

Example 45 (Fundamental explanations) *Recall the unfinished grammar of chapter 2, and the observations of sentences that the completed program must parse (see figure 1.1 on page 8). Consider the first such positive example from* $E^+_{grammar_0}$*, that is*

$$e_1^+ = sentence([the, president, has, a, cat], []) \leftarrow .$$

Let the literals that are considered for fundamental explanations – or abducibles – be $noun_phrase/2$ *and* $verb_phrase/2$*.*

One abductive explanation of e_1^+ *in the presence of* $B_{grammar_0}$ *is that the list of words* "the president" *is a* noun phrase*, and that* "has a cat" *is a* verb phrase*. As a set of logical facts the explanation* $\Delta_{e_1^+}$ *is as follows.*

$$\Delta_{e_1^+} = \left\{ \begin{array}{l} noun_phrase([the, president, has, a, cat], [has, a, cat]) \leftarrow \\ verb_phrase([has, a, cat], []) \leftarrow \end{array} \right\}$$

The fundamental explanation $\Delta_{e_1^+}$ *is* extracted *from* $B_{grammar_0}$ *and* e_1^+*.*

Notice that $\Delta_{e_1^+} \notin \mathcal{M}^+(B_{grammar_0})$ *and, that* $B_{grammar_0} \cup \Delta_{e_1^+} \models e_1^+$*. Unfortunately* $\Delta_{e_1^+}$ *does not explain the remaining examples in* E^+*.*

It is possible to consider the explanatory abducibles as negated literals (or atoms from outside the minimum Herbrand model of the background knowledge $\mathcal{M}^-(B)$) that have been selected to support the observation (also an atom from $\mathcal{M}^-(B)$). In a sense the example is *replaced* by the abducibles. Abduction can therefore be understood as the replacing of one atom outside the minimum model of the background knowledge with alternative atoms outside the minimum model. The replacement atoms are, in some way, *suggested* by the background knowledge from the example.

Example 46 (Poor explanations) *Not all atoms outside the minimum model of the background knowledge* $\mathcal{M}^+$ *are suitable explanations for the observation(s).*

Consider the following atoms proposed in $\Delta'_{e_1^+}$. *As an explanation, they pass the test of* $B_{grammar_0} \cup \Delta'_{e_1^+} \models e_1^+$. *However they do not satisfy the "normal" notions of English grammar.*

$$\Delta'_{e_1^+} = \left\{ \begin{array}{l} noun_phrase([the, president, has, a, cat], [the, president, has, a, cat]) \leftarrow \\ verb_phrase([the, president, has, a, cat], []) \leftarrow \end{array} \right\}$$

The next two sections introduce two techniques for performing extraction-case abduction in the presence of an existing background theory from a single positive observation. Both techniques are essentially methods for *deriving* sets of negated atoms from one negated atom and a background program.

3.3 Extraction-case Abduction using Logical Back-propagation

This section investigates the technique of *Logical Back-propagation* (LBP) [Moyle and Muggleton, 1997] for deriving *single negated* atoms. The following restrictions apply:

- The background theory B and, examples E are restricted to sets of *definite* clauses.
- The clauses in B and E are range-restricted or *generative* (see [Muggleton and Feng, 1992] page 288).
- The derivation procedure for finding negated consequences is SLD-resolution (as implemented in PROLOG).

In general, SLD-resolution is only sound for deriving positive consequences from definite programs. Stickel [Stickel, 1992] has developed an extended theorem prover, based on PROLOG using SLD resolution, that both is sound and complete for (amongst other things) deriving negative consequences from (non) definite programs. The soundness and completeness is partially achieved[7] by transforming the original program B. This transformation process adds extra *contra-positive* clauses for each clause $C_i \in B$.

Definition 47 (Contra-positive clause) *Let* $\mathcal{C} = C_0 \leftarrow C_1 \wedge \ldots \wedge C_{j-1} \wedge C_j \wedge C_{j+1} \wedge \ldots \wedge C_n$ *be a definite clause. A* contra-positive clause *for* $\mathcal{C}$ *is* $\neg\mathcal{C}_j = \neg C_j \leftarrow \neg C_0 \wedge C_1 \wedge \ldots \wedge C_{j-1} \wedge C_{j+1} \wedge \ldots \wedge C_n$

[7] Full soundness and completeness requires that both an *occurs check* and a complete search strategy be implemented (e.g. breadth first search).

The negated atoms $\neg C_j$ and $\neg C_0$ can be considered as new predicate symbols whose meaning is the *logical negation* of C_j and C_0 respectively. Note that there is no guarantee (except careful introduction) that if $\neg C_j$ is derivable from a theory $\mathcal{T}$ then C_j is *not derivable* also.

Definite clauses have more than one contra-positive clause. A definite clause with n body literals may have up to n definite contra-positive clauses[8].

Remark 48 (Logical equivalence of contra positive clauses) *If C is a clause and $\neg C_j$ is a contra-positive of C then $C \equiv \neg C_j$*

Definition 49 (Stickelisation) *The process of creating a definite contra-positive clause $\neg \mathcal{C}_j$ from $\mathcal{C}$ is called* stickelisation. *The resultant clause $\neg \mathcal{C}_j$ is called a* stickel clause.

3.3.1 Transforming the Definite Logic Program for LBP

The transformation of definite programs in a PROLOG system present potential problems relating to unlinked variables. This is illustrated in the following example.

Example 50 (Stickelising the `sentence/2` clause) *The clause for sentence/2 from the grammar example (figure 1.1 page 8) can be written as*
(3.2)
$$((\forall S, R) sentence(S, R) \leftarrow (\exists T)(noun_phrase(S, T), verb_phrase(T, R)))$$

The sentence as a PROLOG *program clause is:*
$C =$ `sentence(S,R) :- noun_phrase(S,T), verb_phrase(T,R).`
Stickelising C *to produces* $\neg C$*:*
$\neg C =$ $\neg$`verb_phrase(T,R) :-` $\neg$`sentence(S,R), noun_phrase(S,T).`
The late binding of the (now existential) variable S *is potentially problematic for* PROLOG *in the literal* `noun_phrase/2`. *These problems are as a result of the variable no longer appearing in the head of the clause (now* $\neg$`verb_phrase`*).*

For these reasons the PROLOG implementation of LBP[9] ensures that all stickelised clauses are generative by adding an extra literal – called a *starred* literal – immediately prior to the stickel literal in the body of the clause.

Definition 51 *[Generator or* Star *clause] A* generator *or* star *clause is a clause of the form $p^*(X_1, \ldots, X_n) \leftarrow type_{x1}(X_1), \ldots, type_{xn}(X_n)$. Where p^* has the same arity of $\neg p$ (and hence p); $type_{xi}$ defines the allowable ground function symbols that may be bound to X_i.*

[8] Giving a total of $n + 1$ *logically equivalent* clauses.

[9] LBP is implemented in the ILP system `CProgol5.0` (see [Muggleton and Bryant, 2000]).

Example 52 (The stickelised `sentence/2` clauses for C-Progol) *The* `CProgol5.0` *clauses that are added to the original* ***`sentence/2`*** *clause of the grammar program are:*

$$B^{\neg}_{grammar_0} = \left\{ \begin{array}{lll} \neg\texttt{verb_phrase(T,R)} & \texttt{:-} & \neg\texttt{sentence(S,R)}, \\ & & \texttt{*verb_phrase(T,R)}, \\ & & \texttt{noun_phrase(S,T).} \\ \\ \texttt{*verb_phrase(A,B)} & \texttt{:-} & \texttt{listofwords(A)}, \\ & & \texttt{listofwords(B).} \end{array} \right.$$

The definition for `listofwords/1` *must exist and be generative. For instance see the definition provided in example 14 in the previous chapter (page 26).*

Example 53 (Abducing $\neg$`verb_phrase` from an observation of `sentence/2`) *In this setting the stickelised program is:*

$$B'_{grammar_0} = B_{grammar_0} \cup B^{\neg}_{grammar_0} \tag{3.3}$$

From this background program and the presumed fact

$$\neg e_1^+ = \neg sentence([the, president, has, a, cat], [])$$

it is possible to derive – by SLD resolution – the atom $G_1 = \neg verb_phrase([has, a, cat], [])$.

That is:

$$B'_{grammar_0} \cup \neg e_1^+ \cup \overline{\neg verb_phrase([has, a, cat], [])} \vdash_{SLD} \square \tag{3.4}$$

The SLD derivation is shown in figure 3.1.

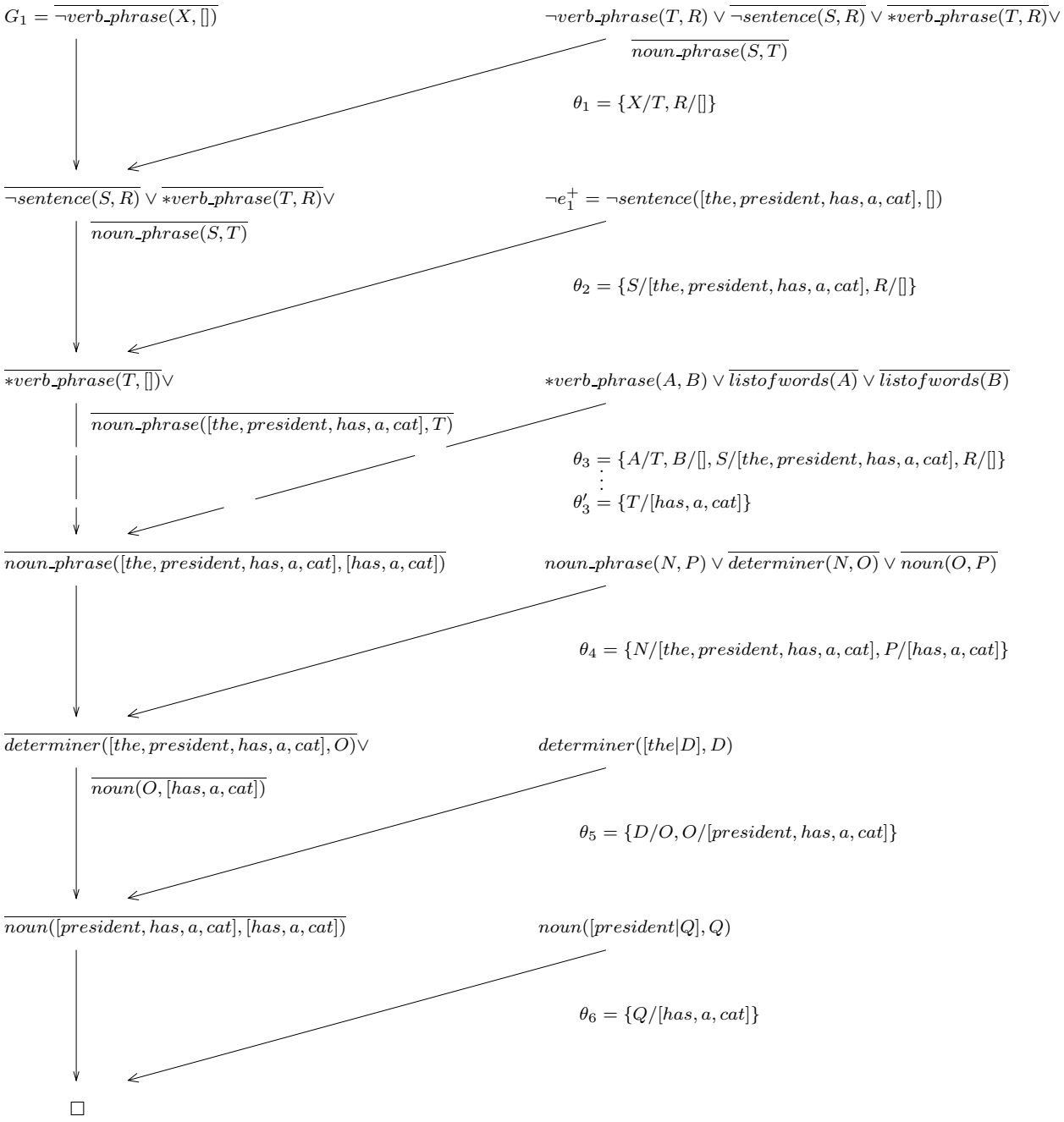


Figure 3.1: The derivation of $\neg verb_phrase([has, a, cat], [])$ by linear resolution from $B'_{grammar_0} \cup \neg e_1^+$ and $\neg sentence([the, president, has, a, cat], [])$. Notice that the dashed arrows indicate some resolution steps are not shown.

$saturate_{LBP}(B, e^+, \mathcal{M}, MSC_{e^+})$: Given a background theory B, a single positive observation e^+, a mode language $\mathcal{M} = \mathcal{M}_{head} \cup \mathcal{M}_{body}$; return a set of most specific clauses MSC_{e^+} such that for each $\perp_i \in MSC_{e^+}$, $B \cup \overline{e^+} \models \overline{\perp_i}$

1. $MSC_{e^+} = \{\}$,
2. If $\mathcal{M}_{head} = \emptyset$ then return MSC_{e^+}, otherwise continue.
3. remove the first mode declaration m from $\mathcal{M}_{head}$
4. if $level(e^+) < level(m)$ go to Step 2 otherwise continue.
5. Produce auxiliary stickel clauses $\neg B$, star clauses $*B$, stickelised example $\neg e^+$, and stickelised mode atom $\neg m$.
6. If $B \cup \neg B \cup \neg e^+ \cup \overline{\neg m} \vdash_{SLD} \square$ then continue otherwise go to Step 2.
7. Use the variable bindings returned in $\neg m$ to produce body literals $b_1, b_2, \ldots$ using Inverse Entailment and $\mathcal{M}_{body}$.

   Form bottom clause $\perp = m \leftarrow b_1, b_2, \ldots$.
8. $MSC_{e^+} = MSC_{e^+} \cup \perp$ go to Step 2.

Figure 3.2: An algorithm to generate most specific clauses from a single observation e^+ and background knowledge B, using Inverse Entailment extended with Logical Back-propagation.

An algorithm for LBP

When an attempt to generalise an observation e^+ does not succeed directly because there is no `modeh` definition for the predicate of e^+; then the Logical Back-propagation algorithm searches for a predicate definition which has: 1) a `modeh` definition; and 2) the predicate is in a lower *level*[10] than e^+. In the previous example the observations are `sentence/2`, and the first predicate to conform to these requirements is `verb_phrase/2` which appears in the body of the clause for `sentence/2`. This clause, and all other clauses that have heads in the levels between the moded predicate and the example predicate, have a Stickel transformation applied. The negated atom (for the example it is ¬`verb_phrase/2`) is tested for derivability. If the atom is derivable then

[10]Here *level* refers to the level mapping of predicates in a logic program (see [Lloyd, 1993] page 83).

the bindings returned are used as part of the start set for Inverse Entailment (see [Muggleton, 1995]), by which a most specific clause is generated. This process is described in more detail in [Muggleton and Bryant, 2000] page 135. A sketch algorithm for generating a set of most specific clauses from a background program, and a single positive example using Logical Back-propagation is shown in figure 3.2.

3.4 Extraction-case Abduction using SOLD-Resolution

SOLD-resolution is also a technique that may be used to derive negated literals for Inverse Entailment. Yamamoto [Yamamoto, 1997] presents SOLD-resolution as a subset of SOL-resolution [Inoue, 1992] which itself is an extension of OL-resolution for clausal logic[11]. SOLD-resolution is capable of deriving a set of *multiple* negative atoms.

Definition 54 (SOLD-derivation) *(From [Yamamoto, 1997].) Let P be a definite program, G be a goal, and R be a computation rule. A* SOLD-derivation *for (P, G) is a finite sequence of quadruplets $\langle G_i, F_i, \theta_i, C_i \rangle$ $(i = 0, 1, 2, \ldots, n)$ which satisfies the following conditions:*

1. *G_i and F_i are goal clauses, θ_i is a substitution, and C_i is a variant of a clausal formula in P, the variables of which are standardized apart by renaming.*
2. *$G_0 = G$ and $F_0 = \square$.*
3. *$G_n = \square$ and $F_n = F$.*
4. *If $G_i = \leftarrow A_1, \ldots, A_k$, $F_i = \leftarrow B_1, \ldots, B_h$ and A_m is the atom selected from G by R, then either of the following two holds:*
   (a) *(Skip) $G_{i+1} = \leftarrow A_1, \ldots, A_{m-1}, A_{m+1}, \ldots, A_k$ and $F_i = \leftarrow B_1, \ldots, B_h, A_m$.*
   (b) *(Resolve) θ_i is an mgu of C_i^+ and A_m, $G_{i+1} = (A_1, \ldots, A_{m-1}, C_i^-, A_{m+1}, \ldots, A_k)\theta_i$, and $F_{i+1} = F_i\theta_i$.*

The above definition can be summarised as:

> If in the process of performing SLD-resolution on goal G in the presence of program P, where some sub-goals are not derivable, then skip the sub-goals, and provided that $\square$ is finally derived, then the set of all skipped sub-goals F is a logical consequence of $P \cup G$.

[11] OL=Ordered Linear; SOL=Skip OL; SOLD=SOL for Definite clauses

The following points are noteworthy.

1. The goal F_n is called the *consequence* of the SOLD-derivation. In the implementation described in section 3.6, the consequence will be termed a **skiplist**.

2. If the skipping rule is not applied when constructing an SOLD-derivation, then every F_i is $\square$, and the derivation can be regarded as an SLD-derivation.

Recall equation 3.1 on page 48. This equation requires the negation of an example to be generated. In the general case such an example, e^+, can be any clause E; that is, not restricted to ground unit clauses. The following definition of a *negated clause* or the *complement* of a clause will be required when negating the example clause.

Definition 55 (Complement) *(Adapted from [Yamamoto, 1997].) The* complement *of* $E = A_0 \leftarrow A_1, \ldots, A_m$ *is a formula*

$$\overline{E} = (\neg A_0 \wedge A_1 \wedge \ldots \wedge A_m)\theta_E$$

where θ_E *is a substitution which replaces each variable in* E *with a Skolem constant symbol*[12].

For a Horn clause E, $\overline{E}^+$ *is a definite program*[13], *and* $\overline{E}^-$ *is a goal clause*[14], *and* $\overline{E} = \overline{E}^+ \cup \overline{E}^-$

SOLD-resolution is used in Inverse Entailment to derive negated atoms from a background program B, and a single positive example e^+ as follows: $SOLD(B, \overline{e^+}) = \overline{H}$. Atoms from the negation of $\overline{H}$ are used to form the heads of bottom clauses.

Proposition 56 (Completeness of SOLD-resolution) *Let* B *be a definite clause and* G *be a goal clause. Then a goal clause* F *is a logical consequence of* $B \cup G$ *iff there is a SOLD-derivation whose consequence subsumes* F.

In other words
$B \cup G \models F$ *iff* $SOLD(B, G) = \overline{E}$ *and* $\overline{E}\theta \subseteq F$.

Proof. Refer to [Yamamoto, 1997] ■

Lemma 57 *Let* A *be a ground atom. Suppose that* E *is not a tautology and that* $B \not\models A$. *Then* $B \wedge \overline{E} \models A$ *iff* $B \wedge \overline{E}^- \models A$

[12] The implementation of abduction in the system `ALECTO` (see chapter 4) obviates the need for Skolem constants by being restricted to ground atoms only.

[13] $\overline{E}^+$ is $\overline{(E)}^+$, and should be read as "the positive part of $\overline{E}$".

[14] $\overline{E}^-$ is $\overline{(E)}^-$, and should be read as "the negative part of $\overline{E}$".

Proof. Refer to [Yamamoto, 1997] ■

The previous lemma states that when performing SOLD-resolution proofs, only the negative component $\overline{(E)}^{-}$ need be considered.

Example 58 (Abducing $\overline{\texttt{verb_phrase}}$ from an observation of `sentence/2`) *In this setting the relationship of interest is:*

$$SOLD(B_{grammar_0}, \overline{e_1^+}) = F \tag{3.5}$$

From this background program and the observation

$$e_1^+ = sentence([the, president, has, a, cat], []),$$

using SOLD-resolution it is possible to derive the goal clause

$$F_1 = \overline{verb_phrase([has, a, cat], [])}.$$

That is:

$$SOLD(B_{grammar_0}, \overline{sentence([the, president, has, a, cat], [])}) = \overline{verb_phrase([has, a, cat], [])} \tag{3.6}$$

The SOLD-derivation of $\overline{verb_phrase([has, a, cat], [])}$ *is shown in figure 3.3.*

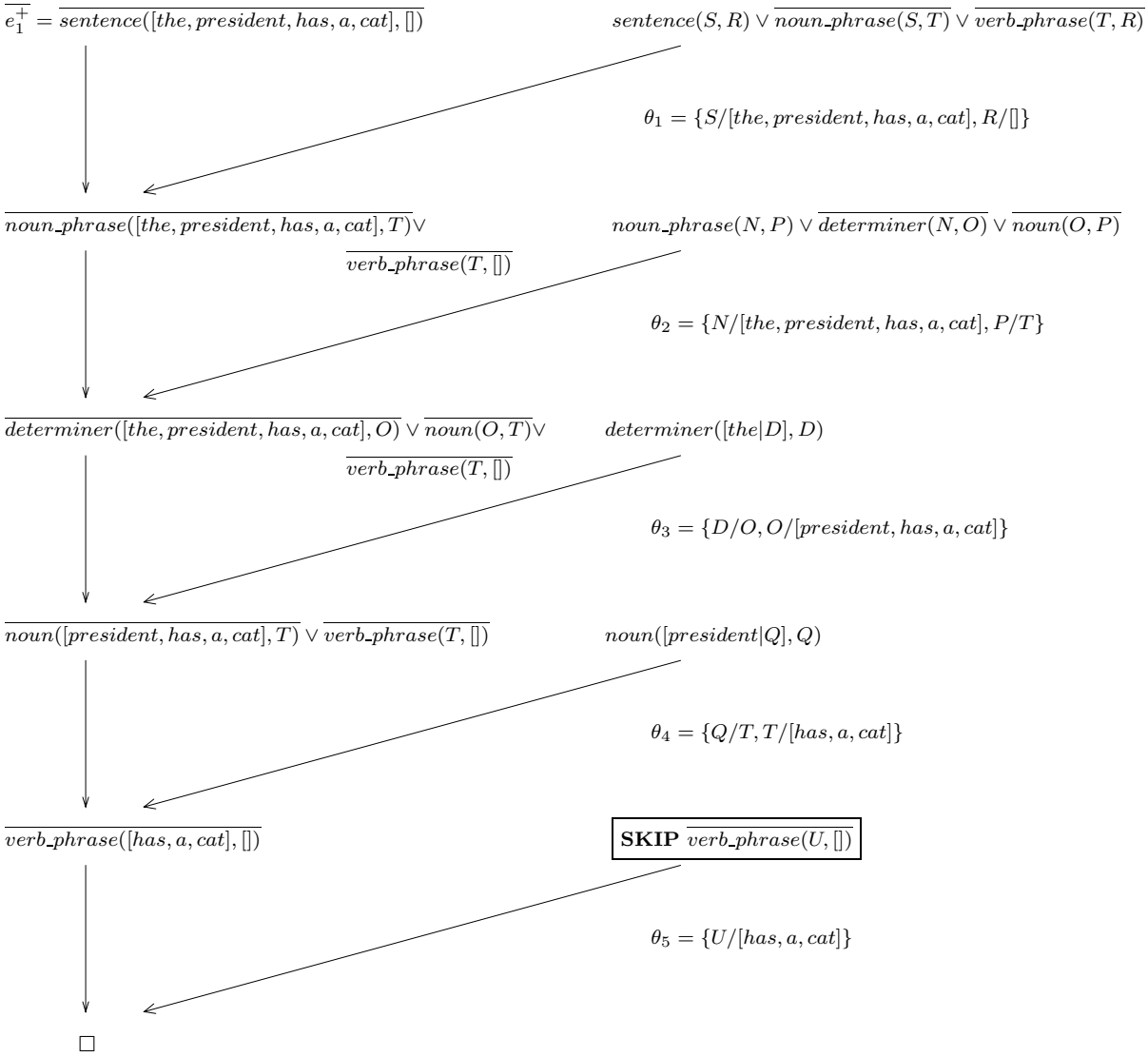


Figure 3.3: The derivation of $\overline{verb_phrase([has, a, cat], [])}$ by SOLD-resolution from $B_{grammar_0}$ and $\overline{e_1^+}$.

3.5 Comparisons of SOLD-resolution and Logical Back-propagation

This section considers the relative merits of SOLD-resolution and LBP in their abilities to derive negated atoms from a background theory and a single negated atom. Every negated atom that is derivable by LBP can also be derived by SOLD-resolution *or* by applying a grounding substitution to the SOLD-resolution derived atom. This is made more precise in the following proposition.

Proposition 59 (SOLD subsumes LBP) *For every negated atom $\overline{h}$ that is correctly derived by Logical Back-propagation from a set of definite clauses B, and a single negated atom $\overline{e^+}$ there exists a negated atom $\overline{g}$, generated by $SOLD(B, \overline{e^+})$ that subsumes $\overline{h}$.*

In other words

If $LBP(B, e^+) = \overline{h}$ then and h is correct then $SOLD(B, \overline{e^+}) = \overline{g}$ and $\overline{g}\theta = \overline{h}$.

Proof. If $LBP(B, e^+) = \overline{h}$ and h is correnct then $B \cup \overline{e^+} \models \overline{h}$. However, by proposition 56 $B \cup \overline{e^+} \models \overline{h}$ iff $SOLD(B, \overline{e^+}) = \overline{g}$ and $\overline{g}\theta \subseteq \overline{h}$. As both g and h are atomic, then $\overline{g}\theta = \overline{h}$. ■

Remark 60 (SOLD-resolution is stronger than LBP) *As LBP only generates* ground *atoms (as guaranteed by typing constraints) it is not capable of generating some atoms produced by SOLD-resolution. In fact, SOLD-resolution is capable of producing* multiple *negative atoms, as illustrated in the following example.*

Example 61 (Simultaneous derivation of $\overline{\texttt{verb_phrase}}$ and $\overline{\texttt{noun_phrase}}$) *Consider the background theory $B_{grammar_1} = \{B_{grammar_0} \backslash noun_phrase/2\}$, that is the original grammar program with the clause for* **noun_phrase/2** *removed. Such a program has no definitions for either* **verb_phrase/2** *or* **noun_phrase/2**.

The SOLD derivation of

$$SOLD(B_{grammar_1}, \overline{sentence([the, president, has, a, cat], [])}) \\ = \overline{verb_phrase(T, [])} \wedge \overline{noun_phrase([the, president, has, a, cat], T)}$$

is shown in figure 3.4.

Compare this to attempting to derive the same form of negated atoms using LBP. In this case it is not possible. Consider the stickelised clauses necessary.

$$B^{\neg\prime}_{grammar_0} = \left\{ \begin{array}{lll} \neg\texttt{verb_phrase(T,R)} & \texttt{:-} & \neg\texttt{sentence(S,R)}, \\ & & \texttt{*verb_phrase(T,R)}, \\ & & \texttt{noun_phrase(S,T)}. \\ \\ \texttt{*verb_phrase(A,B)} & \texttt{:-} & \texttt{listofwords(A)}, \\ & & \texttt{listofwords(B)}. \\ \\ \neg\texttt{noun_phrase(S,T)} & \texttt{:-} & \neg\texttt{sentence(S,R)}, \\ & & \texttt{verb_phrase(T,R)}, \\ & & \texttt{*noun_phrase(S,T)}. \\ \\ \texttt{*noun_phrase(A,B)} & \texttt{:-} & \texttt{listofwords(A)}, \\ & & \texttt{listofwords(B)}. \end{array} \right.$$

When attempting to derive the atom $\neg$ ***verb_phrase/2*** *eventually the literal* $\overline{noun_phrase/2}$ *will be "called". However, in this example there are no definitions for* ***noun_phrase/2****. The similar problem arises when attempting to derive an atom of* $\neg$ ***noun_phrase/2****.*

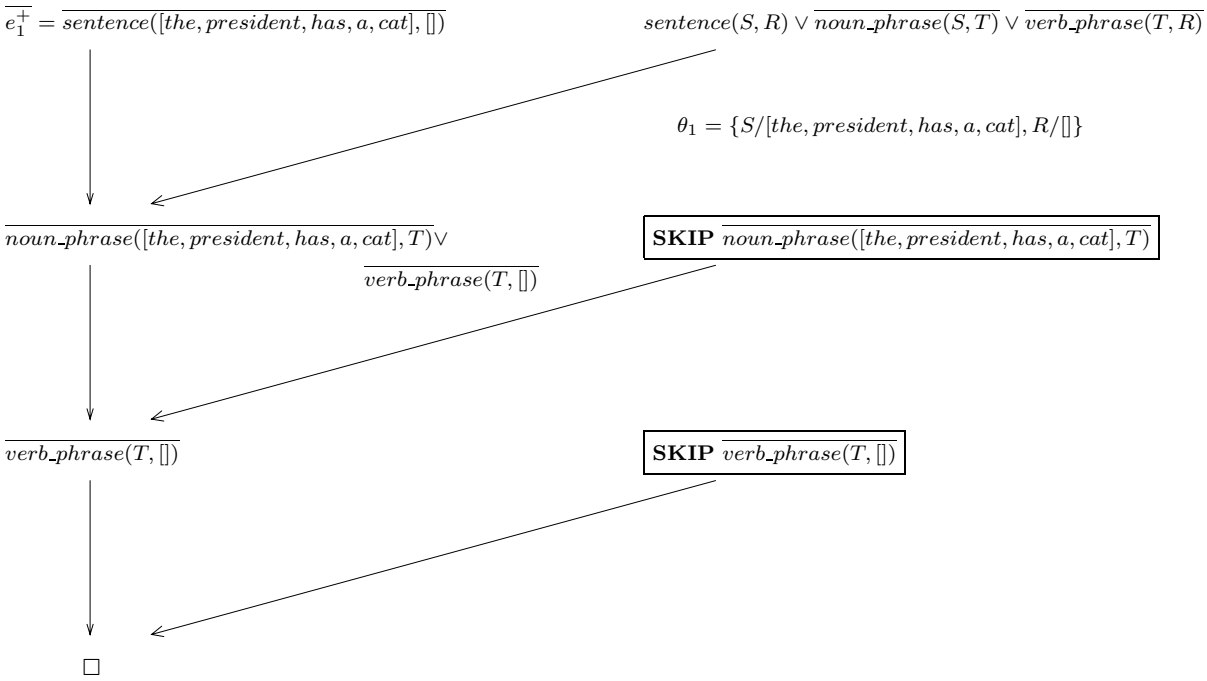


Figure 3.4: The derivation by SOLD-resolution from $B_{grammar_0}$ and $\overline{e_1^+}$ of $\overline{verb_phrase(T, [])} \vee \overline{noun_phrase([the, president, has, a, cat], T)}$.

3.6 Implementing SOLD-resolution

The implementation of the SOLD-resolution interpreter follows directly from the definition of SOLD-resolution given in definition 54. The PROLOG program to perform SOLD-resolution is given in code snippet 3.1.

The interpreter takes as input a definite logic program (implicitly), and a goal clause (explicitly), and returns a goal clause – or skiplist. Alternative skiplists may be returned on backtracking. Notice that this interpreter requires that the predicates which are "skippable" be defined *a-priori* by the `skippable/1` predicate[15].

```
%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%
%%  A SOLD interpreter
%%
%% sold_solve(+Goal,-List_of_negated_atoms).

sold_solve(true,[]):-!.
sold_solve(X,[X]):-
     sold_skip(X).
sold_solve((A,B),Skip3):-!,
     sold_solve(A,Skip1),
     sold_solve(B,Skip2),
     append(Skip1,Skip2,Skip3).
sold_solve(A,[]):-builtin(A),!,A.
sold_solve(A,SkipList):-
     clause(A,B),
     sold_solve(B,SkipList).

sold_skip(X):-
     functor(X,F,N),
     skippable(F/N).

builtin(A):-predicate_property(A,built_in).
```

Code snippet 3.1: A SOLD-resolution interpreter to implement Yamamoto's SOLD-resolution in PROLOG.

The SOLD-interpreter is simple, and provides a sound way of generating goal clauses from a definite program and another goal clause. This is in effect using the background program to *translate* the original goal clause into alternative goal clauses. In Theory Completion single ground negated atoms are required. SOLD-resolution is a procedure that can generate more general goal clauses than simple single negated atoms.

Example 62 (Unconstrained SOLD-resolution) *Consider the setting presented in example 61 and the aim is to find the* skiplist F *where*

$$SOLD(B_{grammar_1}, \overline{sentence([the, president, has, a, cat], [])}) = F$$

There are three *alternative solutions for* F

[15]Defining a predicate as *skippable* is similar to the definition of *abducibles* in ALP.

$$
\begin{array}{lcl}
F_1 & = & \overline{verb_phrase(A,[]) \wedge noun_phrase([the, president, has, a, cat], A)} \\
F_2 & = & \overline{verb(A,B) \wedge noun_phrase(B,[]) \wedge noun_phrase([the, president, has, a, cat], A)} \\
F_3 & = & \overline{noun_phrase(B,[]) \wedge noun_phrase([the, president, has, a, cat], [has|B])}
\end{array}
$$

The generation of these skiplists by the SOLD-interpreter is shown in the output presented in code snippet 3.2. The variables A, and B are existentially quantified.

```
?- sold_solve(sentence([the, president, has, a, cat] ,[]),F).

F = [verb_phrase(A, []),
     noun_phrase([the, president, has, a, cat], A)] ;

F = [verb(A, B),
     noun_phrase(B, []),
     noun_phrase([the, president, has, a, cat], A)] ;

F = [noun_phrase(B, []),
     noun_phrase([the, president, has, a, cat], [has|B])] ;

No
```

Code snippet 3.2: Multiple *skiplists* produced by the SOLD-interpreter with the query $SOLD(B_{grammar_1}, \overline{sentence([the, president, has, a, cat], [])})$.

The previous example highlights that skiplists generated by SOLD-resolution have properties that are undesirable for the efficient practical implementation of Theory Completion systems. Some of these are listed as follows.

1. Skiplists are not always unique.

   It is possible for SOLD-resolution to generate multiple skiplists from the same goal and the same background program.

2. Any predicates may be skipped in producing a skiplist.

   In the above example `verb/2` was defined as "skippable" even though there was a clausal definition for it. Even under these circumstances SOLD-resolution used a literal of `verb/2` for the skiplist (see skiplist F_2 above).

   In the worst case, if all predicates are chosen to "skip" on, completely unconstrained skiplists are returned.

3. Skiplists are not sets of abducibles[16].

   As they are not ground, skiplists are not abducible explanations in the normal sense of abduction, but they are what are called *constructive abduction* (see [Kakas and Riguzzi, 2000]).

[16] All abducible explanations are required to be *ground*.

3.6.1 Further steps towards Inverse Entailment

The preceding description has not provided a means to generate a bottom clause which does not have the same *head predicate* as the *observation* predicate. This section introduces a two step process which uses SOLD-resolution as a basis for such. First, Yamamoto's *skiplists* are generated, and then converted into sets of ground abducibles. Second, the abducible sets are used as *seed* atoms for a form of mode directed inverse entailment.

Generating abducibles from *skiplists*

The responsibility for converting non-ground skiplists to ground abductions can be placed on the *inductive logic programmer*. This can be justified by the fact that domain knowledge may be available to aide in the grounding process[17].

A schema – called **acceptability constraints** – must be provided as part of the background knowledge[18] so that sets of *ground* abducibles are generated from a *non-ground* skip-list. Such schemas are a form of *language constraint* to be applied to abducibles. In what follows the ground skiplist will be called an **h-list**[19].

Example 63 (Generating abducibles from a skiplist) *Consider the skiplist F_1 in example 62 above, containing the existential variable A.*

$$F_1 \quad = \quad \overline{verb_phrase(A, []) \wedge noun_phrase([the, president, has, a, cat], A)}$$

The schema is "Both arguments in a grammar predicate are lists of words. The second argument in a grammar predicate is a tail or *suffix list* of the first argument".

To produce a "valid" h-list from a skiplist it is necessary to provide a grounding relationship for each predicate symbol that appears in the skiplist. For example a grounding relationship for the grammar predicate `verb_phrase/2` *ensures that the second argument is a suffix-list of the first argument, and that both arguments are ground. Code snippet 3.4 provides a* PROLOG *code that encodes this information.*

The first argument to `ground_lit/2` *is the (potentially non-ground) element (atom) of a skiplist, the second is a ground version of the element. The short* PROLOG *program of code snippet 3.3 takes as input a complete skiplist (possibly not ground) and produces a ground h-list.*

[17] Useful domain knowledge is available for use in grounding skiplists in all of the applications of Theory Completion in Part II.

[18] This is a form of *meta-knowledge* similar to the *hypothesis structure constraints* (see section 2.4.2 on page 37).

[19] The term *h-list* is derived from [Yamamoto, 1997].

Notice that for every *skippable predicate a user defined predicate* must *appear in the background knowledge file defining* `ground_lit/2` *for the predicate.*

The valid bindings for non-ground variables, and the resulting abducibles (or h-lists) produced from the skiplist F_1 are as follows[20].

$$
\begin{array}{lcl}
A_{(1,1)} & = & [president, has, a, cat] \\
H_{(1,1)} & = & \{noun_phrase([the, president, has, a, cat], [president, has, a, cat]), \\
 & & verb_phrase([president, has, a, cat], [])\}
\end{array}
$$

$$
\begin{array}{lcl}
A_{(1,2)} & = & [has, a, cat] \\
H_{(1,2)} & = & \{noun_phrase([the, president, has, a, cat], [has, a, cat]), \\
 & & verb_phrase([has, a, cat], [])\}
\end{array}
$$

$$
\begin{array}{lcl}
A_{(1,3)} & = & [a, cat] \\
H_{(1,3)} & = & \{noun_phrase([the, president, has, a, cat], [a, cat]), \\
 & & verb_phrase([a, cat], [])\}
\end{array}
$$

$$
\begin{array}{lcl}
A_{(1,4)} & = & [cat] \\
H_{(1,4)} & = & \{noun_phrase([the, president, has, a, cat], [cat]), \\
 & & verb_phrase([cat], [])\}
\end{array}
$$

```
% skip_to_hlist
% Takes as input a list of atoms (possibly containing variables)
% and produces a list of ground atoms.
skip_to_hlist([],[]).
skip_to_hlist(Skiplist,[Lit1|T]):-
    delete(Lit,Skiplist,Rest),
    ground_lit(Lit,Lit1),
    skip_to_hlist(Rest,T).
```

Code snippet 3.3: Trivial PROLOG code for generating a ground *h-list* from a (potentially) non-ground *skiplist*.

Notice in the preceding example that not all the abducible sets (h-lists) make intuitive sense. For example the atom $verb_phrase([cat], [])$ in the abducible set $H_{(1,4)}$ does not equate to the "normal" understanding of a verb phrase in the English grammar.

Notice also that the use of `ground_lit/2` when implementing SOLD-resolution serves a similar purpose to the *starred* clauses generated by `CProgol5.0` using Logical Back-propagation. The *starred* clauses[21] are generated automatically, whereas definitions for `ground_lit/2` must be provided manually. The *starred* clauses are generated to enforce simple typing constraints,

[20] The first subscript refers to the *skiplist number*, and the second to the *h-list number*.

[21] See definition 51, page 51 for more on *starred* clauses in `CProgol5.0`.

```
ground_lit(noun_phrase(X,Y),noun_phrase(X,Y1)):-
    Y=Y1,
    suffixlist(Y1,X).
ground_lit(verb_phrase(X,Y),verb_phrase(X,Y1)):-
    Y=Y1,
    suffixlist(Y1,X).

suffixlist(T,[_H|T]).
suffixlist(T2,[_H|T1]):-
    suffixlist(T2,T1).
```

Code snippet 3.4: An example of acceptability constraints for `noun_phrase/2` and `verb_phrase/2`. Variables `X` and `Y` are grounded as a side effect of unification.

whereas `ground_lit/2` provides the inductive logic programmer enhanced power to incorporate domain knowledge.

Example 64 (Literal grounding in `ALECTO` and `CProgol5.0`) *Consider the `*verb_phrase/2` clause introduced by* `CProgol5.0` *in example 61 on page 59.*

```
*verb_phrase(A,B) :- listofwords(A), listofwords(B).
```

The body literals `listofwords(A)` and `listofwords(B)` are automatically produced from the mode declaration

```
:- modeh(1,verb_phrase(+listofwords,-listofwords))?.
```

An allowable atom from such a star clause is (both `[has, the, has,the]` and `[cat, a, a, a, a]` are valid lists of words*).*

```
*verb_phrase([has, the, has, the],[cat, a, a, a, a]).
```

Although the `*verb_phrase/2` clause is strictly accurate it does not encode the subtle domain knowledge that "the second argument is a **suffixlist** of the first argument"*. The use of `ground_lit/2` in the implementation allows such domain knowledge to be encoded (see example 63 and code snippet 3.4).*

Generating Most Specific Clauses from abducibles

The sets of abducibles – or h-lists – provide a basis for generating most specific clauses. The h-lists can be used to *replace* the original seed observation e^+ because the following holds.

$$B \cup \Delta_i \models e^+$$

Now $\forall a_j \in \Delta_i$, mode directed Inverse Entailment (see section 2.2.2 on page 25) can be used to generate a most specific clause $\bot_{(B,a_j)}$ such that:

$$B \cup \bot_{(B,a_j)} \models a_j$$

Considering all of the n most specific clauses (one for each element of the h-list), then the following holds.

$$\begin{aligned} B \cup \bot_{(B,a_1)} \cup \ldots \cup \bot_{(B,a_n)} &\models \Delta \\ &\models e^+ \end{aligned}$$

For any clause $C_{\bot \succeq_{(B,a_j)}}$ in the θ-subsumption lattice (see page 32 chapter 2) between $\square$ and $\bot_{(B,a_j)}$ then

$$B \cup C_{\bot \succeq_{(B,a_j)}} \models a_j$$

The collection of all single clauses that subsume individual bottom clauses

$$\begin{aligned} B \cup C_{\bot \succeq_{(B,a_1)}} \cup \ldots \cup C_{\bot \succeq_{(B,a_n)}} &\models \Delta \\ &\models e^+ \end{aligned}$$

From the above it is possible to formulate an algorithm for generating a set of most specific clauses based on SOLD-resolution. Such an algorithm is shown in figure 3.5. Examples of most specific clauses output from the implementation of the algorithm in the `ALECTO` system are shown in Code snippets 3.5 and 3.6 (on page 68).

$saturate_{SOLD}(B, e^+, \mathcal{M}, \mathcal{A}, I_{acceptability}, S_{max}, H_{max}, MSC_{e^+})$: Given background theory B, a single positive observation e^+, a mode language $\mathcal{M} = \mathcal{M}_{head} \cup \mathcal{M}_{body}$, a set of predicates defined as *skippable* $\mathcal{A}$, set of grounding acceptability constraints $I_{acceptability}$, and upper bounds on the number of skiplists and hlists S_{max}, H_{max}, returns a set of most specific clauses MSC_{e^+} such that for each $\perp_i \in MSC_{e^+}$, $B \cup \overline{e^+} \models \overline{\perp_i}$

1. $MSC_{e^+} = \{\}$; $\Delta = \{\}$,
2. Generate skiplists with respect to B and the predicates in $\mathcal{A}$: $\mathcal{S}_{B,e^+} = \{S_i | SOLD(B, \overline{e^+}) = S_i \wedge i \leq S_{max}\}$
3. Generate hlists: $\mathcal{H}_{B,e^+} = \{H_{ij} | H_i \text{ satisfies } I_{acceptability} \wedge j \leq H_{max}\}$
4. Initialise abduced atoms: $\Delta = \{a | a \in H_{ij} \wedge i \leq S_{max} \wedge j \leq H_{max}\}$
5. If $\Delta = \emptyset$ then return MSC_{e^+}, otherwise continue.
6. remove the first atom m from Δ
7. Use the variable bindings in m to produce body literals $b_1, b_2, \ldots$ using Inverse Entailment and $\mathcal{M}_{body}$.

   Form bottom clause $\perp = m \leftarrow b_1, b_2, \ldots$.
8. $MSC_{e^+} = MSC_{e^+} \cup \perp$ go to Step 5.

Figure 3.5: An algorithm to generate most specific clauses from a single observation e^+ and background knowledge B, using Inverse Entailment extended with SOLD-resolution.

```
[abduced sat] [abduced observation used] [verb_phrase([has,a,cat],[])]
[Raw bottom] [Number of raw lits] [12]
[bottom clause]
verb_phrase(A,B) :-
   verb(A,C), verb(A,C), cons(A,has,C), determiner(C,D),
   determiner(C,D), noun_phrase(C,B), noun_phrase(C,B), cons(C,a,D),
   noun(D,B), noun(D,B), cons(D,cat,B).
[repeated literals] [4/12]
[bottom clause]
verb_phrase(A,B) :-
   verb(A,C), cons(A,has,C), determiner(C,D), noun_phrase(C,B),
   cons(C,a,D), noun(D,B), cons(D,cat,B).
[literals] [8]
[saturation time] [0.009]
```

Code snippet 3.5: A bottom clause with head `verb_phrase/2` derived using SOLD-resolution, and the observation $sentence([the, president, has, a, cat], [])$. Note that `[repeated literals]` refers to literals that appear in the bottom clause more than once, and are then removed by `ALEPH` when finalising the bottom clause.

```
[abduced sat] [abduced observation used]
        [noun_phrase([the,president,has,a,cat],[has,a,cat])]
[Raw bottom] [Number of raw lits] [12]
[bottom clause]
noun_phrase(A,B) :-
   determiner(A,C), determiner(A,C), cons(A,the,C), noun(C,B),
   noun(C,B), cons(C,president,B), verb_phrase(B,D), verb_phrase(B,D),
   verb(B,E), verb(B,E), cons(B,has,E).
[repeated literals] [4/12]
[bottom clause]
noun_phrase(A,B) :-
   determiner(A,C), cons(A,the,C), noun(C,B), cons(C,president,B),
   verb_phrase(B,D), verb(B,E), cons(B,has,E).
[literals] [8]
[saturation time] [0.009]
```

Code snippet 3.6: A bottom clause with head `noun_phrase/2` derived using SOLD-resolution, and the observation $sentence([the, president, has, a, cat], [])$. Note that `[repeated literals]` refers to literals that appear in the bottom clause more than once, and are then removed by `ALEPH` when finalising the bottom clause.

3.7 Discussion

Theory Completion is a specialisation of Theory Revision, and a generalisation of "classical" ILP. One technical concern for Theory Completion is the derivation of most-specific clauses which have differing predicates to those provided by the observations. In "classical" ILP methods are used to attempt to "glue" (ground) literals from the background theory onto the head of an existing observation (in some manner [Muggleton, 1998]). In Theory Completion, the aim is to "glue" literals from the background theory not onto the head of an observation, but onto the head of some other literal – which is related to the observation in some way via the background theory – but is not itself derivable from the background theory.

This leads to the question *"Which literal(s) should be selected from those that are not derivable from the background theory?"*. This can be seen as the extraction-case abductive task of selecting an abductive explanation for the example in the presence of background theory. Two techniques for producing abductive explanations within a standard Logic Programming framework were investigated: Logical Back-Propagation [Moyle and Muggleton, 1997], and SOLD-resolution [Yamamoto, 1997].

Logical Back-propagation is a procedure for deriving a single negated atom in the presence of background theory. The technique is similar to those used in *constructive negation* (see for example [Van Le, 1993] page 357). Logical Back-propagation requires the transformation of the existing observation(s) and background theory by the addition of Stickel literals, Stickel clauses, and generator or star clauses. Logical Back-propagation was used to generate most specific clauses with different predicate heads to those of the observations. However, LBP was not able to generate bottom clauses when there was not a single atom explanation[22].

Yamamoto's SOLD-Resolution [Yamamoto, 1997] is capable of producing multi-atom explanations. It is similar to "residue" techniques for performing abduction[23]. SOLD-resolution has the additional advantage over LBP that no transformations are required of either the observations nor the background program. It does, however, require a special purpose SOLD-resolution interpreter.

SOLD-Resolution takes an observation (as a goal clause), and the background program, and produces another goal clause – or skiplist. This skiplist may require extra manipulation to make all of its elements ground – this process generates an h-list of ground elements. Each of the elements in an h-list can be used as a seed atom for generating a most specific clause, having a head predicate which differs from the predicate of the original example.

[22]This was worked around in [Moyle and Muggleton, 1997] and [Moyle, 1999] by combining two predicates that needed abducing into a single predicate.

[23]Residues are the literals left behind in resolution proofs [Finger and Genesereth, 1985].

Table 3.1: A comparison of some of the features of the two techniques for producing negated atoms: Logical Back-propagation, and SOLD-resolution.

Feature	LBP	SOLD-resolution
Produces single negated literals	Yes	Yes
Produces non-ground negated literals	No	Yes
Produces multiple non-ground negated literals	No	Yes
Requires transformations to background theory	Yes	No
Requires special purpose meta-interpreter	No	Yes
Requires acceptability constraints	No	Yes

A summary of a comparison of the features of LBP and SOLD-resolution is presented in table 3.1.

3.8 Summary

The main focus of this chapter was to investigate techniques for proposing hypotheses that can be used to complete theories. The ability to generate hypotheses defined by predicates that differ from the predicates of the examples is required.

The main points in this chapter are:

- Theory Completion is a specialisation of Theory Revision in which clauses may only be added (not revised).
- Theory Completion differs from the normal ILP setting of Observational Predicate Learning in that hypotheses generated may have heads different from the examples.
- To generate hypotheses in the non-OPL setting, Inverse Entailment must be extended so that alternative (negated) atoms to the examples are produced for generating bottom clauses.
- Two possible approaches can be considered for deriving negated atoms from examples and background theory: 1) Logical Back-propagation, and 2) SOLD-resolution.
- The methods for deriving negated atoms from examples and background theory are forms of *extraction-case abduction*.

- While LBP is restricted to producing single ground negated atoms, SOLD-resolution is capable of producing *sets* of non-ground atoms, for each example.

The techniques described in this chapter (and those in chapter 2) form the basis of a Theory Completion system. In the next chapter an implementation, of a Theory Completion system using SOLD-resolution and Inverse Entailment, is described.

Chapter 4

ALECTO: An ILP system for Theory Completion

This chapter describes the implementation of the Theory Completion system ALECTO[1]. ALECTO builds on techniques described in the previous chapters, including those for generating most specific clauses, and by providing mechanisms for selecting "good" hypotheses that subsume these most specific clauses. The implementation of ALECTO focuses on two aspects: 1) extending the existing ILP system ALEPH; and 2) providing a positive only learning basis for selecting between competing hypotheses. The ALECTO system is compared to the Theory Completion system CProgol5.0 [Muggleton and Bryant, 2000], and the Abductive Logic Program learner ACL[2] and MACL [Kakas and Riguzzi, 2000].

4.1 Introduction

The foundations laid in the previous chapter allow the generation of bottom clauses which have head predicates different from the heads of the observations, via extraction-case abduction. Saturation is only one of the tasks that a mode directed Inverse Entailment ILP system must perform (see chapter 2). Other required tasks include searching through lattices of alternative clauses, scoring each of the clauses, and somehow selecting the "best" clause.

[1] **A**bductive **LE**arning by **C**ompleting **T**heories from **O**bservations. In Greek mythology Alecto was one of the three Erinyes (also known as the Furies); the three spirits, who were sent to wreak vengeance upon and torment the guilty until they were driven mad and killed themselves in despair. Their names were Alecto (Unresting), Megaera (Jealous), and Tisiphone (Avenger). When, in Hades, the dead repented and sought supplication of the Erinyes, the three became the Eumenides or "Kindly Ones".

[2] ACL learns *Abductive Logic Programs* and in the process performs both learning and abduction.

In the Theory Completion setting the task of searching the subsumption lattice between the top element ($\square$) and the bottom clause ($\perp$) can be employed almost unchanged from the "classical" ILP setting. Here the notions of *refinement operators*, and *searching* the lattice carry over directly from those presented in chapter 2 (see page 32).

The task of scoring clauses, however, must be altered from the mechanisms presented in chapter 2. The main complication in the Theory Completion setting is coping with **multiple-predicate learning** [De Raedt *et al.*, 1993]. One method for coping with multiple-predicate learning is to *provisionally accept* abduced atoms into the minimum Herbrand model of the final program, *while* the lattice for a clause defining a single predicate symbol is being searched.

This chapter introduces the implementation of the Theory Completion system `ALECTO`, and sketches the algorithms involved. In particular the methods for multiple-predicate learning, and theory scoring/selection are presented. The `ALECTO` system is compared to `CProgol5.0` [Muggleton and Bryant, 2000], which is an ILP system which has common Theory Completion objectives to the work here.

4.1.1 Extending the ILP system `ALEPH`

The approach taken was to extend the already developed ILP system `ALEPH`[3] [Srinivasan, 2000]. The `ALEPH` system was chosen as a base system for having the following characteristics.

- **Stability**: `ALEPH` has been constantly developed since 1993. A consequence of this development is that the system is stable.
- **Speed**: `ALEPH` is written in PROLOG, and tailored specifically to the `Yap` [Costa *et al.*, 2000] PROLOG engine. The development of `Yap` has focused on optimising the execution of PROLOG programs on a variety of platforms.
- **Openness**: `ALEPH` is distributed as PROLOG source code. This allows for its algorithms to be easily studied.
- **Features**: `ALEPH` provides many basic features used in ILP. In particular the implementation of theory completion makes use of the following `ALEPH` features: 1) those for saturating an atom to produce a most-specific clause, 2) θ-subsumption based searching, and 3) customisable scoring functions.

These features make `ALEPH` suitable as a platform for implementing the Theory Revision system `ALECTO`.

[3] `ALEPH` is the successor program to `P-Progol`.

4.2 Transformations of observations by generating Abductive explanations

This section considers practical techniques for transforming observations into abductive explanations, in particular those that avoid the problems of *unconstrained abduction* mentioned in chapter 3, section 3.6. Unconstrained abduction (potentially) allows any set of atoms in the Herbrand Base of the program to be "proposed" as an explanation for the (unexplained) observation. An implementation of Theory Completion requires methods for *restricting abduction*.

In this work *plausible* explanations are generated using SOLD-resolution. The approach developed requires input from the inductive logic programmer to constrain explanations in the following ways: 1) ensuring that explanations are ground; and 2) limiting the predicate symbols allowed in the explanations. This is achieved by a mode language for abducibles, constraints for the acceptability of abductive explanations, and constraints on the cardinality of the abductive explanations.

A mode language for abducibles

One technique for constraining the number of generated abductive explanations is to restrict the explanations so that they only contain predicate symbols of interest. In the *Progol* family of ILP systems, the inductive logic programmer communicates which predicate symbols are "interesting" by specifying to the system a set of *mode declarations* (see section 2.2.3 page 30). A similar model is followed in the specification of abducibles to the system `ALECTO`.

Definition 65 (Abducible) *A predicate symbol* P *with arity A is defined as* abducible *in* `ALECTO` *if the* background knowledge file *(`.b`) contains the directive* $:-abducible(P/A)$.

Example 66 (Abducible definitions for `verb_phrase` and `noun_phrase`.) *The abducible directives for completing the missing* ***`verb_phrase`*** *and* ***`noun_phrase`*** *predicates that appear in the background knowledge (`.b`) file are:*

```
:- abducible(verb_phrase/2).
:- abducible(noun_phrase/2).
```

The effect of the abducible definitions is that any skiplist generated from a single observation will not contain any predicates other than those defined as abducible. Recall, however, that the skiplists cannot be considered an explanation unless all elements are ground, for which there is no general guarantee.

Acceptability constraints for abducibles

Following the generation of a skiplist – of pre-defined abducible predicates – it is then processed into (none or more) "acceptable" sets of abducibles, or h-lists. Any domain "constraint" may be applied in the transformation – however, all elements of a h-list must be *ground*. This must also be determined by the inductive logic programmer. More detail on acceptability constraints and their implementation is shown in chapter 3 (see the example provided on page 63).

Cardinality constraints for abductive explanations

A further constraint can be specified that limits the number of atoms in an abductive explanation. This is communicated to `ALECTO` by setting an upper bound on the number of such atoms permitted in an h-list.

Example 67 (Size of explanation for the grammar problem) *The directive for restricting the number of different predicate symbols that may appear in an abductive explanation in the grammar problem is:*

```
:- set(max_hlist_pred_symbol,2).
```

Notice that the h-lists in code snippet 3.3 in chapter 3 page 64 conform to this restriction.

4.3 Transforming observations

The process of generating (constrained) abductive explanations for an observation can be seen as a process of *transforming* the observation into an alternative set of atoms. This section outlines the motivation behind such transformations.

Each h-list $H_{(i,j,k)}$ can be considered as a ground explanation for an observation e_i^+ in the abductive sense.

$$B \cup H_{(i,j,k)} \models e_i^+$$

By selecting a *single* h-list for *each* observation it is possible to provide a *complete* explanation for the entire set of observations.

$$B \cup H_{(1,\ldots)} \cup H_{(2,\ldots)} \cup \ldots \cup H_{(n,\ldots)} \models e_1^+ \wedge \ldots \wedge e_n^+$$

Each observation may have more than one h-list – or ground abductive explanation. Such a set of explanations for an entire set of observations is termed a *crosslist*.

Definition 68 (Crosslist) *A* crosslist χ *for a set of examples* E^+ *is a set of h-lists, where each example is explained*[4] *by* one *h-list.*

(4.1)

$$\chi = \left\{ H_{(i,\ldots)} | \forall e_i^+ \in E^+ \text{ and } (SOLD(B, \overline{e_i^+}) = \overline{H_{(i,\ldots)}} \text{ provided that } B \cup H_{(i,\ldots)} \models o_i^+) \right\}$$

The relationship between alternative crosslists, and observations are shown in figure 4.1. There is an obvious exponential explosion in the number of possible crosslists for any given set of observations.

Remark 69 (Upper bound on $|\chi|$**)** *Let* m *be the number of observations (i.e. the number of elements in* E^+*),* s *be the maximum finite number of skiplists for any one observation, and* h *be the maximum finite number of h-lists for any one skiplist, then an upper bound on the number of possible crosslists,* $|\chi|$ *is given by*

(4.2)
$$|\chi|_{max} = (s \times h)^m$$

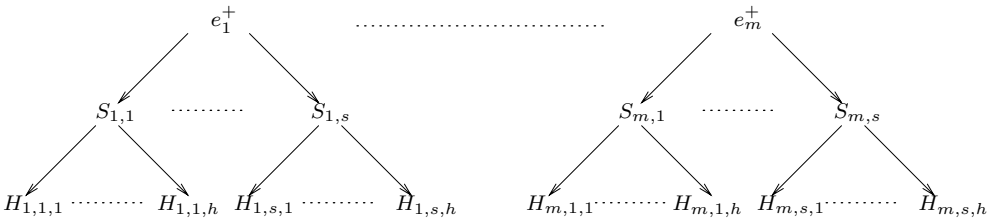


Figure 4.1: The transformation from observations to *h-lists.*

An algorithm that generates sets of crosslists from background knowledge B, and examples E^+ is shown in figure 4.2.

[4]It may be the case in the presence of background theory B that one h-list may explain more than one example.

generate_crosslists$(B, E^+, \Delta, I_{acc}, S_{max}, H_{max}, |\chi|_{max}, \mathcal{X})$: Given background theory B, (positive) observations E^+, abducible predicate definitions Δ, acceptability constraints on abductive explanations I_{acc}, user defined maximum numbers of skip lists, h-lists and crosslists (S_{max}, H_{max}, and $|\chi|_{max}$); return a set of at most $|\chi|_{max}$ crosslists $\mathcal{X}$.

1. generate at most S_{max} skiplists for each observation
   for $1 \leq i \leq |E^+|$, for $1 \leq j \leq S_{max}$ and $e_i^+ \in E^+$
   $\mathcal{S} = \{S_{i,j} | SOLD(B, \overline{e_i^+} = \overline{S_{i,j}})\}$ (with respect to any appropriate acceptability constraints in I_{acc}). If an observation has no skiplist, then accept the observation itself.

2. generate at most H_{max} h-lists from each skiplist
   for $1 \leq m \leq H_{max}$
   $\mathcal{H} = \{H_{i,j,m} | skip_to_hlist(S_{i,j}, H_{i,j,m})\}$ (with respect to any appropriate acceptability constraints in I_{acc}).

3. generate at most $|\chi|_{max}$ crosslists
   for $1 \leq i \leq |E^+|$, for $1 \leq j \leq S_{max}$ and $e_i^+ \in E^+$

$$\chi_k = \bigcup_{j=1}^{m} H_j$$

4. Return χ_k

Figure 4.2: An algorithm for generating *crosslists* from background knowledge B, and observations E^+.

Example 70 (A crosslist for the grammar problem.) *Returning to the grammar problem. With the* `ALECTO` *declarations for abducibles given in example 66, and the constraints for ensuring all atoms in an abductive explanation as shown in code snippet 3.4 in chapter 3 page 65 the following is a crosslists.*

$$\chi_1 = \left\{ \begin{array}{l} verb_phrase([president, has, the, president], []), verb_phrase([president, has, the, cat], []), \\ verb_phrase([president, has, a, president], []), verb_phrase([president, has, a, cat], []), \\ verb_phrase([cat, has, the, president], []), verb_phrase([cat, has, the, cat], []), \\ verb_phrase([cat, has, a, president], []), verb_phrase([cat, has, a, cat], []), \\ noun_phrase([the, president, has, the, president], [president, has, the, president]), \\ noun_phrase([the, president, has, the, cat], [president, has, the, cat]), \\ noun_phrase([the, president, has, a, president], [president, has, a, president]), \\ noun_phrase([the, president, has, a, cat], [president, has, a, cat]), \\ noun_phrase([the, cat, has, the, president], [cat, has, the, president]), \\ noun_phrase([the, cat, has, the, cat], [cat, has, the, cat]), \\ noun_phrase([the, cat, has, a, president], [cat, has, a, president]), \\ noun_phrase([the, cat, has, a, cat], [cat, has, a, cat]), \\ noun_phrase([a, president, has, the, president], [president, has, the, president]), \\ noun_phrase([a, president, has, the, cat], [president, has, the, cat]), \\ noun_phrase([a, president, has, a, president], [president, has, a, president]), \\ noun_phrase([a, president, has, a, cat], [president, has, a, cat]), \\ noun_phrase([a, cat, has, the, president], [cat, has, the, president]), \\ noun_phrase([a, cat, has, the, cat], [cat, has, the, cat]), \\ noun_phrase([a, cat, has, a, president], [cat, has, a, president]), \\ noun_phrase([a, cat, has, a, cat], [cat, has, a, cat]) \end{array} \right\}$$

Generalising crosslists and induction

The generation of *crosslists* is not a form of *induction*. However, *generalisations* of atoms in the crosslist may be considered as induction. Revisiting the setting of ILP, the initial motivation is the condition of Prior Necessity (see page 20 chapter 2):

$$B \not\models E^+$$

with the aim to produce a set of clauses H which can be considered a generalisation of the observation E^+ (Posterior Satisfiability) (see page 2.1.1 chapter 2):

$$B \cup H \models E^+$$

A crosslist χ is an abductive explanation for all the examples E^+ such that:

$$B \cup \chi \models E^+$$

If a generalisation G of χ could be produced where:

$$B \cup G \models \chi$$

then G would also be a generalisation of E^+.

4.3.1 Justifications for the transformation of observations into crosslists

The crosslist is a *complete explanation* for a set of observations. It is, however, only one possible strategy in creating explanations for sets of observations. Two alternative strategies[5] are 1) the *intersection* approach, and 2) the *union* approach.

The union approach is the generation of all restricted abductive explanations and taking the atoms that are in the *union* of these explanations to be the "desired" explanation. The intersection approach differs in that the "desired" explanation consists of the atoms that are in the *intersection* of all the restricted abductive explanations. However, it is not clear how to deal with the problematic case when the intersection is empty. Wellner [Wellner, 1999] (page 45) advocates the use of a *union* of all possible abductive explanations to represent the set of observations[6].

A third approach, which can be seen to lie between the union and intersection approach takes the union of precisely *one* abductive explanation per observation. This approach has a theoretical basis in positive (example) only learning.

[5] Personal communication with Francesca Toni, December 1999.

[6] Benjamin Wellner calls the process of selecting a desired explanation from alternatives "Explanation Refinement". In more complicated settings (for example when explanations of *negative* observations must be considered) alternative refinement schemes have been proposed.

Positive-only learning

The setting for learning from positive-only examples was outlined in chapter 2 (section 2.4.3, page 39). The aim of such a scheme is to prevent *over-generalisations* occurring when only positive data is available to be learnt from. One such scheme is to find an hypothesis scoring function that can be used to tradeoff between *theory generality* and *theory size* during search.

In Muggleton [Muggleton, 1996] a Bayes' positive example framework is used to derive an algorithm which maximises an approximation of the Bayes' posterior probability for an hypothesis. This probability score is then used to rank order the clauses in the search between the most general clause and the most specific clause (i.e. between top and bottom) (see section 2.3.1 page 32).

One assumption that Muggleton's work relies on is that for any conjectured *theory* to have a positive Bayes' posterior probability, all of the observations must be derivable by the conjectured theory. That is, a non-zero likelihood probability $p(H|E^+)$ exists *only if* $\forall e_i^+ \in E^+$, H covers e_i^+. This constraint provides a basis for selecting exactly one abductive explanation for each observation.

By definition a crosslist (see equation 4.1) is a theory that satisfies Muggleton's assumption, as are any generalisations of a crosslist.

4.4 Controlling the search for theories

The *extraction* of crosslists χ from observations E^+ and background theory B changes the focus from directly generalising the observations, to generalising them indirectly by generalising the crosslists. At the conclusion of the process some theory H is desired where $B \cup H \models E^+$.

Such a process presents two difficulties: 1) learning recursive multi-predicate theories; and 2) scoring potential hypotheses in the presence of positive-only observations.

`ALECTO` is capable of generating recursive theories requiring multiple predicate learning. In some cases the generation of a clause in an as yet incomplete theory may rely on literals that do not have a definition. The implementation `ALECTO` is able to cope with such incomplete hypotheses. When searching the clauses in the θ-subsumption lattice, `ALECTO` provides a score for each clause in the absence of any negative observations via a positive-only scoring scheme.

4.4.1 Coping with incomplete hypotheses

When generalising a clause from an atom in a crosslist it may be necessary to test a clause containing a literal when that literal does not *yet have a definition*. Consider the following example.

Example 71 (The refinements of `verb_phrase/2`) *The bottom clause in code snippet 3.5, page 68 is*

$$\bot_{(B,e_1^+)} = \left\{ \begin{array}{ll} verb_phrase(A,B) & \leftarrow verb(A,C), cons(A,has,C), \\ & determiner(C,D), noun_phrase(C,B), \\ & cons(C,a,D), noun(D,B), cons(D,cat,B). \end{array} \right\}$$

The clause $H_* = verb_phrase(A,B) \leftarrow verb(A,C), noun_phrase(C,B).$ *is a valid refinement in the* θ*-subsumption lattice between* $\square$ *and* $\bot_{(B,e_1^+)}$.

However, when testing H_**, it will not cover* **any** *observations because the last literal* $noun_phrase(C,B)$ *will not be provable, as there are no definitions for* ***noun_phrase/2*** *in the background theory*[7].

To overcome the situation described in the preceding example, `ALECTO` implements an *abductive coverage* policy.

Definition 72 (Abductive coverage) *An atom* a *from crosslist* χ *is said to be abductively covered relative to background theory* B *by clause* C *when:*

$$B \cup \chi \setminus \{a\} \cup C \models a$$

The definition can be paraphrased as "when scoring any clause it may be assumed that all other atoms in the crosslist are *true*".

4.4.2 An abductive Positive-Data Only Scoring function

The scoring function employed in `ALECTO` is a modified version of the positive-data only function $f_r(H) = r(1-g(H))-sz(H)$ implemented in [Muggleton, 1996] (see also the introduction to pos-only scoring in section 2.4.3), where $f_r(H)$ is the score for hypothesis H, r is the size of a random population of instances, $g(H)$ is the generality of H, and $sz(H)$ is the size of the hypothesis measured in the number of literals.

Definition 73 (Abductive generality of an hypothesis) *The* abductive generality *of a clause* H*,* $g_\Delta(H)$ *is defined as the Laplace correction of the total number of random observations that are* abductively covered *when* H *is included with the background theory and a crosslist* χ*, divided by the total number of observations in the random set.*

Let C *be the observations covered in the random observation set* E^r *by* $B \cup H \cup \chi$*. That is* $C = \{e_j \in E^r | B \cup H_i \cup \chi \models e_j\}$[8].

Then the generality of H_i *is given by*

$$g_\Delta(H_i) = \frac{|C|+1}{|E^r|+2}$$

[7] This is the reason for Theory Completion in the first place.

[8] This is actually implemented with a depth bounded theorem prover $B \cup H_i \cup \chi \cup \overline{e_j} \vdash_{SLD_h} \square$.

Definition 74 (Abductive size of an hypothesis.) *The* abductive size *of an hypothesis H_i, $sz_\Delta(H_i)$ is measured (crudely) by the number of literals in the hypothesis. The components that make up an hypothesis are 1) any partial hypothesis (set of clauses) $H_{i-1} = \{C_1, C_2, \ldots, C_{i-1}\}$ provisionally accepted; 2) the set of uncovered atoms in the crosslist*

$$\chi_{\not\models} = \{a_k \in \chi | B \cup \{\chi \setminus a_k\} \cup H_{i-1} \cup C_i \not\models a_k\};$$

and 3) the size of the clause being scored C_i.

The abductive size of an hypothesis is the sum of all the literal sizes of the three components:

$sz_\Delta(H_i) = sz(H_{i-1}) + sz(\chi_{\not\models}) + sz(C_i)$

Definition 75 (The abductive score of an hypothesis.) *The* abductive score *$f_\Delta(H)$, of an hypothesis H, is defined as:*

$f_\Delta(H) = r(1 - g_\Delta(H)) - sz_\Delta(H)$

where r is the size of the random population of instances, $g_\Delta(H)$ is the abductive generality of H over the instances, and $sz_\Delta(H)$ is the abductive size of the hypothesis.

4.5 The ALECTO algorithm

The ALECTO algorithm can be summarised as: 1) abductively translate observational predicate examples into ground explanations expressed in the target predicates; 2) generalise the atoms in the explanations to produce a clause, that when added to the original background theory, will allow the derivation of the original examples.

The ALECTO algorithm is shown in figure 4.6. This is simply a *wrapper* program that utilises many functions from the ILP system ALEPH. The main functions that make up ALECTO are 1) *generate_crosslists()* shown in figure 4.2 (page 77) for transforming the observations into abductive explanations; and 2) *generalise_crosslist()* (adapted from [Srinivasan, 2000]) shown in figure 4.3 for generalising the abductive explanations into clausal theories. The return of the "best" clause can be achieved by the algorithm *search* presented in figure 4.4 (page 84).

generalise_crosslist$(B, E^+, I_{\mathcal{L}}, \mathcal{L}, \chi, E^r)$: Given background theory B, a finite set of observations E^+, hypothesis constraints $I_{\mathcal{L}}$, a ground crosslist χ, a set of random observations E^r, returns a theory H in the language $\mathcal{L}$ such that $B \cup H$ explains E^+.

1. $i = 0$;
2. $\Delta_i^+ = \chi, H_i = \emptyset$;
3. if $\Delta_i^+ = \emptyset$ return H_i otherwise continue;
4. increment i;
5. Let $a_i^+ \in \Delta_i^+$, generate $\bot_{(B,a_i^+)}$ (using *saturate*() see figure 2.3 on page 43),
6. Search the θ-subsumption lattice $\square \succeq D_j \succeq \bot_{(B,a_i)}$ (using *search*() see figure 4.4 on page 84), and return the "best" definite clause D_i in $\mathcal{L}$ such that
   $B \cup H_{i-1} \cup \{D_i\} \cup \{\Delta_i^+ \setminus a_i^+\} \models a_i^+$, and
   $B \cup H_{i-1} \cup \{D_i\} \not\models \square$, and
   $B \cup H_{i-1} \cup \{D_i\} \cup I_{\mathcal{L}} \not\models \square$;
7. $H_i = H_{i-1} \cup \{D_i\}$;
8. $\Delta_p = \{a_p | a_p \in \Delta_{i-1}^+ \wedge B \cup T_i \models \{a_p\}\}$;
9. $\Delta_i^+ = \Delta_{i-1}^+ \setminus \Delta_p$;
10. Go to Step 3.

Figure 4.3: An algorithm for abductively generalising *crosslists*.

$search(B, T, E^+, \mathcal{L}, \Delta_k, I_\mathcal{L}, S, n, a)$: Given background theory B, a set of clauses T, a finite set of observations E^+, a finite set of explanatory atoms Δ_k, hypothesis constraints $I_\mathcal{L}$, an "open list" S of potential clauses, an upper limit on the number of clauses searched n, and an explanatory atom a, returns a clause D in $\mathcal{L}$ such that $B \cup T \cup \{D\} \cup \{\Delta_k \setminus a\} \models a$, and D is consistent with $I_\mathcal{L}$.

1. $D = a$, $bestscore = score(a)$, $i = 0$;
2. if $S = \emptyset$ or $i > n$ return D otherwise continue;
3. Remove the first clause C from S;
4. increment i;
5. Refine clause C to a set of clauses $\mathcal{R}_C$ such that for every clause $R \in \mathcal{R}_C$ $B \cup T \cup \{R\} \models a$;
6. Score each clause in $\mathcal{R}_C$. Let h be the highest score and $BestSet$ be the clauses with score h in $\mathcal{R}_C$;
7. If $h > bestscore$ and $\exists Best \in BestSet$ such that $B \cup H\{Best\} \not\models \square$, and
   $B \cup H\{Best\} \cup I_\mathcal{L} \not\models \square$
   the $bestscore = h$ and $D = Best$;
8. Add each clause in $\mathcal{R}_C$ to S. Sort S according to descending *score* of clauses;
9. Go to Step 2.

Figure 4.4: A simple best-first search algorithm for abductively *searching* for the best clause. The score of each clause can be obtained by the *score* algorithm presented in figure 4.5.

score$(B, T, E^+, E^r, \Delta_k, D)$: Given background theory B, a set of clauses T, a finite set of observations E^+, a finite set of random observations E^r, a finite set of explanatory atoms Δ_k, a clause D, returns a score for the clause D.

1. $r = |E^r|$;
2. calculate the abductive generality $g_\Delta(D)$ of the partial theory with respect to the clause D (see definition 73);
3. calculate the abductive size of the partial theory $sz_\Delta(T)$ with respect to the clause D (see definition 74);
4. Return $r(1 - g_\Delta(T)) - sz_\Delta(T)$.

Figure 4.5: An algorithm for abductively *scoring* a clause with respect to the partially complete theory (from definition 75).

alecto$(B, E^+, I_\mathcal{L}, \mathcal{L}, E^r, \Delta, G, S_{max}, H_{max}, |\chi|_{max})$: Given background theory B, observations E^+, hypothesis constraints $I_\mathcal{L}$, a finite set of random observations E^r, abducible predicate definitions Δ, acceptability constraints on abductive explanations G, maximum numbers of skip lists, h-lists and crosslists (S_{max}, H_{max}, and $|\chi|_{max}$) returns a set of theories $\mathcal{T}$ from $\mathcal{L}$ such that each $T_i \in \mathcal{T}$ satisfies the following:
$B \cup T_i \models E^+$, $B \cup T_i \not\models \square$, and $B \cup T_i \cup I_\mathcal{L} \not\models \square$.

1. *generate_crosslists*$(B, E^+, \Delta, G, S_{max}, H_{max}, |\chi|_{max})$ which returns a set of crosslists $\mathcal{X}$;
2. $X_i = \mathcal{X}$, $\mathcal{T}_i = \emptyset$;
3. if $X_i = \emptyset$ return $\mathcal{T}_i$;
4. Increment i;
5. $\chi_i \in X_i$;
6. *generalise_crosslist*$(B, E^+, I_\mathcal{L}, \mathcal{L}, \chi_i, E^r)$ which returns T_i;
7. $\mathcal{T}_i = \mathcal{T}_{(i-1)} \cup T_i$;
8. $X_i = X_{i-1} \setminus \chi_i$;
9. Go to Step 3.

Figure 4.6: The ALECTO algorithm for Theory Completion.

4.5.1 Complexity issues

The ALECTO algorithm can be viewed as two independent components: 1) the extraction of abductive explanations (ground) χ from the background knowledge B and the observations E; and 2) the generalisation of the abductive explanations to produce hypotheses (clauses) H such that $B \cup H \models E$. The first component is described in *generate_crosslists*() (see figure 4.2 on page 77). The second component *generalise_crosslists*() is based on the standard Progol style ILP algorithm (see figure 4.3 on page 83). The complexity of ALECTO is analysed with respect to these two components.

The complexity of *generate_crosslists*()

For the analysis it is assumed that the computing of the abducibles from one observation e^+ using SOLD requires checking the existence of a (SOLD) proof for each example in E. The proof mechanism is essentially SLD resolution via PROLOG, and it can be represented by an AND-OR tree, where AND nodes are due to conjunctions of literals in a clause, OR nodes arise due to different clauses, and leaves are the unit clauses to be proved.

Let b be the maximum branching factor in the AND-OR proof for a goal, and d the maximum proof-depth for a goal, and $|E|$ be the number of examples, and h be the upper-bound on the number of h-lists produced from an example, and x the number of crosslists produced for the set of examples.

There are b^{2d-1} leaves in a tree of depth d. Assuming, as a first approximation, constant indexing and unification costs, checking the SOLD proof for one example to provide a skip-list can be done in b^{2d-1}. Acceptability constraints may invalidate any such proof, so that alternatives must be generated. Furthermore, the generation of ground h-lists also requires extra effort, which may be assumed constant. These combined extra efforts can be accounted by a constant factor c. The complexity of generating one h-list is $O(b^{2d-1}c)$. As there are $|E^+|$ examples to be explained, and an upper bound of h on the number of h-lists, the complexity of generating x crosslists is $O(b^{2d-1}c|E^+|hx)$.

The complexity of *generalise_crosslists*()

Srinivasan ([Srinivasan, 1999]) provides a complexity analysis of each of the main steps in the Progol style ILP algorithm. (See $ILP()$ figure 2.2 on page 42). These are

1. *Selecting* a clause in the refinement lattice;
2. *Refining* a clause in the refinement lattice;
3. *Scoring* the clauses in the refinement lattice.

Both from a theoretical and an empirical study ([Srinivasan, 1999]), the **scoring** step is the most computationally intensive and it is therefore the only step considered in the following analysis.

Similar to the preceding analysis, for the scoring step it is assumed that the computing of the score requires checking the existence of a proof for each abduced atom Δ_i in crosslist χ_j. Assume also that the total number of abduced atoms $|\Delta_\Sigma|$ is a constant factor k greater than the number of observations $|E^+|$ and the upper-bound on the number of h-lists h, and the upper-bound on the number of crosslists x (i.e. $|\Delta_\Sigma| = k|E^+|hx$). Again the proof mechanism is SLD resolution via PROLOG, and it can be represented

by an AND-OR tree, where AND nodes are due to conjunctions of literals in a clause, OR nodes arise due to different clauses, and leaves are the unit clauses to be proved.

There are b^{2d-1} leaves in a tree of depth d. Assuming, as a first approximation, constant indexing and unification costs, checking the proof for all the abduced atoms can be done in $b^{2d-1}|\Delta_\Sigma|$. Since there are at most N elements in the set of hypotheses $\mathcal{H}$ scored, the scoring can be done in $O(b^{2d-1}k|E^+|hxN)$.

Comments

The complexity of performing the extraction case abduction to translate observations into abducibles in the presence of background knowledge is comparable to the complexity of generalisation (or ILP) component. This agrees with experimental evidence presented in

The complexity of the overall `ALECTO` is $O(b^{2d-1}|E^+|hx)$.

The assumption of constant unification time is not met in domains where list processing is present (for example applications 1 and 2 presented in chapters 5 and 6). The unification time will be dependent on the length of the list and the operations required (e.g. full unification, or the unification of the head element only).

The size of the AND-OR tree becomes much greater than b^{2d-1} when the background knowledge contains Negation-as-failure ([Clark, 1978]). This is the case for the Event Calculus application domain which is studied in chapter 7.

4.6 Discussion

This section discusses the similarities and differences in approach of the `ALECTO` system to two related systems: 1) `CProgol5.0` [Muggleton and Bryant, 2000]; and 2) `ACL` and `MACL` [Kakas and Riguzzi, 2000]. All three systems have the ability to complete "missing" predicates and perform multiple-predicate learning.

The implementation of Theory Completion in the system `CProgol5.0` treats every predicate in the background theory (for which there is a definition) as a potential observation for use as training examples. Any atom can become a head of a bottom clause provided that: 1) the atom's predicate symbol appears in a head mode declaration; and 2) the atom's predicate symbol is in a lower level than that of the predicate selected as being "observational"; and finally 3) there is a negated literal derivable by Logical Back-propagation that matches the mode template. When searching for clauses, compression is the arbiter for learning. If the addition of the clause allows a more compressive theory, then it is accepted.

There are circumstances, however, when it is known that particular negated atoms produced by LBP *are not* desired by the Inductive Logic Programmer. Also there are circumstances when it is known in advance which predicates require completion (not simply *any* predicate in a lower level than the observational predicate). Early approaches taken in `CProgol5.0` did not provide sufficient control for the Inductive Logic Programmer. `ALECTO` goes some way to rectify these difficulties by allowing (even requiring) the Inductive Logic Programmer to specify: 1) which predicates define the observations; 2) the predicates to be completed; and 3) acceptability constraints on negated atoms used in the heads of bottom clauses.

When scoring potential clauses `CProgol5.0` relies on a notion of *deductive coverage* of observations. This restricts its ability to perform multiple-predicate learning where the predicates are either not mutually-recursive, or some definition for the predicates already exist in the background theory.

Kakas and Riguzzi's Abductive Concept Learning system `ACL` has much broader objectives than simple Theory Completion. Their system takes a background program that is an *Abductive Logic Program* (ALP)[9] and produces an hypothesis that is also an ALP. Like `ALECTO`, `ACL` takes as input atomic ground facts for examples. However, `ACL` can work with negative examples and it is capable of producing integrity constraints, both of which cannot be performed by `ALECTO`.

`ACL` uses a method for *abductive derivability* ([Kakas and Riguzzi, 2000] definition 2.9, page 251) which produces abductive explanations in a similar fashion to SOLD-resolution. Like `ALECTO`, in using `ACL` the Inductive Logic Programmer can adjust the language bias to ensure that all abductive explanations are ground.

In `ACL` an abductive notion of entailment is used as a basic coverage notion for learning. This is used, in part for scoring hypotheses. In [Kakas and Riguzzi, 2000] (page 261) a scoring function is used that takes into account whether observations are deductively and/or abductively covered. The `ALECTO` system also considers the abductive coverage of an hypothesis, in the presence of the abductive explanation.

`ACL` and `MACL` can perform multiple-predicate learning by using abduction to generate "extra training examples for the target predicates" (see [Kakas and Riguzzi, 2000] page 268). This is similar to the way in which `ALECTO` translates observations into abductive explanations, and then generalises them. In effect, `ALECTO` is similar to Kakas and Riguzzi's sub-system `ACL1` restricted to learning the rule part of an ALP. All three systems restrict themselves to generating "range restricted" definite clauses so as to ensure ground atoms are derived. Table 4.1 summarises the Theory completion features of the three systems.

[9] ALPs are defined on page 48.

Table 4.1: A comparison of some of the features of three implementations which can be used for Theory Completion: ALECTO, CProgol5.0, and ACL.

Feature	ALECTO	CProgol5.0	ACL
Provides control of the status of the observation	Yes	No	Yes
Provides control over which observations can be used for for producing negated literals	Yes	No	Yes
Positive only learning	Yes	Yes	No
Acceptability constraints	Yes	No	Yes
Accepts non-ground unit examples	No	Yes	No
Accepts negative examples	No	Yes	Yes
Deductive Cover testing	No	Yes	Yes
Abductive Cover testing	Yes	No	Yes

4.7 Summary

The Theory Completion techniques presented in chapters 2 and 3 have been implemented in the non-interactive system ALECTO. ALECTO makes use of the following features.

- Transforming observations into a set of abduced ground atoms (crosslists).
  - Each observation is transformed into a ground abductive explanation. Such transformations use SOLD-resolution, guided by settings provided by an inductive logic programmer.
  - One abductive explanation per observation is collected into a set of ground abduced atoms, called a crosslist.
- Mode directed Inverse Entailment is applied to the atoms in the crosslist.
- Searching for clauses utilises a positive-only scoring function, while provisionally accepting the truth of the other atoms in the crosslist.
  - The positive-only scoring function approximates a Bayes' posterior probability for the theory being learnt.
  - Multiple-predicate learning can be achieved by assuming that the atoms in the crosslist are true allowing clauses which contain literals yet to have a definition, to be scored.
- ALECTO is most similar in spirit to the TCIE system CProgol5.0, and also has features similar to those in the Abductive Concept Learning system ACL.

The next chapter details the application of the ALECTO system to three problems including one from the grammar domain, and two from the Event Calculus. The culmination of these applications is the generation of an Event Calculus program for controlling the navigation of a robot.

Part II

Applications of ALECTO

Chapter 5

Application 1: Learning components of a Natural Language Grammar

In this chapter the Theory Completion systems ALECTO and CProgol5.0 are applied to learning clauses from a very simple context free grammar. The initially complete grammar was used to generate *observations* of sentences, then clause definitions were removed from the complete grammar. The remaining grammar was then used as an existing background theory for relearning (or *completing*) the deleted definitions using the observed sentences.

Experiments were performed where first one, then two clauses were removed. In 12/26 cases ALECTO managed to correctly regenerate the deleted clauses by theory completion. CProgol5.0 only managed to correctly regenerate a single clause in one of the 26 cases.

5.1 Introduction

The first of the three application domains considered in this dissertation is that of Natural Language Processing. This can refer to anything from fairly simple string-manipulation tasks like stemming[1], or building concordances[2] of natural language texts, to higher-level tasks like processing user queries in natural language. The techniques outlined in the previous chapter are evaluated on this domain in which ALECTO is used to complete theories. In this chapter a set of experiments investigates the capabilities of ALECTO to generate missing clauses in the grammar domain – using the same inputs as the grammar examples in the preceding chapters.

[1] Algorithms for determining the morphological root of a given inflected word form.

[2] A *concordance* is an index to a text that includes the *contexts* in which the words or phrases occur.

5.2 General Experimental Methodology

The data for all of the experimental problems considered in this dissertation was *constructed artificially*. In each case a complete and correct logic program was available. These complete programs were used as a *source* for generating *observations*. The complete programs were then "tampered with", by removing some clauses, resulting in an incomplete program. The incomplete program, and the observations were then provided as input to `ALECTO` in an attempt to (re)produce clauses that repair or complete the program.

5.3 The toy example grammar domain

The test domain for this experiment is a simple context free grammar, originating from Van Le's book ([Van Le, 1993] pp 479). The grammar – as a PROLOG program – is shown in code snippet 5.1. The grammar is capable of parsing *legal* sentences, which are considered as *lists of words*. The predicates in the grammar define the relationships between two lists of words. For example the relationship *noun_phrase(A,B)* states that *the list of words A contains a prefix which is a noun phrase, and B is the suffix*. In effect such predicates are *consumers* of the list of words that are in their relationship – dumping the "left over" words into the second argument[3]. The second argument may then be passed to other predicates until terminal nodes in the grammar are reached, finally resulting in the empty list (provided that a legal sentence is used).

```
% Grammar from Van Le p479
sentence(S,R):-
    noun_phrase(S,T), verb_phrase(T,R).

noun_phrase(S,T):-
    determiner(S,N), noun(N,T).

verb_phrase(T,R):-
    verb(T,N), noun_phrase(N,R).

determiner([the|R],R).      determiner([a|R],R).

noun([president|R],R).      noun([cat|R],R).

verb([has|R],R).
```

Code snippet 5.1: The simple context free grammar.

The relationship between the predicates in the grammar are shown in

[3]Alternatively: *noun phrase strips off a legal noun phrase from the front of list A, and returns the remaining list into B.*

the calling diagram of figure 5.1. It is obvious that the grammar program is a directed-acyclic-graph, and is *stratified* [Lloyd, 1993] (and as such, it contains no recursion). In these experiments some "arrows" – or clauses – are deleted from the grammar, and then attempts are made to generate replacements by Theory Completion.

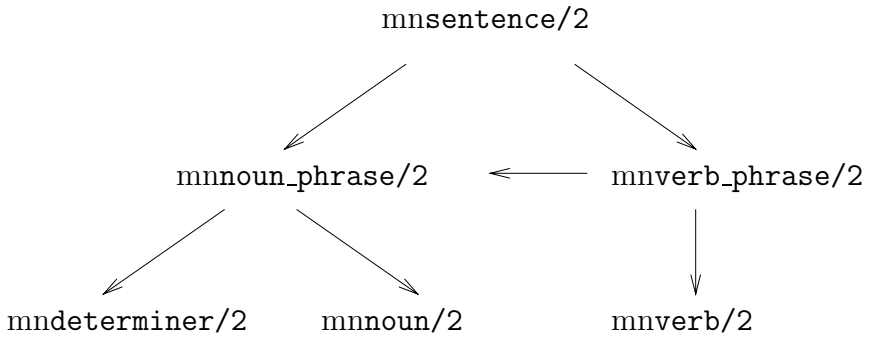


Figure 5.1: The calling diagram for the simple grammar.

What sentences can be generated by the grammar?

The set of sentences[4] that are derivable from the simple grammar are presented in code snippet 5.2. These sentences make up the observations required for completing the deleted clause(s).

```
sentence([the, president, has, the, president] ,[]).
sentence([the, president, has, the, cat] ,[]).
sentence([the, president, has, a, president] ,[]).
sentence([the, president, has, a, cat] ,[]).
sentence([the, cat, has, the, president] ,[]).
sentence([the, cat, has, the, cat] ,[]).
sentence([the, cat, has, a, president] ,[]).
sentence([the, cat, has, a, cat] ,[]).
sentence([a, president, has, the, president] ,[]).
sentence([a, president, has, the, cat] ,[]).
sentence([a, president, has, a, president] ,[]).
sentence([a, president, has, a, cat] ,[]).
sentence([a, cat, has, the, president] ,[]).
sentence([a, cat, has, the, cat] ,[]).
sentence([a, cat, has, a, president] ,[]).
sentence([a, cat, has, a, cat] ,[]).
```

Code snippet 5.2: The legal sentences generated by the simple context free grammar.

The experiments in completing the grammar are detailed in chapter 5 beginning on page 93.

[4]The grammar program generates 16 legal sentences.

The example context free grammar was taken from Van Le's book ([Van Le, 1993] page 479). The original program is shown in the previous chapter in code snippet 5.1 on page 94. At first sight the grammar may appear trivial, but for the purpose of investigating Theory Completion it contains sufficient complexity. The grammar parses simple sentences. Those sentences which are parsed can be considered as legal *observations* of sentences. There are also many illegal sentences in the grammar's Herbrand Base. The set of legal sentences that are derivable from the simple grammar are presented in code snippet B.1, page 174. These 16 sentences make up the observations available for completing deleted clause(s). One example sentence is:

$$sentence([the, president, has, a, cat], []).$$

The intuitive interpretation is that *the list of words* "the president has a cat"*, when parsed as a sentence, has no leftover words, and is therefore a legal sentence.*

Not only does the grammar *accept* legal sentences, but it also *rejects* illegal ones. The experiments in this chapter use positive-only learning (see for example [Muggleton, 1996], and section 2.4.3), and as such they require that generality estimates are calculated on the population of possible observations (legal or otherwise). For the purpose of generality estimation it is necessary to compare *legal* observations of sentences to those that exist in the Herbrand Base of the program.

For this grammar domain, investigations were limited to sentences of five or fewer words. This seems appropriate as none of the original sixteen example sentences contain more than five words[5]. The Herbrand Base of the grammar program contains 3905 sentences and non-sentences of five or fewer words[6]. A sample of these, which happens to contain one of the sixteen legal sentences, is shown in code snippet B.2 in appendix B. An example of a randomly chosen sentence from the Herbrand Base of the program is *sentence([the, has, has, the, president],[]).*

5.4 Objective

The objective of the experiments in this chapter was to *investigate which grammar clauses could be generated by Theory Completion techniques from examples of legal sentences.*

5.5 Method

The experimental procedure used was: 1) remove some clauses D from the complete grammar $T_{grammar}$ to form $T'_{grammar} = T_{grammar} \setminus D$; 2) use

[5]The legal sentences contain exactly five words each.

[6]$3905 = 5^1 + 5^2 \cdots + 5^5$

$T'_{grammar}$ and the observations of sentences as input to both ALECTO and CProgol5.0, and attempt to regenerate the clauses D.

The schedule of experiments

The experiments are categorised into three groups. First those that attempt to complete single clause deletions from the grammar. Second, experiments that complete two clauses deleted from the grammar. Finally, an experiment was performed in the "traditional" ILP learning mode, where the clause learned had the same predicate as the observations. The schedule of experiments, along with the clauses deleted, is shown in table 5.1.

5.6 Background theory specification

Most of the components of the background theory provided to ALECTO and CProgol5.0 across all runs are common. Slight differences in the remaining components is due to different syntactic encoding requirements in the two systems.

Abducibles The abducibles are the target predicates for completion. In CProgol5.0 these abducibles are communicated implicitly through *modeh* directives. For example, when attempting to complete the *verb_phrase/2* predicate the following modeh directive was provided to CProgol5.0:

```
:- modeh(*,verb_phrase(+listofwords,-listofwords))?
```

In the ALECTO system, abducibles are explicitly communicated. For the above example ALECTO would be provided the modeh directive as well as the following:

```
:- abducible(verb_phrase/2).
```

Acceptability constraints An acceptability constraint is required so that the abduced atoms obey the following schema: *" Both arguments in a grammar predicate are lists of words. The second argument in a grammar predicate is a tail or* suffix list *of the first argument"*. In CProgol5.0 this is encoded in the *star* version of predicates. For example to encode this information for the *verb_phrase/2* predicate the following is required:

```
'*verb_phrase'(A,B):-suffixlist(A,B).
```

For ALECTO this information is provided via a *ground_lit/2* predicate as follows:

```
ground_lit(verb_phrase(X,Y),verb_phrase(X,Y1)) :-
                                Y=Y1 , suffixlist(Y1,X).
```

The acceptability constraints for ALECTO for the grammar problem are presented in code snippet B.10 on page 178.

Random observations In positive-only mode, both CProgol5.0 and ALECTO require a random sample of the observation predicate from the Herbrand Base to estimate the generality of potential clauses. In CProgol5.0 these random observations are provided as ground atoms of *'*sentence'/2*, while ALECTO expects them to be stored in a separate *.r* file in the predicate *sentence*/2. Examples of these random observations are presented in code snippet B.2 on page 175.

Examples Both CProgol5.0 and ALECTO are issued with the 16 legal examples of sentence. For ALECTO examples are placed in a separate *.f* file, while for CProgol5.0 they are in the single input file but with the additional *:-observable(sentence/1)?* directive. The examples are presented in in code snippet B.1 on page 174.

Body mode declarations Both CProgol5.0 and ALECTO share common body mode declarations. There is one syntactic difference between the two systems. In CProgol5.0 directives are terminated with a *?* symbol, while in ALECTO they are terminated with a period. The body mode declarations are presented in in code snippet B.7 on page 177.

Other background knowledge The two final components of background knowledge that are provided to CProgol5.0 and ALECTO are: 1) typing; and 2) components of the grammar. The typing information for the grammar problem is specified in code snippet B.8 on page 177. The components of the grammar itself[7] is also provided as background knowledge. This is presented in code snippet B.11 on page 178.

5.7 Results

The theories produced by the two systems are presented in appendix B section B.3 starting on page 179.

The theories generated using CProgol5.0 and ALECTO for the Group I Grammar problems – single clause deletions – are shown in code snippets B.1 and B.4 respectively (on pages 179 and 181). The theories generated for the clause deletions problems (Group II) are shown in code snippets B.2 and B.5 using CProgol5.0 and ALECTO respectively (pages 179 and 182).

Finally, the case of learning the top level predicate in the usual ILP Observational Predicate Learning fashion produced the clauses for *sentence/2* shown in code snippets B.3 and B.7 using CProgol5.0 and ALECTO respectively on pages 181 and 183.

[7] With the target clauses removed appropriately.

The results are summarised in table 5.2. In this table a deleted clause is defined as having been recovered if it appears as a clause in the completed theory generated by the Theory Completion system.

5.8 Discussion

The success criterion for the summary table is quite strict. In some experiments clauses were produced with similar structure to the target clauses, but as they were not identical they were not considered. For example, the final experiment using `CProgol5.0` produced the following clause:

```
sentence([A|B],C) :-
    determiner([A|B],B), noun_phrase([A|B],D), verb_phrase(D,C).
```

This clause contains the logically redundant literal[8] `determiner([A|B],B)`, but is not identical to the target clause.

In the Group I – single clause deletion – experiments, only the *verb_phrase/2* case had singleton abductive explanations. For example one *noun_phrase/2* explanation was

$$\Delta = \left\{ \begin{array}{l} noun_phrase([the, president], []), \\ noun_phrase([the, president, has, the, president], [has, the, president]) \end{array} \right\}$$

The need for multiple atoms in the abductive explanations occurs whenever there is more than a single path from observation to abducible in the calling diagram (see the calling diagram for the simple grammar in figure 5.1 on page 95).

`ALECTO` had some success in generating two missing clauses at the same time. However, when the two deleted clauses were on the same path to the observation predicate in the calling diagram, then only the one closest to the observation was completed. For example no abductive explanation is obtainable in the predicate *noun/2* when *noun_phrase/2* is also removed (this is the same for *determiner/2* when *noun_phrase/2* is also removed, and for *verb/2* when *verb_phrase/2* is also removed). The notion of a path between the *abducible* and the observation is explicit in Logical Back-propagation, while it is implicit in SOLD-resolution.

`CProgol5.0` does quite poorly on this problem. It suffers when a definition is completely missing from the background knowledge. The software distribution of `CProgol5.0` contains a more complicated grammar example than the one studied in this chapter. For this more complicated grammar, partial definitions already exist for each of the predicates being completed. In this situation `CProgol5.0` performs better.

[8] A literal l is logically redundant in clause $C \vee l$ with respect to background theory B if $B \cup (C \vee l) \models C$.

5.9 Summary

In general ALECTO performed better than CProgol5.0 in the setting where no definitions existed for the predicates to be completed.

In the grammar experiments ALECTO performed as follows:

- 5/6 of single deletions were recovered;
- 7/20 of double deletions were recovered;
- 12/26 in total.

The features of ALECTO that were used were:

1. The transformation of observations into sets of abducibles.
2. A positive-only scoring function.
3. Alternative theories were generated using alternative *crosslists*.
4. The use of abductive evidence during scoring.

In the following chapter, an application that is an extension to this one is investigated. The domain is a context-dependent parser for detecting *intrusions* in a computer system.

Table 5.1: The schedule of grammar experiments, detailing the clauses to be removed from the original theory.

Group	Experiment number	*Clause(s) deleted from* $T_{grammar}$
I	1	`verb_phrase(A,B):-verb(A,C),noun_phrase(C,B).`
	2	`noun_phrase(A,B):-determiner(A,C),noun(C,B).`
	3	`determiner([a|B],B).  determiner([the|B],B).`
	4	`noun([cat|B],B).  noun([president|B],B).`
	5	`verb([has|B],B).`
II	6	`noun_phrase(A,B):-determiner(A,C),noun(C,B).` `verb_phrase(A,B):-verb(A,C),noun_phrase(C,B).`
	7	`noun_phrase(A,B):-determiner(A,C),noun(C,B).` `verb([has|B],B).`
	8	`noun_phrase(A,B):-determiner(A,C),noun(C,B).` `determiner([a|B],B).  determiner([the|B],B).`
	9	`noun_phrase(A,B):-determiner(A,C),noun(C,B).` `noun([cat|B],B).  noun([president|B],B).`
	10	`verb_phrase(A,B):-verb(A,C),noun_phrase(C,B).` `noun([cat|B],B).  noun([president|B],B).`
	11	`verb_phrase(A,B):-verb(A,C),noun_phrase(C,B).` `verb([has|B],B).`
	12	`verb_phrase(A,B):-verb(A,C),noun_phrase(C,B).` `determiner([a|B],B).  determiner([the|B],B).`
	13	`determiner([a|B],B).  determiner([the|B],B).` `noun([cat|B],B).  noun([president|B],B).`
	14	`determiner([a|B],B).  determiner([the|B],B).` `verb([has|B],B).`
	15	`noun([cat|B],B).  noun([president|B],B).` `verb([has|B],B).`
III	16	`sentence(A,B):-noun_phrase(A,C),verb_phrase(C,B).`

Experiment number	*Clauses to be completed*	Clause(s) recovered by ALECTO	Clause(s) recovered by CProgol5.0
1	verb_phrase/2	1/1	0/1
2	noun_phrase/2	1/1	0/1
3	determiner/2	0/1	0/1
4	noun/2	1/1	0/1
5	verb/2	1/1	0/1
6	noun_phrase/2 and verb_phrase/2	2/2	1/2
7	noun_phrase/2 and verb/2	2/2	0/2
8	noun_phrase/2 and determiner/2	0/2	0/2
9	noun_phrase/2 and noun/2	0/2	0/2
10	verb_phrase/2 and noun/2	0/2	0/2
11	verb_phrase/2 and verb/2	0/2	0/2
12	verb_phrase/2 and determiner/2	1/2	0/2
13	determiner/2 and noun/2	0/2	0/2
14	determiner/2 and verb/2	1/2	0/2
15	noun/2 and verb/2	1/2	0/2
16	sentence/2	1/1	0/1
	Total	12/26	1/26

Table 5.2: Summary of results of regenerating deleted clauses from the grammar using ALECTO and using CProgol5.0.

Chapter 6

Application 2: Learning components of an Intrusion Detection System

This chapter follows on from the previous by applying the Theory Completion system ALECTO to learning clauses that make up components of an Intrusion Detection System (IDS). Unlike the simple natural language grammar studied previously the IDS is a *context dependent* grammar. Such a detection system is capable of detecting *buffer-overflow attacks* by considering not simply the syntax of an attack stream, but by analysing the semantics of the attack sequence.

As in the previous chapter, the initially complete Intrusion Detection System was used to generate *observations* of buffer-over attacks, then clause definitions were removed from the complete Intrusion Detection System. The remaining Intrusion Detection System was then used as an existing background theory for relearning (or *completing*) the deleted components using the observed attacks.

Individual experiments were performed where single clauses were removed. A further experiment was performed where two predicate definitions were removed.

6.1 Introduction

Intrusion detection is the identification of potential breaches in computer security policy. The objective of an attacker is often to gain access to a system that they are not authorised to use. The attacker achieves this by exploiting a (known) software vulnerability by sending the system a particular input. Many intrusion detection systems examine input for syntactic signatures of known intrusions. In [Moyle and Heasman, 2002] it was demonstrated that logic programming is a suitable formalism for specifying the semantics of

attacks. Logic programs can then be used as a means of detecting attacks in previously unseen inputs. Furthermore, the work demonstrated that ILP can be used to induce detection clauses from examples of attacks. The description that follows is adapted from [Moyle and Heasman, 2002].

6.1.1 The buffer overflow attack

Buffer overflow attacks are a type of security vulnerability relating to incomplete input validation. The circumstances in which a buffer overflow can occur are well understood and wholly avoidable if the software is developed according to one of many secure programming checklists [Garfinkel and Spafford, 1996][1].

A Technical Description of a Buffer Overflow attack

The concepts behind exploiting a buffer overflow vulnerability are simple but the implementation details are subtle. They are discussed thoroughly from the perspective of the attacker in *"Smashing the Stack for Fun and Profit"* [Aleph-One, 1996]. Sufficient background and implementation details are reproduced here so that the reader may understand how such attacks can be detected.

Buffer overflow attacks exploit a lack of *bounds checking* on a particular *user-supplied field* within a computer software service that performs some transaction. The attacker supplies an *overly large input* transaction to the service. In the course of processing the transaction, the service copies the field into a buffer of a *fixed size*[2]. As a result, memory located immediately after the buffer is overwritten with the remainder of the attacker's input. The memory that is overwritten may have held other data structures, the contents of registers, or, assuming the system is executing a procedure, the *return address* from which program execution should continue once the procedure completes. It is the *return address* that is of interest to the attacker.

The function that copies data from one location to another without regard for the data size is referred to in the sequel as simply *the dangerous function*. Before the service executes the procedure containing the function call that leads to the overflow, the *return address* is pushed onto the stack so that program execution can continue from the correct place once the procedure is complete. Information about the micro-processor's *registers* is then pushed onto the stack so that the original state can be restored. Space for variables local to the procedure is then allocated.

[1] This does not demean the target of this work – although preventable – buffer overflow vulnerabilities are still prolific in computer software.

[2] The size of the buffer is smaller than that necessary to store the entire input of the attacker.

Arbitrary data	Sequence of of idle instructions	Exploit code	Multiple copies of return address

Figure 6.1: Components of a malicious transaction containing a buffer overflow attack.

The attacker aims to overwrite the return address thus controlling the flow of the program. A malicious transaction input of a buffer overflow attack is a sequence of bytes which can be considered as consisting of several macro-level components as shown in Figure 6.1.

The dangerous function call takes the attacker's input and overwrites the intended buffer in the following order: 1) the space allocated for the procedure's local variables and the register save area, and 2) the return address. The purpose of the sequence of *idle* instructions at the beginning of the malicious transaction sequence is to *idle* the processor[3]. Idle instructions are used to reduce the uncertainty of the attacker having to guess the correct return address which must be hard coded in the attack sequence. If there are no idle instructions the return address must be set to the beginning of the exploit code which requires the perfect knowledge of the state of the process' stack. By including a sequence of idle instructions at the start of the exploit code, the return address need only be set to an address within this sequence which greatly improves the chances of an exploit succeeding. This sequence of idle instructions is often referred to as a *NOP sledge*. *NOP* is the opcode that represents a *no operation* instruction within the CPU.

The attacker then aims to set the return address to a location within the idle sequence so that his exploit code is executed correctly. He supplies multiple copies of his desired return address to compensate for the uncertainty of where the actual return address is located. Once the dangerous function procedure has completed, values from the (overwritten) register save area are popped off the stack back into system registers and program execution continues from the new return address – now **executing the attacker's code.**

For the attacker to correctly build the malicious portion within the transaction, he must know some basic properties of the target system, including the following:

- The *processor architecture* of the target host as machine code instructions have different values in different processors.
- The *operating system* the target is running as each operating system has a different kernel and different methods of changing between user and kernel mode.

[3] *Idle* means "*do nothing*" and move on to the next instruction.

- The *stack start address* for each process which is fixed for each operating system/kernel.
- The approximate *location of the stack pointer*. This is a guess at how many bytes have been pushed onto the stack prior to the attack. This information together with the stack start address allow the attacker to correctly calculate a return address that points back into the idle sequence at the beginning of the exploit code.

The work described in the sequel is limited to the discussion of buffer overflow exploits for the Intel x86 family of processors running the Linux operating system. These are only pragmatic constraints. The work is easily extended to different processor architectures, operating systems, and other vulnerabilities.

6.1.2 An Intrusion Detection System for recognising buffer overflow attacks

An attacker must provide a stream of bytes, that, when executed after being stored in an overflowed buffer environment, will gain him control of the CPU, which will in turn, perform some task for him (say execute a shell on a Unix system). When crafting a string of bytes the attacker is faced with many alternatives, given the constraints that he must work within (e.g. the byte stream must be approximately the size of the vulnerable buffer). These alternatives are both at the *strategic* and *implementation* levels.

At the strategic level, the attacker must decide in what order the tasks will be performed by the attack code. For example, will the address of the command to the shell be calculated and stored before the string that represents the shell type to be called? Having decided on the components that make up the strategy, at the implementation level, the attacker has numerous alternative encodings to choose from. Which register, for example, should the null byte be stored in?

From the description above, it is clear that for any given attack strategy, there are vast numbers of alternative attack byte code streams that will, when executed, enact the strategy. This also means that simple syntax checking intrusion detection systems can easily be thwarted even by attackers using the same strategy that the syntactic IDS has been programmed to detect.

A *semantic* Intrusion Detection System was developed as a PROLOG program in [Moyle and Heasman, 2002], which was further used for the basis of the work in this chapter. The calling diagram of the system is shown in figure 6.2. The program includes a dis-assembler that encodes a subset of the semantics of Intel x86 class processor byte codes. The IDS includes a byte sequence generator that utilizes the same dis-assembler components. This byte sequence generator, when executed, produces a sequence of bytes by

generating a randomised implementation of the chosen intrusion strategy. To test the efficacy of the generated attach sequence of bytes it is then compiled into a dummy executable which simulates the process of the byte stream being called. This simply simulates the situation of the buffer having already been over flowed, and not the process of over flowing it per se. Each attack byte sequence is verified in its ability to open a shell (and hence its success for an attacker). These byte streams are considered as examples in the work that follows.

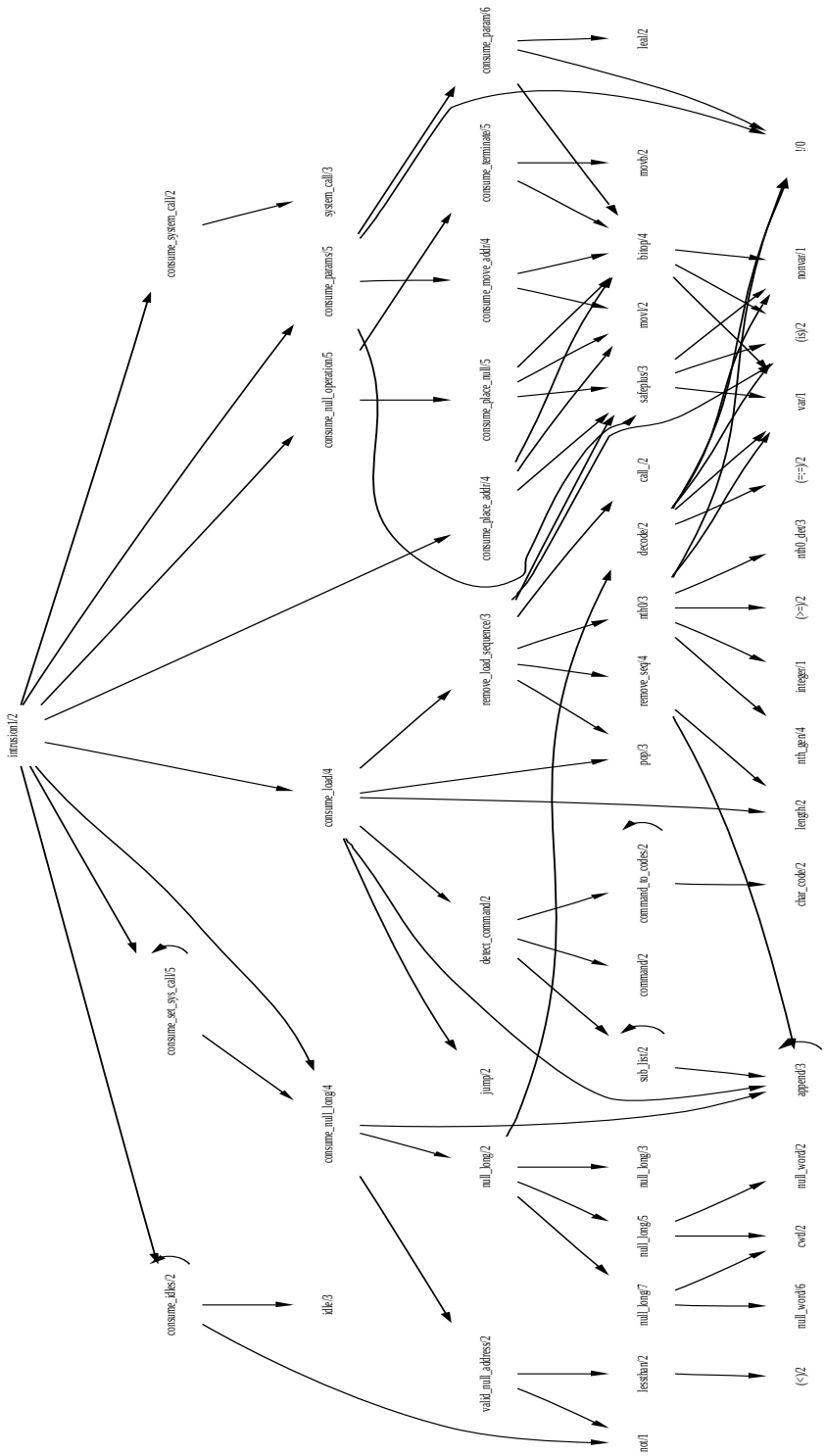


Figure 6.2: The calling diagram for the complete Intrusion Detection System [Moyle and Heasman, 2002].

An illustration of one high level clause for detecting a particular attack strategy is presented in code snippet 6.1. Example byte streams that implement the attack strategy (and are also detected by the high level clause) are shown in code snippet 6.2. Lower strata clauses that form part of the IDS are in code snippet 6.3. These behave in a similar fashion to the natural language grammar in the previous chapter, by *consuming* bytes from an input stream to form an output stream. The difference from the grammar in the IDS is that it is context dependent and other arguments are required.

```
intrusion1(A,I) :-
         consume_idles(A, B),
         consume_load(B, C, Addr, Len),
         consume_place_addr(C, D, Addr, Len),
         consume_null_long(D, E, Null, Addr),
         consume_null_operation(E, F, Len, Null, Addr),
         consume_set_sys_call(F, G, Null, 0x0b, Addr),
         consume_params(G, H, Addr, Null, Len),
         consume_system_call(H, I).
```

Code snippet 6.1: The *top level* intrusion detection clause for one particular buffer overflow attack strategy.

```
intrusion1([252,249,249,245,252,245,248,245,144,235,34,93,137,109,9,
              185,255,255,255,255,65,136,77,8,137,77,13,184,255,255,
              255,255,64,176,11,137,235,141,77,9,141,85,13,205,128,
              232,217,255,255,255,/,b,i,n,/,a,s,h],[]).
intrusion1([235,24,95,137,127,10,49,192,136,71,9,137,71,14,176,11,
              137,251,141,79,10,141,87,14,205,128,232,227,255,255,
              255,/,b,i,n,/,t,c,s,h],[]).
```

Code snippet 6.2: Two *examples* of malicious byte streams that conform to the intrusion attack strategy in code snippet 6.1. Note that the final few bytes of each example have been written as ASCII characters to convey the intuition of the attacker's intention.

6.2 Experiments for Theory Completion

The broad aim of this section is to explore the performance of `ALECTO` when applied to the recovery of clauses in the intrusion detection system. For this single clauses were selected for recovery from stratum 1 and stratum 2 (see figure 6.2), and a pair of clauses for joint recovery. These are the *target* clauses shown in code snippet 6.3.

6.2.1 Objective

The objective of these experiments was to test the hypothesis that: *Theory Completion techniques can be used to induce clauses for "consumer" clauses*

```
consume_load([Jump,Offset1,Pop|Rest],Rest3,Reg,Len) :-
    jump(Jump,_),
    pop(Pop,Reg,_),
    remove_load_sequence(Offset1,[Pop|Rest],Rest2),
    detect_command(Rest,Command),
    append(Rest3,Command,Rest2),
    length(Command,Len).

consume_place_addr([B1,B2,L1|Rest],Rest,RegSrc,L2) :-
    safeplus(L2,1,L1),
    movl(B1,_),
    bitop(B2,RegSrc,RegSrc,0x40).

consume_null_long([A,B|InList],OutList,Null,Addr):-
    null_long([A,B|NullLongList],Null),
    append([A,B|NullLongList],OutList,[A,B|InList]),
    valid_null_address(Null,Addr).

consume_null_operation(A,C,Len,Null,Addr) :-
    consume_terminate(A,B,Len,Null,Addr),
    consume_place_null(B,C,Len,Null,Addr).

consume_params(A,D,Addr,Null,Len) :-
    calculate_offsets(Len,Offset1,Offset2),
    consume_move_addr(A,B,Addr),
    consume_param(B,C,Offset1,Addr,Null,2'001),
    consume_param(C,D,Offset2,Addr,Null,2'010).

consume_place_null([B1,B2,L1|Rest],Rest,L2,RegSrc,RegDest) :-
    safeplus(L2,5,L1),
    movl(B1,_),
    bitop(B2,RegSrc,RegDest,0x40).
```

Code snippet 6.3: *Target* clauses from the IDS for Theory Completion.

for the Intrusion Detection System from observations consisting of examples of attack byte streams.

6.3 Method

The basic method was to use the correct and complete PROLOG program to randomly generate sequences of bytes that successfully exploit a buffer overflow. Such byte sequences are considered as *positive observations*, E^+, of attacks. One clause from a lower stratum was then removed from the IDS program leaving a common background program to be used as background knowledge, B_{IDS}, to `ALECTO`. This background program, including the definition of the top level predicate, was then used, along with positive examples of attack byte sequences, to complete definitions of predicates different to that of the observations.

Table 6.1: The IDS target clauses for each Theory Completion experiment.

Number	Predicate/Arity	number of literals	stratum
1	`consume_load/4`	7	1
2	`consume_place_addr/4`	4	1
3	`consume_null_long/4`	4	1 and 2
4	`consume_null_operation/5`	3	1
5	`consume_params/5`	5	1
6	`consume_place_null/5`	4	2
7	`consume_place_addr/4` and `consume_null_long/4`	4 4	1 1

The schedule of experiments

The schedule of the experiments and the clauses targeted for recovery are listed in table 6.1. Ten-fold cross validation was used to estimate the accuracy of the generated clauses for different sizes of example sets (see figure 6.3 for more details).

6.3.1 Input components for the IDS Theory Completion

The input components for the IDS Theory Completions were as follows:

Mode declarations A typical listing of mode declarations and settings for the recovery of a consumer clause (in particular the clause `consume_load/4`) are shown in code snippet C.5 on page 188.

Other background knowledge Two components make up the background knowledge: the grammar component, and the dis-assembler. The grammar component is primarily a set of *consumer* clauses. The target clauses in code snippet 6.3 are examples of these. The complete list of the consumer clauses is shown in code snippet C.3 in appendix C.

Sample background code for the dis-assembler is shown in code snippet C.4 in appendix C.

Examples Positive examples were produced by using the complete IDS and the chosen intrusion strategy to randomly generate valid attacks. Each of the attack strings were compiled into a test function to confirm that they caused a buffer overflow. 824 such confirmed attack strings were produced. Two examples are shown in code snippet 6.2, and a further 10 can be found in code snippet C.1 in appendix C.

Estimating the accuracy of `ALECTO` on recovering IDS target clauses

Let m be the number of randomly selected examples from a pool of M positive examples E^+ and n be the number of folds:

for $m \in \{2, 5, 10, 20, 50\}$

1. for $i = 1$ to n do
   (a) randomly select examples: $E^+_{(m,i)} \subset E^+$;
   (b) construct IDS theory H with `ALECTO` using B_{IDS} and $E^+_{(m,i)}$;
   (c) test H on remaining positive examples $E^+ \setminus E^+_{(m,i)}$ to obtain an estimate of accuracy $a_{(m,i)}$;
2. Return an estimate of accuracy $\hat{a}_i = \frac{\sum a_{(m,i)}}{n}$.

Figure 6.3: The experimental procedure for the recovery of IDS target clauses by Theory Completion. Note that for the experiments $n = 10$, $M = 824$.

A source of random examples was produced in the following manner. The 824 positive examples were used to produce 10,000 random examples by randomly mutating the byte streams at an average mutation rate of 1 bit in 1000 bits. These examples are sufficiently similar to "pass" some way into the IDS grammar, but not completely. This pool of random examples was sampled for use as input to `ALECTO` for the experiments. Two random examples are shown in code snippet C.2 in appendix C.

6.4 Results

The results of the seven theory completion experiments for the intrusion detection system are mixed. The learning curves and run times for each of the experiments are tabulated in appendix C (see tables C.1 to C.7 on pages 189 to 191). The learning curves for the recovery of `consume_params/5` and `consume_null_operation/5` (separately) are plotted in figure 6.4. The run times for the recovery of `consume_params/5` and `consume_null_operation/5` (separately) are plotted in figure 6.5.

Sample completed theories for the seven experiments are recorded in appendix C (see code snippets C.6 to C.12 on pages 192 to 193).

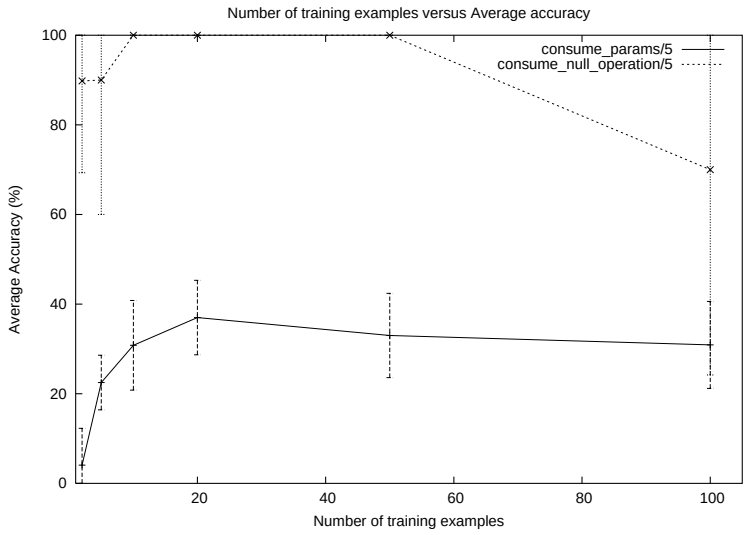


Figure 6.4: Learning curves for regenerating `consume_params/5` and `consume_null_operation/5` (separately). The estimates of accuracy are averaged from 10 fold cross-validation, while error bars are 1 population standard deviation.

For the five single clause completion experiments from stratum 1, one (`consume_load/4`) did not produce any theory; one produced theories with zero accuracy (`consume_place_addr/4`); and three produced theories with some accuracy (`consume_null_long/4`, `consume_null_operation/5`, and `consume_params/5`). The correct original clause for `consume_null_operation/5` was recovered. For the one single clause completion experiments from stratum 2, (`consume_place_null/5`) no accurate clauses were generated. For the one dual clause completion experiments (`consume_place_addr/4` and `consume_null_long/4`) only one clause was produced and on a theory level they were not accurate (caused by the lack of definitions for one of the clauses).

6.5 Discussion

It is clear from the experimental results that `ALECTO` does not perform as well on the IDS domain as it did on the toy NLP domain in the previous chapter. This can be explained, in part, by the context dependent nature of the IDS grammar. In such a situation, it becomes more difficult to encode the functional dependencies of the grammar in a manner that enables extraction-case abduction to produce atoms that are true in the intended in-

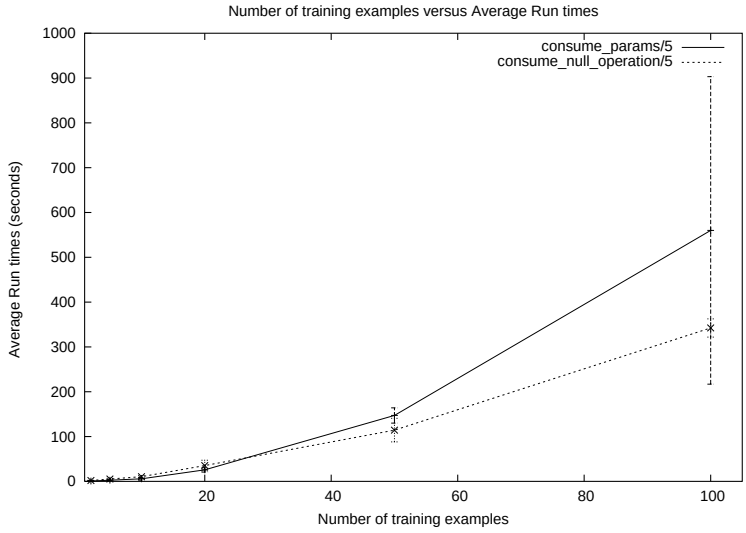


Figure 6.5: Run times for regenerating `consume_params/5` and `consume_null_operation/5` (separately). The run times are averaged from 10 fold cross-validation, while error bars are 1 population standard deviation.

terpretation. Consider, for instance, the original clause for `consume_load/4` in code snippet 6.3 (for which `ALECTO` was unable to recover any clauses) which has an intricate argument input-output map as shown in figure 6.6. The final `Len` argument, when the complete clause is executed deductively, provides the length of the attacker's command that gets executed by the operating system. This is calculated via three literal calls. This is an example where the simple scheme implemented in `ALECTO` for producing ground literals from SOLD produced atoms is not able to generate ground abduced atoms with bindings for length that are correct, and thus suitable for saturation.

The extraction simple strategy for generating abducibles does not make use of any *type theory*, nor any functional knowledge that might be available in the domain. One approach to improve the situation might be to incorporate elements of Constraint Logic Programming (CLP) [Jaffar and Lassez, 1987]. This would be similar to approaches in both Abductive Logic Programming ([Kakas and Michael, 1995]) and Inductive Logic Programming [Puget, 1997]. Such an implementation would be expected to produce abducibles that are more likely to be true in the intended interpretation, and may reduce the space of abductions which would also improve the computational efficiency.

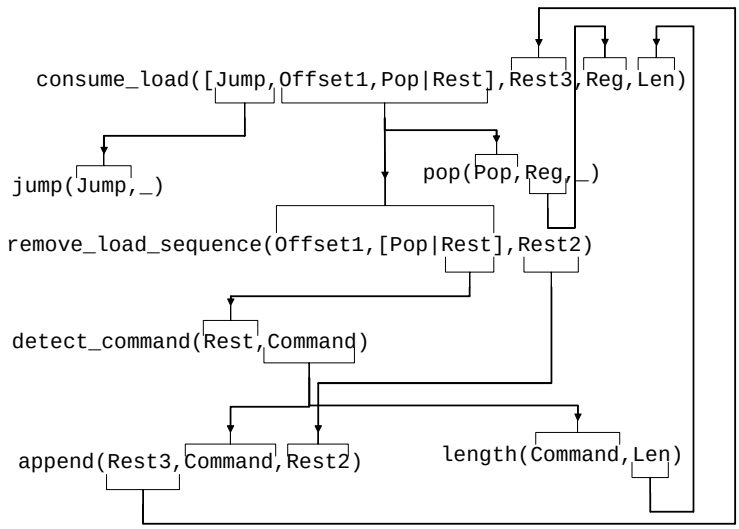


Figure 6.6: The argument input–output map for the clause `consume_load/4`.

6.6 Summary

Experiments were preformed to explore how ALECTO performs in the Theory Completion of a context dependent grammar used for detecting computer intrusions causing buffer overflows.

In the experiments ALECTO performed as follows:

- clauses were produced for 3/5 of single deletions from stratum 1;
- clauses were produced for a single deletion from stratum 2;
- a partial recovery of two clauses from stratum 1.

The following chapter investigates an application that is particularly well suited to Theory Completion: learning Event Calculus programs.

Chapter 7

Application 3: Learning Event Calculus programs by Theory Completion

This chapter introduces the Event Calculus and details experiments in learning Event Calculus programs. The objective was to investigate whether Theory Completion techniques could be used to induce Event Calculus domain specific axioms for a system from observations of its dynamic behaviour.

The approach was to take a working Event Calculus program and *run it forwards* to generate time trace data. Having collected the data, the domain specific axioms of the program were then removed, and the remaining *skeleton* program was added to the time trace data to form the input data for induction. The input data was then used by the Theory Completion system `ALECTO` to *regenerate* a set of domain specific axioms.

`ALECTO` was able to complete Event Calculus programs. In particular, the sets of domain specific axioms induced for controlling the navigation of a robot contained (on average) 14 clauses and had an average accuracy of 70.2%.

7.1 Introduction

The Event Calculus [Kowalski and Sergot, 1986] was developed as a general approach for representing and reasoning about events and their effects in a logic programming framework, and as an attempt to solve the frame problem [McCarthy, 1959]. An Event Calculus program allows a dynamic system to be modelled so that future states of the system can be determined, thus enabling the derivation of system values that *hold at* future times based upon events that have *happen*ed at previous times.

The Event Calculus utilises the *common sense law of inertia* [Shanahan, 1997] also known as *default persistence*. This law formalises the notion that properties remain unchanged throughout time unless effected by the action of an event. Change is mediated solely through the action of events, where events are related to system properties if they initiate (or terminate) periods in which the properties hold (or do not hold). Default persistence allows reasoning about properties even when information about the world is not precisely known.

Event calculus programs are typically normal logic programs consisting primarily of two sets of clauses. All such programs contain a common set of *core axioms* which encode the default temporal reasoning for expressing the common sense law of inertia [Shanahan, 1997]. The second component is the set of *domain specific axioms* (DSAs) which encode the dynamics of the particular domain being represented. Together, the core axioms and the DSAs allow future states of the program to be derived from some known state of the domain, and a sequence of events.

An Event Calculus program can be viewed as a meta-level interpreter which is capable of deriving the state of the world at points in time This meta-level interpreter – based on *core axioms*, requires as input: 1) initial conditions of the world, and 2) a temporally ordered list of actions as input. The final component required by an Event Calculus program the set of effect axioms – or domain specific axioms. It is, in effect, the domain specific axioms which provide any particular Event Calculus program with its dynamic behaviour. The domain specific axioms are a form of state-transition rules that describe the preconditions that must be satisfied before an action can have its desired effect on any properties. Typically the domain specific axioms for an Event Calculus program are provided by a human programmer. Complete event calculus programs have worthy properties [Denecker *et al.*, 1996], including being able to prove that the program will behave as desired. Creating correct and complete sets of DSAs - in common with any programming task - can be difficult [Denecker, 1995; Shanahan, 1996].

A burden of all event calculi is that they require humans to provide domain specific axioms (DSAs) for definitions of predicates that encode the dynamics of the domain - a process that can be difficult. This difficulty is exacerbated by the non-monotonicity of the formalism which is caused,

in part, by the use of Negation-as-failure. One way to reduce this burden would be to induce predicate definitions for the domain specific axioms from observed time traces of property values from an existing dynamic system. This is the key motivation for the work in this chapter.

The Event Calculus has been used to describe several domains including an Air Traffic Control system [Sripada, 1995], and a decision support system for vascular surgery [Abeysinghe, 1993]. Researchers in Leuven have successfully implemented a sliding window protocol with go-back-n in the Event Calculus [Denecker *et al.*, 1996], from which they were able to prove that the communications system as modelled was deadlock free. The Event Calculus has also been used as the basis for solving planning problems (e.g. [Eshghi, 1988], [Kowalski, 1979], [Kakas and Michael, 1995], [Van Belleghem *et al.*, 1994]). More recently Shanahan [Shanahan, 2000; Shanahan and Witkowski, 2000] has been working with what is termed *Cognitive Robots* which are physical autonomous robots whose behaviour is programmed using the Event Calculus. In fact, one application in this chapter is to generate one of the Cognitive Robot's programs automatically.

This chapter is structured as follows. A detailed description of the Event Calculus follows immediately. Section 7.3 outlines the motivation, experimental methodology, and the background for the applications of learning Event Calculus programs. The descriptions of the actual experiments and their results follow in Sections 7.4 (the Blocks World) and 7.5 (the Robot Navigation Control Program). The chapter concludes with a discussion and summary in sections 7.6 and 7.7.

7.2 Introduction to the Event Calculus

As an introduction to the Event Calculus consider the blocks world shown in figure 7.1. In this section an Event Calculus program T_{Blocks} is developed that permits reasoning about what *fluents*[1] hold at different times in the blocks world. Reasoning about events in the Event Calculus is based on a *narrative* of *events*. For example, the process of moving the two blocks shown in figure 7.1 individually requires that block A is first moved (first event), and then block B is moved (second event). Notice that the definitions for the terms in italics appear shortly in section 7.2.1 on page 121.

The Event Calculus program T_{Blocks} contains clauses that describe the initial state of the blocks world T_0, which can be represented by the following Horn clauses.

$$
\begin{array}{ll}
 & initially(on(c,table)) \quad initially(on(b,c)) \quad initially(on(a,b)) \\
\text{(BI)} & initially(on(d,table)) \quad initially(clear(a)) \quad initially(clear(d)) \\
 & initially(clear(table))
\end{array}
$$

[1] McCarthy [McCarthy, 1959] states that the term *fluent* was used by Newton to describe a physical quantity that depends on time.

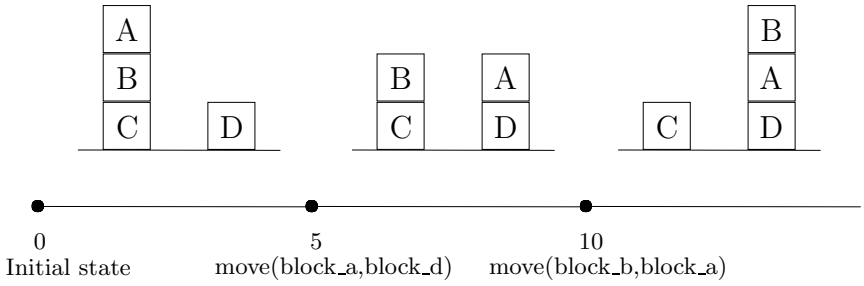


Figure 7.1: The Blocks World Narrative, illustrating the sequence of events and the subsequent evolution of the state of the world.

The narrative T_N, detailing the occurrence of events for the rearrangement of the blocks world, is given by the clauses

(BH1) $\quad happens(move(a, d), 5)$

(BH2) $\quad happens(move(b, a), 10)$

The intended meaning of clause BH1 is that the event moving block A onto block D ($move(a, d)$) occurs at time instant 5.

At this point the list of clauses presented is not sufficient to reason about the temporal nature of the state of the blocks. That is to say that the conjunction of T_0 and T_N do not logically entail statements about the position of the blocks at later times - time 6 for instance. For example

$$T_0 \wedge T_N \nvDash holds_at(on(a, d), 6)$$

To derive information about the state of the world at later times two more components of the Event Calculus must be included; the *core axioms* and the *domain specific axioms*. The *core axioms* of the Event Calculus, T_{Core},

consist of the following normal program clauses[2].

$$\begin{array}{ll} & holds_at(Fluent, Time_2) \leftarrow \\ \text{(EC1)} & \quad happens(Event, Time_1) \wedge\ initiates(Event, Fluent, Time_1) \\ & \quad \wedge\ Time_1 < Time_2 \wedge\ not\ clipped(Time_1, Fluent, Time_2). \\ & clipped(Time_1, Fluent, Time_2) \leftarrow \\ \text{(EC2)} & \quad happens(Event, Time) \wedge\ terminates(Event, Fluent, Time_1) \\ & \quad \wedge\ Time_1 < Time \wedge\ Time < Time_2. \\ & holds_at(Fluent, Time) \leftarrow \\ \text{(EC3)} & \quad initally(Fluent) \wedge\ not\ clipped(0, Fluent, Time). \end{array}$$

The intended meaning of clause EC1 is that a fluent value holds at a later time, $Time_2$, if an event happens at an earlier time, $Time_1$, and that event initiates the fluent value, provided that the fluent value is not clipped between those two times. Clause EC2 defines what it is for a fluent value to be *clipped*. A fluent value is clipped between times $Time_1$ and $Time_2$ if there is an event that occurs at some time in between the two that terminates the fluent value. Finally, the third core axiom (EC3) caters for the special case of the initial state from which a fluent continues unchanged provided that it has not been clipped. The inclusion of the core axioms with those of T_0 and T_N still does not permit the correct derivation of properties other than those from the initial conditions.

A clausal description of which events trigger the state of particular fluents is required. These clauses are the domain specific axioms (DSAs). The clauses that describe the relationship between events in the blocks world and the state of the fluents in the blocks world, T_{DSA}, are

$$\begin{array}{ll} & initiates(move(X, Y), on(X, Y), Time) \leftarrow \\ \text{(BDSA1)} & \quad holds_at(clear(X), Time) \wedge\ holds_at(clear(Y), Time) \\ & \quad \wedge\ diff(X, Y) \wedge diff(X, table). \\ & terminates(move(X, Y), on(X, Z), Time) \leftarrow \\ \text{(BDSA2)} & \quad holds_at(clear(X), Time) \wedge\ holds_at(clear(Y), Time) \\ & \quad \wedge\ holds_at(on(X, Z), Time) \wedge\ diff(X, Y) \\ & \quad \wedge\ diff(X, table). \\ & initiates(move(X, Y), clear(Z), Time) \leftarrow \\ \text{(BDSA3)} & \quad holds_at(clear(X), Time) \wedge\ holds_at(clear(Y), Time) \\ & \quad \wedge\ holds_at(on(X, Z), Time) \wedge\ diff(X, Y) \\ & \quad \wedge diff(Y, Z). \\ & terminates(move(X, Y), clear(Y), Time) \leftarrow \\ \text{(BDSA4)} & \quad holds_at(clear(X), Time) \wedge\ holds_at(clear(Y), Time) \\ & \quad \wedge\ diff(X, Y) \wedge diff(X, table). \end{array}$$

[2]A *normal program clause* is a clause that may contain one or more negated literals in its body. e.g. $p \leftarrow not\ q.$, where *not* represents negation-as-failure.

The intended interpretation of the predicate $diff(X, Y)$ is that objects X and Y are unique.

Let T_{Blocks} be the conjunction of T_0, T_N and T_{DSA}. The addition of the core axioms completes the Event Calculus program. It is now possible to derive statements about the state of the blocks world after the initial configuration. For example

$$T_{Blocks} \wedge T_{Core} \models holds_at(on(a, d), 6)$$

The Event Calculus presented above is known as the Simplified Event Calculus (SEC) [Sadri and Kowalski, 1995], which is often used instead of the original Event Calculus.

7.2.1 Components of Event Calculus programs

Language elements

The ontological primitives used by the Event Calculus are *events* and *fluents*. Fluents are time-varying properties of the world, which may have different values at different times. Events are considered as instantaneous occurrences which may cause change in the values of fluents. A set of events is called a *narrative*.

An Event Calculus program is built from a domain description consisting of time points, fluents, actions and an ordering over time. The elements of the language are a non empty set of time points Π; an order over the time points $\preceq$; a non empty set of events Δ; and a non empty set of Fluents Φ [Kakas and Michael, 1995].

Definition 76 (Domain elements) *The domain of discourse for an Event Calculus program T_{EC} is a finite set of objects D.*

The set of domain elements in the blocks world example is

$$D_{Blocks} = \{block_a,\ block_b,\ block_c,\ block_d,\ table\}\,.$$

Definition 77 (Fluent) *A* fluent F^n *is an n-ary predicate*[3] *symbol that takes domain $n \geq 0$ elements as arguments.*

The set of fluents in the blocks world example is:

$$\Phi_{Blocks} = \{on/2, clear/1\}$$

Fluents can be considered as values that may be measured or observed in the domain.

$$F_{Blocks} = \left\{ \begin{array}{l} on(block_a, block_a),\ on(block_a, block_b),\ on(block_a, block_c), \\ \ldots, \\ clear(block_a),\ clear(block_b) \end{array} \right\}$$

[3] Strictly it is a function symbol.

Definition 78 (Time points) *A time point t is a unique point in time. Usually $t \in \mathcal{N}$*

The set of time points in the blocks world example is $\Phi_{Blocks} = \{0, 1, 2, \ldots\}$. Only time points that are totally ordered will be considered here.

Definition 79 (Events) *An* event A^n *is an n-ary predicate*[4] *symbol that takes $n \geq 0$ elements as arguments.*

The set of events in the blocks world example is

$$\Delta_{Blocks} = \{move(block_a, block_d),\ on(block_b, block_a), \ldots\}\,.$$

Events are instantaneous actions that may be observed occurring in the domain.

Definition 80 (Fluent occurrence) *A fluent occurrence $\beta : \Phi \times T \rightarrow \{\blacksquare, \square\}$. Fluent occurrences are written in terms of the meta-predicate -* holds_at*.*

An example of a fluent occurrence in the blocks world is

$$holds_at(on(block_a, block_d), 6).$$

Definition 81 (Event occurrence) *An event occurrence $\alpha : A \times T \rightarrow \{\blacksquare, \square\}$*

An example of an event occurrence in the blocks world is

$$happens(move(a, d), 5).$$

Definition 82 (Event Calculus program) *An Event Calculus program T_{EC}, is a Normal Logic Program containing the conjunction of core axioms, static axioms, and domain specific axioms.*

Definition 83 (Static axioms) *T_{Static} consists of*

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

$$initially(\beta)$$

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

$$happens(\alpha, t)$$

[4]This is also strictly it is a function symbol.

- *A finite set of general clauses not containing the predicates,* initially*,* happens*,* initiates *or* terminates*,* holds_at*, or* $<$*.*

Definition 84 (Domain specific axioms) *T_{DSA} consist of*

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

$$initiates(\alpha, \beta, T) \leftarrow \Gamma$$

and a finite set of terminates *clauses of the form*

$$terminates(\alpha, \beta, T) \leftarrow \Gamma$$

where Γ *does not include the predicates,* initially*,* happens*,* initiates *or* terminates*, and every occurrence of the* holds_at *predicates is of the form*

$$holds_at(\beta, T)$$

Definition 85 (Event Calculus core axioms) *The three Event Calculus core axioms T_{Core} are EC1, EC2, and EC3 (see page 120).*

T_{Core} consists of a set of default persistence clauses, and a set of default termination clauses.

Axiom structure

The axiom structure of an Event Calculus program T_{EC}, can be considered as consisting of three components. The first component is the set of *core axioms* that are immutable and provide an encoding of default persistence. The second component is the set of *domain specific axioms* which encode the causal relations between events and the properties of the system being modelled. The third component provides an encoding of an initial state of the system, the narrative of events, and any auxiliary predicates.

$$T_{EC} = T_{Core} \cup T_{DSA} \cup T_{Static}$$

7.2.2 The Event Calculus as a Logic Program

Difficulties may be encountered when executing an Event Calculus program as a logic program, for example in Prolog [Clocksin and Mellish, 1987]. Some of these difficulties arise due to negation-as-failure as the default rule of non-inference. Negation as failure effects both the semantics of logic programs, and their computational efficiencies.

Semantic Issues

The meaning of a definite logic program, P, is usually taken to be the minimum Herbrand model, $\mathcal{M}(P)$ [Lloyd, 1993]. $\mathcal{M}(P)$ is not suitable for normal logic programs (those containing a literal prefixed by *not*). One technique for providing a semantics to a normal logic program is the program's Clark completion [Clark, 1978] where *not* is interpreted by negation as failure (to prove) .

The completion of a logic program specifies negative information that the programmer implicitly defines when writing the program. Unfortunately, only a limited class of logic programs have been shown to have consistent completions (i.e. those that do not contain both $\neg a$ and a for any atom a). For example hierarchical logic programs have been shown to have consistent completions [Lloyd, 1993], but they are of limited use given that they preclude the use of recursion. Similar results have been shown for *stratified logic programs* [Apt *et al.*, 1987], and *call consistent logic programs* [Kunen, 1989]. Within an Event Calculus program the predicate *holds_at* is defined indirectly in terms of *initiatates* and *not terminates*, which may in turn be defined in terms of *holds_at* [Shanahan, 1997]. Such a predicate definition renders Event Calculus programs as not being call consistent logic programs, and thus none of the standard theorems can be applied to guarantee a consistent completion. However, if a restriction is placed on the Event Calculus static clauses, then classes of Event Calculus programs exist which have consistent completions.

Theorem 86 (Consistent completion of Event Calculus programs) *(From [Shanahan, 1997] pages 247-249.) Let T_{EC} be an Event Calculus program including a set of* static *clauses S. If S has a consistent completion then so does T_{EC}.*

The above theorem provides a guideline for writing Event Calculus programs with clear semantics.

Computational issues

The exhaustive search of negation-as-failure is exacerbated for Event Calculus programs because in the process of deriving what fluents hold at any time it may be necessary to recompute answers for other *holds_at* facts that may have also been derived by previous queries. It is possible to alleviate this problem by use of a lemma storage mechanism [Dryllerakis, 1992; Shanahan, 1991; Shanahan, 1997]. Efficient implementations of the Event Calculus are also discussed in [Sripada, 1995].

One further problem that may be encountered when executing an Event Calculus program is that the ordering of the time of the events may not be bound at the time of computation - in particular variables of the $<$ relation.

This problem may be overcome by applying a suitable delaying mechanism so that the proof may continue without complete bindings. This type of approach is taken in Constraint Logic Programming systems [Jaffar and Lassez, 1987]. Dryllerakis [Dryllerakis, 1992] has successfully implemented Event Calculus programs in the constraint logic programming system CLP(R).

7.2.3 Knowledge representation in the Event Calculus

The domain specific axioms of an Event Calculus program provide humanly comprehensible rules that relate the relationships between objects in the domain. These include both non-temporal (static) relationships and dynamic relationships. For example BDSA3 relates the following notions.

- The event of moving one object from another object - $move(X, Y)$
- The state of an object being clear in the future (a dynamic property) - $clear(Z)$
- The state of objects at the time of the event (a dynamic property) - $holds_at(clear(X), Time)$,$holds_at(clear(Y), Time)$, $holds_at(on(X, Z), Time)$.
- Static relationships exist between objects - $diff(X, Y)$, $diff(Y, Z)$.

It can be argued that from a knowledge representation perspective, the Event Calculus domain specific axioms, by separating dynamic and relationships, are an improvement over representations that have a time argument associated with every predicate [Kuipers, 1986].

7.2.4 Programming in the Event Calculus

In common with regular programming languages, constructing correct and complete Event Calculus programs that model the dynamics of the domain of discourse is not easy. Such a task involves both human creativity and expertise.

Domain specific axioms in Event Calculus programs that refer to specific fluents often contain clauses for the initiation and termination of fluent properties for correct dynamic modelling. The removal of one set of these clauses will result in the derivation of incorrect fluent values, caused by the non-monotonicity of the formalism. This can make the writing of Event Calculus programs difficult. It is this difficulty that motivates attempts to learn Event Calculus programs.

7.3 Learning Event Calculus programs

The learning of Event Calculus domain specific axioms (DSAs) is particularly well suited to Theory Completion. This learning process can be stated as:

- *Given*:
  - a set of integrity constraints I
  - a partial theory $T_{EC} = T_{Core} \cup T_{Narrative} \cup T_{Background}$
  - evidence E (observations of dynamic behaviour, in the predicate $holds_at/2$)
- *Find*: a theory $T'_{EC} = T_{Core} \cup T_{Narrative} \cup T_{Background} \cup T_{DSA}$ obtained by completing T_{DSA} (that is by adding definitions for the predicates $initiates/3$ and $terminates/3$) such that T'_{EC} explains E and is consistent with I.

This is very similar in style to the definition of Theory Completion provided at the beginning of chapter 3 (see page 47).

Specifying the settings for completing Event Calculus programs is a straightforward task for the inductive logic programmer. In fact, most of the effort required in the writing of an Event Calculus program (either manually or with the aid of a system) is expended on the completion of two predicate definitions: `initiates/3` and `terminates/3`. Furthermore, the consequences of these predicates are visible in the `holds_at/2` predicate – which is the observational evidence used for completing the Event Calculus program.

In the sequel the Theory Completion system `ALECTO` is applied to the problem of generating the domain specific axioms for two Event Calculus programs from observations of desired dynamic behaviour. First, the simple blocks world Event Calculus program is used. The motivation for using this problem is to compare the two Theory Completion systems `CProgol5.0` and `ALECTO`. The data-set used in [Moyle, 1999] is also used here. Second, the real-world Event Calculus program to control the navigation of a robot, is completed.

7.3.1 Background and Motivation

The experiments in learning Event Calculus programs follow on from the work presented in [Moyle and Muggleton, 1997; Moyle, 1999] in which `CProgol5.0` was used as the system for Theory Completion. Some of the difficulties encountered in this earlier work included: 1) having to significantly reformulate the Event Calculus so that domain specific axioms were

encoded in a single predicate; 2) the need to learn DSAs for each fluent separately; 3) lack of control over abduced atoms; and 4) the use of an inappropriate cost function for scoring clauses. Other criticisms of the work include the experimental domain chosen (the Blocks World) which was too restricted; and that too much preprocessing of the data was required (see [Wellner, 1999] page 101).

The presentation of the experimental work relating to the simple Blocks World is included here for comparative purposes only. It is not intended to completely replicate the work done in [Moyle and Muggleton, 1997; Moyle, 1999]. It is, however, included to illustrate that in principle, `ALECTO` can be used to solve the same problem as that solved by `CProgol5.0`.

7.3.2 Objective

The objective of the experiments was to test the hypothesis that: *Theory Completion techniques can be used to induce clauses for domain specific axioms of Event Calculus programs from observations of a dynamic system.*

7.3.3 Method

The general procedure that was used can be broadly categorised into two tasks: 1) data generation; and 2) induction. The data generation was performed by using an existing, correct event calculus program and executing it to generate observations of the program's dynamic behaviour. This execution is a deductive process which produces time trace data. The time trace data is then processed so that it is suitable for input to the Theory Completion system `ALECTO`. The input data is used with the existing background theory to induce domain specific axioms. The process is depicted in figure 7.2.

Definitions of data items

This section defines terminology used to describe the data items for the experiments. Some definitions follow from those in section 7.2.

Definition 87 (Fluent trace) *A* **fluent trace**, **observational fluent trace**, *or simply* **trace** *for a fluent is a set of ground fluent facts recorded at different times. Typically a fluent trace represents all known values*[5] *of the fluent throughout the time period of observation. For example, the fluent trace for* `on(block_a,block_b)` *in the blocks world (see figure 7.1 on page 119) is* {`holds_at(on(block_a,block_b),0),holds_at(on(block_a,block_b),1), ..., not_holds_at(on(block_a,block_b),6),not_holds_at(on(block_a,block_b),7), ..., not_holds_at(on(block_a,block_b),11),not_holds_at(on(block_a,block_b),12)`}.

[5]The fact that a fluent does not hold is recorded (for convenience) in atoms of *not_holds_at/2*.

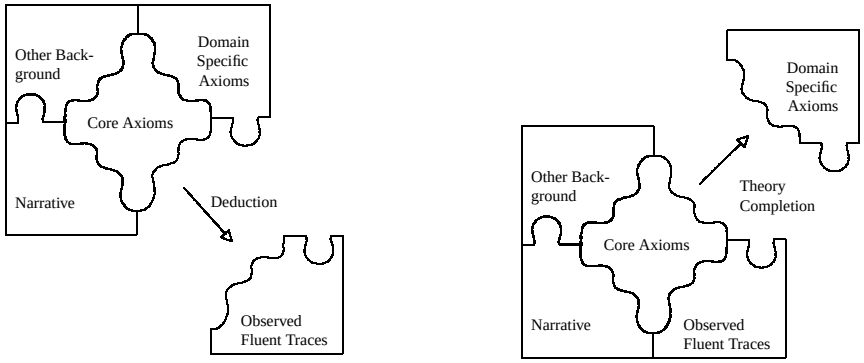


Figure 7.2: Depiction of experimental method. A: Generate time trace data; B: Induce Domain Specific Axioms using Theory Completion.

Definition 88 (Dependent and independent fluents) *A* **dependent fluent** *is a fluent for which domain specific axioms are required to be induced.* **Independent fluents** *are fluents whose traces are known, but for which DSAs are not required. However, the value of the independent fluents may be used when forming domain specific axioms to describe the behaviour of dependent fluents. For example, from the Robot Navigation Control program, the objective is not to learn DSAs for predicting whether doors will be open or closed at a particular time. However, the state of the doors is required to be known for learning the target DSAs. In this domain* `loc/2` *is an example of a dependent fluent, while* `door_open/1` *is an independent fluent (see code snippet 7.9 on page 140).*

Definition 89 (Initiating and terminating transitions) *A fluent trace contains an* **initiating transition** *when a fluent changes from* not holding *to* holding *in the next recorded time instance. Similarly a fluent trace contains a* **terminating transition** *when a fluent changes from* holding *to* not holding *in the next recorded time instance. For example a terminating transition from figure 7.1 (on page 119) is*
`holds_at(on(block_a,block_b),5),not_holds_at(on(block_a,block_b),6).`

The impetus to learn DSAs for a dependent fluent is provided whenever a fluent trace contains an initiating or terminating transition that is not predicted by the existing Event Calculus program.

7.3.4 Common background knowledge specification

Some of the components of the input data to `ALECTO` for learning domain specific axioms are common to both the Blocks World and Robot Navigation Control problems. This section details some of these common components.

The specification of most of these components follow from the definition of Event Calculus programs, and are likely to be general enough to use on other Event Calculus programs.

Event Calculus Core Axioms The core axioms for the Event Calculus are common to all Event Calculus programs. The core axioms provide the semantic and computational mechanisms for reasoning about actions and change. They are shown in code snippet D.1 on page 194.

Abducibles The abducibles communicate to `ALECTO` which predicates may be used to form the heads of clauses when completing a theory. The predicates `initiates/3` and `terminates/3` are used to define domain specific axioms in the Event Calculus, and they are therefore defined as abducibles (see code snippet D.2 in Appendix D on page 194).

Acceptability constraints The acceptability constraints for the Event Calculus encode the following domain knowledge:

> Abduced *initiates/3* atoms must refer to a fluent that has been observed having an *initiating transition* at that time point. Similarly, abduced *terminates/3* atoms must refer to a fluent that has been observed having a *terminating transition* at that time point.

This information is encoded in the program in code snippet D.3 in Appendix D on page 195.

Hypothesis structure constraints The hypothesis structure constraints for the Event Calculus restrict the clauses that are considered when completing the program. These constraints are shown in Appendix D in code snippet D.4 on page 196.

- The first constraint prevents DSA clauses having fewer than two literals.
- The second constraint ensures that all DSA clauses are *strongly generative*. Strongly generative clauses are those that do not have any singleton variables. This restriction is stronger than that of *syntactically generative clauses* defined in [Muggleton and Feng, 1992] in which every variable in the head also appear in the body. In the constraint here, the restriction is implemented by ensuring that any variable in a DSA clause appears at least twice.
- The third constraint expresses the notion that "a terminates clause should mention in the body the state of the fluent that is referred to in the head" For example when terminating the fluent *on/2*, the DSA clause should contain the literal *holds_at(on(X,Y),T)*,

ensuring that the fluent is indeed in a state that can be terminated.

- The fourth constraint prevents the fluent referred to in the head holding in the body in an *initiates/3* clause. This is almost the reverse of the previous constraint. This ensures that a fluent may not be initiated if it already holds.
- The fifth constraint ensures that the fluent referred to in the head of an initiates clause is mentioned in the body of the clause.

```
%%% Event Calculus Core Axioms

holds_at(F,T2):-
    happens(A,T1), lt(T1,T2), initiates(A,F,T1), not clipped(T1,F,T2).

holds_at(F,T):-
    initially(F), not clipped(0,F,T).

clipped(T1,F,T2):-
    happens(A,T), le(T1,T), lt(T,T2), terminates(A,F,T).

lt(T1,T2):-
    time_point(T1), time_point(T2), T1 < T2.
le(T1,T2):-
    time_point(T1), time_point(T2), T1 =< T2.
```

Code snippet 7.1: The Event Calculus core axioms (as a PROLOG program).

7.3.5 An Event Calculus program to describe the blocks world

The blocks world setting is described earlier in this chapter in section 7.2 (page 118). The domain specific axioms describe how *move* events may trigger blocks to become *on* top of other blocks, or to become *clear*. The DSAs as a PROLOG code is shown in code snippet 7.2. The following describes the components of the Blocks World Event Calculus program.

Core Axioms The core axioms for the Blocks Worlds are presented in code snippet 7.1.

Domain Specific Axioms The DSAs for the Blocks World are shown in code snippet 7.2.

Initial Situation The initial situation for the Blocks World has blocks C and D on the table, block A on block B, block B on block C, and blocks A and D are clear. This is shown in code snippet 7.3, and is illustrated in figure 7.1.

Narrative of events Two move events occur in the Blocks World. These are also shown in code snippet 7.3.

Other background knowldedge A predicate describing that the objects in the Blocks World domain are *diff*erent is also provided. This clause is `diff(X,Y):- X \==Y`.

```
initiates(move(X,Y),on(X,Y),T):-
     holds_at(clear(X),T), holds_at(clear(Y),T),
     diff(X,Y), diff(X,table).

terminates(move(X,Y),on(X,Z),T):-
     holds_at(clear(X),T), holds_at(clear(Y),T),
     diff(X,Y), holds_at(on(X,Z),T),
     diff(Y,Z).

initiates(move(X,Y),clear(Z),T):-
     holds_at(clear(X),T), holds_at(clear(Y),T),
     diff(X,Y), holds_at(on(X,Z),T),
     diff(Y,Z).

terminates(move(X,Y),clear(Y),T):-
     holds_at(clear(X),T), holds_at(clear(Y),T),
     diff(X,Y), diff(X,table).
```

Code snippet 7.2: The domain specific axioms to describe the Blocks World in the Event Calculus.

The consequences of the Blocks World Event Calculus program is the fluent trace. For example the following fluent values *hold at* time point 8:

```
% Initial situation

initially(on(block_c,table)).
initially(on(block_b,block_c)).
initially(on(block_a,block_b)).
initially(on(block_d,table)).
initially(clear(block_a)).
initially(clear(block_d)).
initially(clear(table)).
```

```
% Narrative of events

happens(move(block_a,block_d),5).
happens(move(block_b,block_a),10).
```

Code snippet 7.3: The initial situation and the narrative of events to describe the Blocks World in the Event Calculus.

on(block_c,table)
on(block_d,table)
on(block_b,block_c)
on(block_a,block_d)
clear(block_b)
clear(block_a)
clear(table)

The experiments and their results for completing the Blocks World DSAs using time traces similar to these are described in section 7.4.

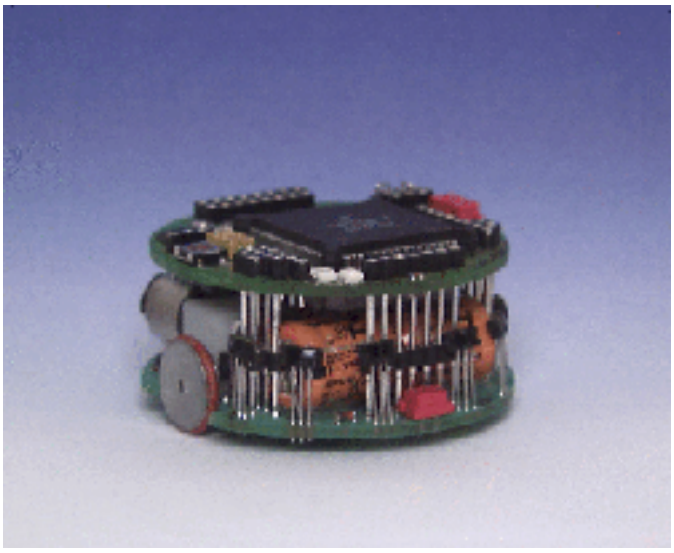


Figure 7.3: A photograph of the Khepera® Miniature Mobile Robot (reproduced with permission of K-Team SA, Switzerland).

7.3.6 An Event Calculus program to control a robot's navigation

Murray Shanahan and colleagues ([Shanahan, 2000], [Shanahan and Witkowski, 1999], [Shanahan, 2000], and [Shanahan and Witkowski, 2000]) have developed software for controlling real robots using the Event Calculus as the programming language. One of Shanahan's motivations for choosing the Event Calculus was that it is

> ... a well understood and mathematically rigorous formalism for reasoning about action, change, and space ...

The robots studied by Shanahan [Shanahan, 2000] are small (55mm diameter, and 30mm high), and are relatively low cost. Their motion is provided by three wheels, which are driven by two DC motors. They have sensory input from eight infra-red proximity and light sensors. A photograph of one such robot is pictured in figure 7.3.

The robot, suitably programmed, is capable of moving within the confines of a "model office". The layout of the office is shown in figure 7.4. The actual model office is constructed from wood with an overall floor plan of 3 m by 6 m (the walls are 10 mm high).

There are many aspects of robot control that the researchers have studied: these include *planning* and *navigating*. Planning typically produces an ordered set of *actions* to move the robot from its initial position to some desirable final position. Navigating can be viewed as the execution of the

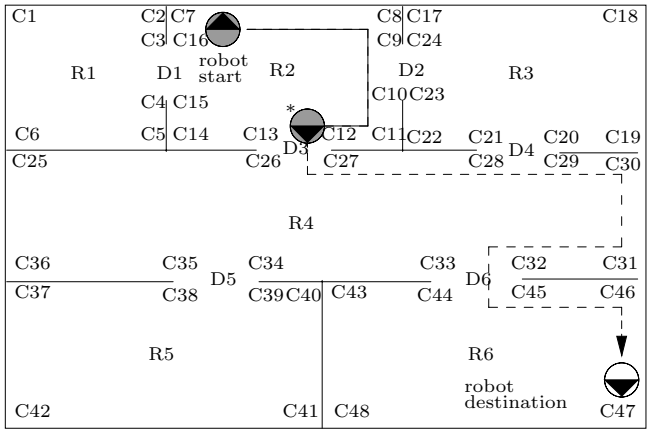


Figure 7.4: A schematic of the robot and its "model office" environment. The robot is represented by a circle with a triangle inside. The right angle of the triangle represents the direction the robot is facing. Shaded robots indicate that the robot is in transit. The unshaded robot is stationary. The robot's initial position and destination are shown. The robot at $*$ is enroute to the final destination. The office consists of *rooms* - *R*, *doors* - *D*, and *corners* - *C*.

planned set of *actions*. The Event Calculus program that takes the actions and the start position, and then controls the robot to its destination is known here as the **Robot Navigation Control program**.

The Robot Navigation Control program

The Robot Navigation Control Program is made up of the "usual" elements needed for an Event Calculus program: core axioms, domain specific axioms, an initial state, a narrative of events, and other background information. The following outlines fragments of PROLOG programs to encode these elements.

Core Axioms The core axioms for the Robot Navigation Control program are the same as those presented in code snippet 7.1.

Domain Specific Axioms The domain specific axioms that control the robot's navigation are listed in code snippet 7.4. These axioms[6] or "rules" are situation-action rules that specify what actions – or events – cause what changes in state, provided that the preconditions are met when the event occurs. There are two fundamental changes in state that can occur: initiation and termination of fluent states. Typically

[6]These axioms were kindly provided by Murray Shanahan.

the domain specific axioms referring to a particular event trigger have rules for both initiation and termination.

The robot's navigation strategy can be informally described as "cling-to-the-wall" and as such the robot is restricted to performing a set of primitive actions. These actions (or events) are described in table 7.1. Notice that these rules, in theory, can be applied to different situations – for instance alternative "room" layouts, different event narratives, and different initial situations.

The first domain specific axiom shown in code snippet 7.4 defines the *initiating* relationship between the *follow_wall* action and the robot's *loc*ation fluents.

$$\begin{aligned} initiates(follow_wall, loc(corner(C2), ahead), T) \leftarrow \\ holds_at(loc(corner(C1), behind), T) \wedge \\ next_visible_corner(C1, C2, left, T). \end{aligned}$$

This axiom states that "*the occurrence of a follow_wall event causes the robot to move to a location where the next visible corner on the left is immediately ahead – provided that the robot is located at a corner immediately behind – at the time the event occurs*".

The second domain specific axiom shown in code snippet 7.4 defines the *terminating* relationship between the *follow_wall* action and the *loc*ation fluent. This axiom simply states that "*the occurrence of a follow_wall event* **always** *causes the robot to cease being located at a corner immediately behind – from the time the event occurs*".

Initial Situation An example initial state is presented in code snippet 7.5 (refer also to figure 7.4). It is a set of special fluent values, stating the location of the robot, and open-closed state of the doors in the office at time point 0. The example initial state defines that all six doors are open, and that the robot is located in room *R2*, at corner *C7*, with the corner *ahead*[7].

Narrative of Events The narrative of events is the temporally ordered set of actions that, when correctly "applied", will enable the robot to navigate from its initial state to its destination. An example narrative, generated by the Abductive Event Calculus Planner as described in [Shanahan, 2000], is shown in code snippet 7.6. In this example, the fact $happens(turn(right), 9)$ states that at time point 9 the action $turn(right)$ occurs.

[7] A corner is *ahead* of the robot when the robot is facing *into* the corner, while a corner is *behind* when the robot is facing *away* from the corner.

Other background information The floor plan of the "building" that the robot navigates within is presented in figure 7.4. In this floor plan the following elements are labelled: $C1$ to $C48$ represent unique corners; $D1$ to $D6$ for doors; and $R1$ to $R6$ for rooms. The description of such a plan is provided to the Event Calculus in the form of a PROLOG program shown in code snippet 7.7.

The complete robot control program (and its physical embodiment) accounts for the fact that the doors in the environment can open or close at anytime. The robot, however, can only deduce that a door is closed by realising that by following the wall it gets to the next corner, and *not* a doorway. This deduction is provided via the clauses in code snippet 7.8.

```
initiates(follow_wall,loc(corner(C2),ahead),T):-
    holds_at(loc(corner(C1),behind),T),
    next_visible_corner(C1,C2,left,T).
terminates(follow_wall,loc(corner(C),behind),T).

initiates(go_straight,loc(corner(C2),behind),T):-
    holds_at(loc(corner(C1),ahead),T), door(D,C1,C2).
terminates(go_straight,loc(corner(C1),ahead),T):-
    holds_at(loc(corner(C1),ahead),T),door(D,C1,C2).

initiates(turn(right),loc(corner(C),behind),T):-
    holds_at(loc(corner(C),ahead),T), inner(C).
terminates(turn(right),loc(corner(C),ahead),T):-
    holds_at(loc(corner(C),ahead),T), inner(C).

initiates(turn(left),loc(corner(C2),behind),T):-
    holds_at(loc(door(D),in),T), holds_at(in(R1),T),
    connects(D,R1,R2), door(D,C1,C2), next_corner(R2,C1,C2).
terminates(turn(left),loc(corner(C1),ahead),T):-
    holds_at(loc(corner(C1),ahead),T), door(D,C1,C2),
    holds_at(door_open(D),T).

initiates(turn(left),loc(door(D),in),T):-
    holds_at(loc(corner(C1),ahead),T), door(D,C1,C2),
    holds_at(door_open(D),T).
terminates(turn(left),loc(door(D),in),T):-
    holds_at(loc(door(D),in),T).

initiates(turn(left),in(R2),T):-
    holds_at(loc(door(D),in),T), holds_at(in(R1),T), connects(D,R1,R2).
terminates(turn(left),in(R1),T):-
    holds_at(loc(door(D),in),T), holds_at(in(R1),T).
```

Code snippet 7.4: The Event Calculus domain specific axioms to describe the Robot Navigation Control program.

Table 7.1: The primitive actions of the robot. These actions depend on the situation the robot is in.

Action	Description	Pre-condition	Post-condition
turn(right)	Change the robot's direction $+\pi/2$	corner is ahead	corner is behind
turn(left)	Change the robot's direction $-\pi/2$	corner is ahead	corner is behind
follow_wall	Move the robot along the wall to the next corner	corner is behind	corner is ahead
go_straight	Move across an (open) doorway	doorway ahead	doorway behind
turn(left)	Move into a doorway	standing at a corner to an open door	standing in the door
turn(left)	Move into the adjacent room	standing in a connecting doorway	standing in a connecting doorway

```
/* Initial Situation */

initially(door_open(d1)).
initially(door_open(d2)).
initially(door_open(d3)).
initially(door_open(d4)).
initially(door_open(d5)).
initially(door_open(d6)).
initially(loc(corner(c7),ahead)).
initially(in(r2)).
```

Code snippet 7.5: The robot's initial situation (Taken from Shanahan [Shanahan, 2000]).

```
happens(turn(right), 4).      happens(go_straight,31).
happens(follow_wall, 5).      happens(follow_wall,33).
happens(turn(right), 9).      happens(turn(right),39).
happens(follow_wall, 11).     happens(follow_wall,43).
happens(go_straight, 13).     happens(turn(right),49).
happens(follow_wall, 17).     happens(follow_wall,53).
happens(turn(right), 19).     happens(turn(left), 55).
happens(follow_wall, 23).     happens(turn(left), 57).
happens(turn(left),  25).     happens(follow_wall,59).
happens(turn(left), 27).      happens(turn(right),62).
happens(follow_wall,29).      happens(follow_wall,65).
```

Code snippet 7.6: The robot's narrative of events – a plan of execution.

```
/* Room layout */

next_corner(r1,c1,c2).
next_corner(r1,c2,c3).
next_corner(r1,c3,c4).
next_corner(r1,c4,c5).
next_corner(r1,c5,c6).
next_corner(r1,c6,c1).

door(d1,c3,c4).

inner(c1).
inner(c2).
inner(c5).
inner(c6).

next_corner(r2,c7,c8).
next_corner(r2,c8,c9).
next_corner(r2,c9,c10).
next_corner(r2,c10,c11).
next_corner(r2,c11,c12).
next_corner(r2,c12,c13).
next_corner(r2,c13,c14).
next_corner(r2,c14,c15).
next_corner(r2,c15,c16).
next_corner(r2,c16,c7).

door(d1,c15,c16).
door(d2,c9,c10).
door(d3,c12,c13).

inner(c7).
inner(c8).
inner(c11).
inner(c14).

next_corner(r3,c17,c18).
next_corner(r3,c18,c19).
next_corner(r3,c19,c20).
next_corner(r3,c20,c21).
next_corner(r3,c21,c22).
next_corner(r3,c22,c23).
next_corner(r3,c23,c24).
next_corner(r3,c24,c17).

door(d2,c23,c24).
door(d4,c20,c21).

inner(c17).
inner(c18).
inner(c19).
inner(c22).

next_corner(r4,c25,c26).
next_corner(r4,c26,c27).
next_corner(r4,c27,c28).
next_corner(r4,c28,c29).
next_corner(r4,c29,c30).
next_corner(r4,c30,c31).

next_corner(r4,c31,c32).
next_corner(r4,c32,c33).
next_corner(r4,c33,c34).
next_corner(r4,c34,c35).
next_corner(r4,c35,c36).
next_corner(r4,c36,c25).

door(d3,c26,c27).
door(d4,c28,c29).
door(d5,c34,c35).
door(d6,c32,c33).

inner(c25).
inner(c30).
inner(c31).
inner(c36).

next_corner(r5,c37,c38).
next_corner(r5,c38,c39).
next_corner(r5,c39,c40).
next_corner(r5,c40,c41).
next_corner(r5,c41,c42).
next_corner(r5,c42,c37).

door(d5,c38,c39).

inner(c37).
inner(c40).
inner(c41).
inner(c42).

next_corner(r6,c43,c44).
next_corner(r6,c44,c45).
next_corner(r6,c45,c46).
next_corner(r6,c46,c47).
next_corner(r6,c47,c48).
next_corner(r6,c48,c43).

door(d6,c44,c45).

inner(c43).
inner(c46).
inner(c47).
inner(c48).

connects(d1,r1,r2).
connects(d2,r3,r2).
connects(d3,r2,r4).
connects(d4,r3,r4).
connects(d5,r4,r5).
connects(d6,r4,r6).

connects(d1,r2,r1).
connects(d2,r2,r3).
connects(d3,r4,r2).
connects(d4,r4,r3).
connects(d5,r5,r4).
connects(d6,r6,r4).
```

Code snippet 7.7: The PROLOG encoding of the layout of the "model office" (Taken from [Shanahan, 2000]).

```
next_visible_corner(C1,C2,left,T):-
    holds_at(in(R),T),
    next_corner(R,C1,C2), not(invisible_corner(C2,T)).
next_visible_corner(C1,C3,left,T):-
    holds_at(in(R),T),
    next_corner(R,C1,C2), invisible_corner(C2,T),
    next_visible_corner(C2,C3,left,T).

invisible_corner(C1,T):-
    door(D,C1,C2), not holds_at(door_open(D),T).
invisible_corner(C1,T):-
    door(D,C2,C1), not holds_at(door_open(D),T).
```

Code snippet 7.8: Miscellaneous predicates for the robot (Taken from [Shanahan, 2000]).

Executing the Robot Navigation Control program

Having assembled all the components of the Robot Navigation Control (RNC) program, what can it be used for? Being a PROLOG program, the RNC may be interpreted by a PROLOG engine. With the RNC loaded it is possible to query the system as to the value of any fluents at any particular time.

For example, to find out what the state of the world is at time '*' in figure 7.4, the PROLOG engine can be queried for `holds_at(Fluent,26)`, which returns the following fluent values.

door_open(d1)
door_open(d2)
door_open(d3)
door_open(d4)
door_open(d5)
door_open(d6)
in(r2)
loc(door(d3),in)

The meaning of the above bindings for the fluent is that at time point 26 all the doors are open, and the robot is in door *D3*, having approached the door from room *R2*.

Examples of the fluent values that can be derived by assembling all the components of the Robot Navigation Control program are shown in code snippet 7.9. The experiments relating to the generation of DSAs for the RNC are presented in section 7.5, while the next section describes the experiments relating to the generation of DSAs for the Blocks World domain.

```
holds_at(door_open(d1),0).
holds_at(door_open(d2),0).
holds_at(door_open(d3),0).
holds_at(door_open(d4),0).
holds_at(door_open(d5),0).
holds_at(door_open(d6),0).
holds_at(in(r2),0).
holds_at(loc(corner(c7),ahead),0).
holds_at(door_open(d1),1).
holds_at(door_open(d2),1).
holds_at(door_open(d3),1).
holds_at(door_open(d4),1).
holds_at(door_open(d5),1).
holds_at(door_open(d6),1).
holds_at(in(r2),1).
holds_at(loc(corner(c7),ahead),1).
holds_at(door_open(d1),2).
holds_at(door_open(d2),2).
holds_at(door_open(d3),2).
holds_at(door_open(d4),2).
holds_at(door_open(d5),2).
holds_at(door_open(d6),2).
holds_at(in(r2),2).
holds_at(loc(corner(c7),ahead),2).
     :
     :
```

```
holds_at(door_open(d1),25).
holds_at(door_open(d2),25).
holds_at(door_open(d3),25).
holds_at(door_open(d4),25).
holds_at(door_open(d5),25).
holds_at(door_open(d6),25).
holds_at(in(r2),25).
holds_at(loc(corner(c12),ahead),25).
holds_at(door_open(d1),26).
holds_at(door_open(d2),26).
holds_at(door_open(d3),26).
holds_at(door_open(d4),26).
holds_at(door_open(d5),26).
holds_at(door_open(d6),26).
holds_at(in(r2),26).
holds_at(loc(door(d3),in),26).
holds_at(door_open(d1),27).
holds_at(door_open(d2),27).
holds_at(door_open(d3),27).
holds_at(door_open(d4),27).
holds_at(door_open(d5),27).
holds_at(door_open(d6),27).
holds_at(in(r2),27).
holds_at(loc(door(d3),in),27).
     :
     :
```

Code snippet 7.9: Fluent values at different time points as derived from the Robot Navigation Control program.

7.4 Learning Domain Specific Axioms for the Blocks World

This section outlines the generation of DSAs to describe the Blocks World using `ALECTO`. This experiment was designed for simple comparative purposes with earlier work ([Moyle and Muggleton, 1997; Moyle, 1999]). The specific input components, additional to those outlined in section 7.3.4, are described, and the results presented.

7.4.1 Input components specific to the Blocks World

The input components specific to the blocks world were as follows:

Mode declarations the target predicates for learning follow from the objective of learning domain specific axioms. The two *head* mode predicates are `initiates/3` and `terminates/3`. The *body* mode predicates contain both *dynamic* literals[8] and the single static domain literal `diff/2`. The complete mode declarations for the Blocks World are presented in code snippet D.5 on page 197.

Inititial Situation and Narrative of events The initial situation defines the values of fluents at the first time point. This information is provided by atoms of `initially/1`. For example the atom that defines the starting position of block A being *on* block B is `initially(on(block_a,block_b)).`

The narrative of events for the Blocks World is the two *move* events expressed in the `happens/2` atoms: `happens(move(block_a,block_d),5)` and `happens(move(block_b,block_a),10).`

The initial situation and narrative for the Blocks World are shown in code snippet D.6 on page 197 (also refer to figure 7.1 on page 119).

Integrity constraints Two integrity constraints, produced from the fluent trace data were included. These state that fluents are *not clipped* throughout a period of time. These integrity constraints were as follows: $\square \leftarrow clipped(0, clear(block_a), 10)$ and $\square \leftarrow clipped(0, clear(table), 15)$.

Other background knowledge Some other background knowledge for the Blocks World is provided in code snippet D.7 on 198. This includes information on typing[9], as well as information about what fluents do *not* hold at varying time points (examples are presented in code snippet D.8 on page 198).

[8] A *dynamic* literal is one that refers to *time*. In the Event Calculus the literal `holds_at/2` is dynamic.

[9] See definition 16 on page 27.

Examples Observations of dynamic behaviour are the examples for induction. In all Event Calculus programs these examples are atoms of the predicate `holds_at`. For example fluents that hold in the Blocks World at time point 11 are provided as: `holds_at(on(block_a,block_d),11)`, `holds_at(on(block_b,block_a),11)`, `holds_at(on(block_c,table),11)`, `holds_at(on(block_d,table),11)`, `holds_at(clear(block_b),11)`, `holds_at(clear(block_c),11)`, and `holds_at(clear(table),11)`. The complete fluent trace for the Blocks World is shown in code snippet D.9 on page 199.

7.4.2 Results

The domain specific axioms generated by `ALECTO` on the Blocks World data are shown in code snippet 7.10.

```
initiates(move(A,B),on(A,B),C) :-
   holds_at(clear(B),C), holds_at(clear(A),C), diff(B,A).
terminates(move(A,B),on(B,C),D) :-
   holds_at(clear(A),D), holds_at(on(B,C),D), diff(C,A), diff(B,A).

initiates(move(A,B),clear(C),D) :-
   holds_at(on(B,E),D), holds_at(on(A,C),D), diff(E,A), diff(B,C).
terminates(move(A,B),clear(B),C) :-
   holds_at(clear(B),C), holds_at(clear(A),C), diff(B,A).
```

Code snippet 7.10: Completed domain specific axioms produced by `ALECTO` for the Blocks World.

The DSAs produced by `ALECTO` contain definitions for both `initiates/3` and `terminates/3` (two of each). The clauses are similar to the sets produced by `CProgol5.0` in both [Moyle, 1999] and [Wellner, 1999].

A set of domain specific axioms induced in [Moyle, 1999] are reproduced in Appendix D in code snippet D.10 on page 200. Domain specific axioms induced by Wellner in [Wellner, 1999] are reproduced in code snippet D.11 on page 200.

This experiment differs in that `ALECTO` was learning in positive-only mode. In [Moyle, 1999] negative information was provided in the form of constraints, derived from the trace data. Wellner, in a similar fashion, provides negative examples of the target predicate which are derived from the trace data.

7.5 Learning Domain Specific Axioms for the Robot Navigation Control program

In this section, the specific components required for learning the Robot Navigation Control program are described. The experimental method is outlined, and the results presented.

7.5.1 Input components specific to the Robot Navigation Control program

The input components specific to the Robot Navigation Control were as follows:

Mode declarations As with the previous experiment the *head* mode predicates were `initiates/3` and `terminates/3`. The *body* mode predicates contain both dynamic literals and other static domain literals (e.g. `door/3`, `next_visible_corner/3`, `connects/3`, `next_corner/3`, `inner/1`, `corner/1`). The complete mode declarations are presented in code snippet D.14 on page 202.

Inititial Situation and Narrative of events Again, the initial situation is provided by atoms of `initially/1`. One example atom is `initially(loc(corner(c7),ahead))` which defines the starting *loc*ation of the robot to be at corner $C7$, with the wall ahead. An example of a complete initial situation was explained on page 139.

The narrative of events for the Robot Navigation Control program contains many more events than the Blocks World. The following three events are taken from one narrative as an example: `happens(go_straight,13)`, `happens(follow_wall,17)`, and `happens(turn(right),19)`.

An example initial situation and narrative for the Robot Navigation Control are shown in code snippet D.13 on page 201.

Other background knowledge Other Background knowledge for the Robot Navigation Control program includes information on typing, the encoding of the layout of the "model office" (see code snippet 7.7 on page 138, and figure 7.4 on page 134) as well as information about what fluents do *not* hold at varying time points (examples are presented in code snippet D.15 on page 203).

Examples Observations of dynamic behaviour are the fluents that *hold* in the Robot Navigation Control program. Some example fluents holding at time points 5 and 6 are: `holds_at(loc(corner(c7),behind),5)`, `holds_at(in(r2),5)`, `holds_at(facing(east),5)`,

`holds_at(loc(corner(c8),ahead),6)`, `holds_at(in(r2),6)`, and `holds_at(facing(east),6)`.

One complete fluent trace for the Robot is shown in code snippet D.16 on page 204.

Custom Scoring Function

Initial experiments in learning the Robot Navigation Control program using `ALECTO`'s standard positive-only scoring function produced over-general theories. To overcome this problem, a custom scoring function was developed that incorporates a form of *transition minimization*. The aim is to minimize the number of transitions predicted by the domain specific axioms.

The cost function checks for each event the number of predicted fluents that change state, and compares them with the actual number (as seen in the trace data). The code for the scoring function is shown in Appendix D in code snippet D.12 on page 201.

7.5.2 Experimental Method

The series of experiments for generating domain specific axioms for the robot navigation control program were performed in the following manner. The experimental method consisted of: 1) randomly generating trace data; 2) generating DSAs using the data with `ALECTO`; and 3) testing the performance of the generated DSAs on multiple test sets. For these experiments the data for thirty "runs" were generated from thirty pairs of randomly selected start and end corners in the model office.

For each run a plan was constructed [10] that would enable the robot to navigate from the start corner to the end corner (or goal location). The complete Event Calculus program was then run deductively to produce fluent traces. As outlined in figure 7.2 the domain specific axioms were removed and the remaining portions of the program was input to `ALECTO`, which produced a set of clauses.

Each of the thirty generated DSAs was evaluated on the other 29 traces. This evaluation was based on the number of correctly predicted transitions by the induced DSAs compared with those of the original DSAs. For each DSA set an accuracy on each of the other 29 traces was produced, and these were averaged to produce a mean accuracy and variance for the set. This procedure is shown in figures 7.5 and 7.6. The number of training runs (n in figure 7.5) was 30, while the number of test runs (k in figure 7.6) was 29.

[10] It was originally envisaged that the plans would be constructed using Murray Shanahan's *Abductive Planner*. However, this was not possible. The actual plans used here were therefore produced manually and checked with the *Abductive Planner*.

Estimating the accuracy of ALECTO on generating domain specific axioms

Let n be the number of training runs:
for $i = 1$ to n do

1. randomly pick start corner–end corner pair: (s_i, e_i);
2. generate input data E_i;
3. construct DSA set DSA_i with ALECTO$\{E_i\}$;
4. test DSA_i on new data $\mathcal{E}_j$ to get an estimate of accuracy $\hat{a_i} = \frac{\sum a_i}{n}$;

Return $\hat{a_i}$.

Figure 7.5: The experimental procedure for the generation of domain specific axioms by Theory Completion. Note that for the experiments $n = 30$. Step 4 can be refined as shown in figure 7.6.

Estimating the accuracy of a single set of domain specific axioms

Let k be the number of test runs:
for $j = 1$ to k do

1. randomly pick start corner – end corner pair: (s'_j, e'_j);
2. estimate the accuracy of DSA_i using data from (s'_j, e'_j): $\hat{a_j}$;
3. determine variance $\hat{S_j}$;

Return $\hat{a_i} = \frac{\sum \hat{a_j}}{k}$, $\hat{S_i} = \frac{\sum \hat{S_j}}{k}$

Figure 7.6: A procedure for measuring the variance of the DSA recovery procedure. Note that for the experiments $k = 29$.

7.5.3 Results

The results of the thirty training runs are tabulated in Appendix D on pages 205 and 206. Table D.1 presents the accuracies for the 30 induced sets of DSAs for the Robot Navigation Control program using `ALECTO`. Miscellaneous statistics for generating the 30 induced sets of DSAs for the Robot Navigation Control program using `ALECTO` are presented in table D.2

The estimated average accuracy over the 30 runs was 70.2%, with a standard deviation of 6.8%. The average of the standard deviations[11] within each run was 13.1% The average number of clauses in the 30 DSAs generated was 14.

One example of a set of clauses produced by `ALECTO` is the DSAs shown in code snippet 7.11. These compare favourably to the DSAs actually[12] used for deductively generating the traces (see code snippet D.17 on page 207 in Appendix 194).

[11] This is a measure of variance within each run on the 29 test cases.

[12] Some effort was required to transform the original program supplied by Murray Shanahan so that it produced the correct behaviour.

```
terminates(turn(right),loc(corner(A),ahead),B) :-
   holds_at(loc(corner(A),ahead),B), inner(A).

terminates(turn(A),facing(B),C) :-
   holds_at(facing(B),C), change_direction(A,D,B), direction(D),
   direction(B).

terminates(turn(left),loc(door(A),in),B) :-
   holds_at(loc(door(A),in),B), holds_at(door_open(C),B), door(C,D,E),
   corner(D), corner(E).

terminates(turn(left),loc(corner(A),ahead),B) :-
   holds_at(loc(corner(A),ahead),B), corner(A).

terminates(turn(left),in(A),B) :-
   holds_at(in(A),B), holds_at(loc(door(C),in),B), door(C,D,E),
   corner(D), corner(E).

terminates(go_straight,loc(corner(A),ahead),B) :-
   holds_at(door_open(C),B), door(C,A,D), corner(D),
   holds_at(loc(corner(A),ahead),B).

terminates(follow_wall,loc(corner(A),behind),B) :-
   holds_at(loc(corner(A),behind),B), corner(A).

initiates(turn(right),loc(corner(A),behind),B) :-
   holds_at(loc(corner(A),ahead),B), inner(A).

initiates(turn(A),facing(B),C) :-
   holds_at(facing(D),C), holds_at(door_open(E),C), door(E,F,G),
   corner(F), corner(G), change_direction(A,D,B).

initiates(turn(left),loc(door(A),in),B) :-
   holds_at(door_open(A),B), door(A,C,D), corner(D),
   holds_at(loc(corner(C),ahead),B).

initiates(turn(left),loc(corner(A),behind),B) :-
   holds_at(facing(C),B), direction(C), holds_at(loc(door(D),in),B),
   door(D,E,A), corner(E).

initiates(turn(left),in(A),B) :-
   holds_at(in(C),B), holds_at(loc(door(D),in),B), door(D,E,F),
   corner(E), corner(F), connects(D,C,A).

initiates(go_straight,loc(corner(A),behind),B) :-
   holds_at(door_open(C),B), door(C,D,A), holds_at(loc(corner(D),ahead),B).

initiates(follow_wall,loc(corner(A),ahead),B) :-
   holds_at(door_open(C),B), door(C,A,D), next_visible_corner(D,E,B),
   corner(E), holds_at(loc(corner(F),behind),B), corner(F).

initiates(follow_wall,loc(corner(A),ahead),B) :-
   holds_at(loc(corner(C),behind),B), next_visible_corner(C,A,B).
```

Code snippet 7.11: Sample DSAs generated by `ALECTO` for the robot navigation control program.

Learning curves

Two forms of learning curves have been produced to understand how well ALECTO performs Theory Completion of the RNC Event Calculus program. The first form considers the accuracy of theories produced with respect to the length of the sub-sequence of time traces used as input observations. The second form considers the accuracy of the theories with respect to the size of the set of extracted abduced atoms.

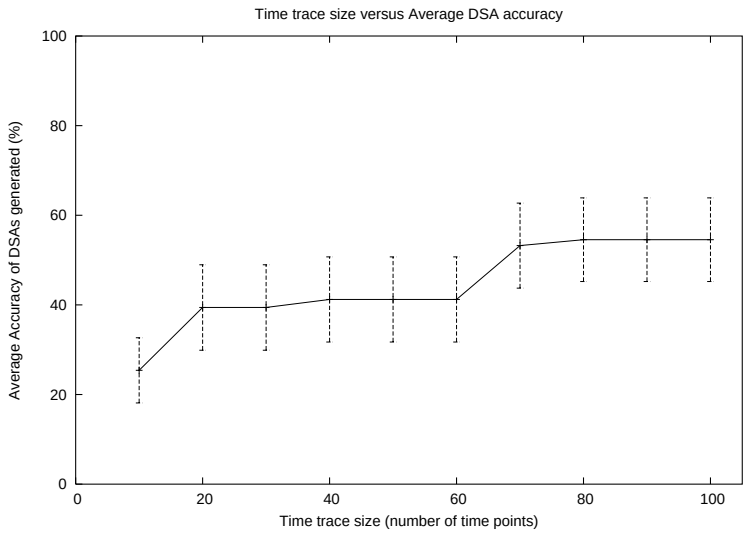


Figure 7.7: A learning curve generated from a single route through the model office. Examples are the number of contiguous time points from which the observed fluents were drawn. The estimates of accuracy are averaged from 30 test sets, while error bars are 1 population standard deviation.

Figure 7.7 plots the accuracy of the generated theories with respect to contiguous lengths of the time traces which are used as Theory Completion inputs for one route through the model office. The accuracy increases with respect to the length of the time traces. This increase is monotonic, however it is not smooth – the step-like changes reflect that the time traces contain portions that do not contain novel state transitions.

Figure 7.8 considers the accuracy of generated theories with respect to the number of abducible observations. In this case the figure reflects combinations of events (i.e. `happens/2`) and changes in the fluent traces (i.e. `holds_at/2`). This plot does not show a clear trend, indicating that it is the inclusion of a particular type of abducible rather than the number of such particular elements produced by ALECTO.

Figure 7.9 plots the run time of both the extraction-case abduction and

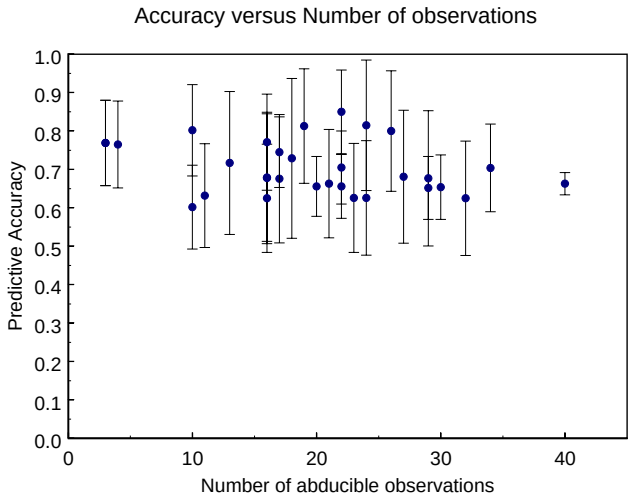


Figure 7.8: Accuracy versus the number of abducible observations (related to the number of events that act on the robot's trace). Note that two data items appear overlaid at the abscissa value of 16 – 67.81 ± 17.14% and 67.90 ± 16.57%.

the induction against the number of abducible observations. This shows trends that indicate that the abduction and induction efforts increase with the number of abduced atoms, and that the two elements are comparable. This is in accordance with the analysis presented in section 4.5.1 (see page 86).

7.6 Discussion

This chapter has demonstrated that Theory Completion techniques can be applied to learning domain specific axioms for Event Calculus programs. A general framework for inducing Event Calculus DSAs was developed. This framework is not specific to the Robot Navigation Control domain, and should be sufficiently general to be applied to other Event Calculus problems.

`ALECTO` was able to produce DSAs for the Blocks World that were comparable to those produced by `CProgol5.0` in earlier work. The amount of preprocessing effort required for `ALECTO` was less than that expended in [Moyle, 1999] and [Wellner, 1999].

The results of generating domain specific axioms for Robot Navigation Control were encouraging. `ALECTO` was able to produce DSAs containing up

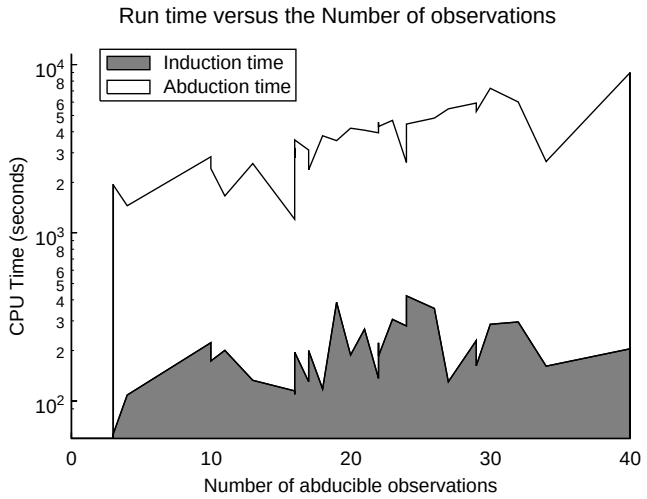


Figure 7.9: The run time of both the extraction-case abduction and the induction versus the number of abducible observations. The experiments were performed on hardware using Intel Pentium III 550MHz processors, and 512 MB RAM.

to 20 clauses, and had an average estimated accuracy of 70.2%.

7.6.1 Related Work

Other researchers have studied the induction of situation-action rules in systems that reason about actions and change. Ben Wellner [Wellner, 1999] uses abductive logic programming techniques in his work on Logic based agents. He employs `CProgol5.0` for the inductive process, and consequently encounters similar problems to those reported in [Moyle, 1999] (where `CProgol5.0` was also used). One serious concern is the need to re-code the two transition predicates *initiates* and *terminates* into a single *flips* predicate. In concluding Wellner suggests applying his techniques to Shanahan's robots.

David Lorenzo-Blanco [Lorenzo and Otero, 1999] [Lorenzo and Otero, 2000] have been investigating learning "action theories" in the alternative formalism, the Situation Calculus [McCarthy and Hayes, 1969] They, too, have discovered the need for Theory Completion. Their initial work has been focused on using `CProgol5.0` on their problem domain.

7.7 Summary

This chapter has demonstrated experimentally that Theory Completion can be effective in generating domain specific axioms for Event Calculus programs that describe the behaviour of dynamic systems. The main results of this chapter are:

- `ALECTO` could be used to generated domain specific axioms to describe the Blocks World. The theory induced was similar to those produced by `CProgol5.0` in earlier work.
- `ALECTO` was able to generate domain specific axioms to control the navigation of a real world robot.
- The Robot Navigation Control programs induced by `ALECTO` had an average accuracy of 70.2% and contained on average 14 clauses.

This concludes the experimental part of the thesis.

Part III

Discussion and Concluding Remarks

Chapter 8

Discussion and Concluding Remarks

8.1 Summary of the Thesis

This thesis has been concerned with machine assisted theory formation in scientific discovery. Theory Completion is considered as the augmenting of an existing theory with new hypotheses so that new observations can be explained. The generalisation of observations to form a theory is common to many forms of Machine Learning, but Inductive Logic Programming is one of the few able to utilise existing theories. Theories in ILP are represented in the form of Logic Programs. Hypotheses produced by ILP are sets of Horn clauses which form the description of the new scientific discoveries. A further advantage possessed by ILP is that these hypotheses can be communicated to humans so that the new discoveries can be understood. Discoveries in science can benefit from two–way interactions between a human and a machine assistant – in which the human provides the assistant with the existing theory, surprising unpredicted observations, and some hunch about the nature of the missing theory components; while the machine assistant communicates its plausible Theory Completions that explain the observations to the human.

Inductive Logic Programming provides theory and methods for proposing hypotheses based on an incomplete existing background theory and some seed observation. The main technique investigated here for this purpose was Inverse Entailment (which was discussed in chapter 2). Another ILP technique used is the enumeration of plausible hypotheses. These hypotheses can then be ranked according to varying criteria. ILP systems have contributed to many scientific discoveries, however they do require a substantial amount of human input. Human input is required to encode the existing theory, the observations, and a raft of settings to encode the scientist's hunches about the "missing theory".

One limitation of the standard ILP setting is that the description of the theories generated are often restricted to being in the same language – or predicate – of the observations. This is called Observational Predicate Learning (OPL). There are, however, circumstances in scientific theory formation where the observations provide evidence for an alternative description that is desired – this is the non-OPL setting. To perform Theory Completion in the non-OPL setting requires extensions to the implementation techniques for performing Inverse Entailment. Two such extensions were investigated in chapter 3: Logical Back-propagation and SOLD-resolution. Both of these techniques have the similar objective of producing a negative explanation for the non explanation of the seed example in the presence of the existing background theory. In logic programming terms this equates to the derivation of negated facts (or goal clauses) from a definite program and a goal clause.

In Logical Back-propagation, to test for the derivation of negated facts, first the background theory is required to be translated and additional clauses for the negated facts added. The derivation of any true negated facts can then proceed using standard SLD-resolution (for example by using a PROLOG system). Logical Back-propagation is a form of constructive negation. When using the second technique, SOLD-resolution, to derive the negated facts, the background theory clauses remain unchanged, but the mechanism for deriving them is altered. To derive the true negated atoms from the background theory and the negated observation a SOLD-resolution system must be used. This, however, is straightforward to implement in PROLOG as a *meta interpreter*. These two methods for extending Inverse Entailment were investigated in chapter 3. Both techniques are related to extraction-case abduction which "extracts" facts to produce an explanation for an observation in the presence of an existing theory.

The techniques of Logical Back-propagation have previously been implemented in the Theory Completion by Inverse Entailment system `CProgol5.0`. An implementation based on the SOLD-resolution method of extraction-case abduction is developed in chapter 4. The resulting system, `ALECTO`, has a similar philosophy to `CProgol5.0`. This is the philosophy of providing simple extensions to the Inverse Entailment approach, to cater for non observation predicate learning. `ALECTO` first uses SOLD-resolution and other techniques proposed by Yamamoto [Yamamoto, 1997] to translate the observations into ground abductive explanations. It is these abductive explanations that are generalised in the usual ILP manner. When evaluating proposed hypotheses `ALECTO` uses a positive-only data scoring function. While searching for a theory, `ALECTO` assumes that all yet to be processed abductions are true. This strategy allows for multiple-predicate learning to be achieved in a serial process, and it is similar to one used in the Abductive Concept Learning system `ACL` [Kakas and Riguzzi, 2000].

Three application domains have been used to investigate the perfor-

mance of the Theory Completion systems `CProgol5.0` and `ALECTO`. The following three application domains were utilised: completing a grammar, completing a semantic intrusion detection system, and learning Event Calculus programs. In these experiments the Theory Completion setting was restricted so that the existing background theories prior to completion contained **no** definitions of the desired predicates. In chapter 5 the first set of experiments study which combinations of deleted definitions can be regenerated for a simple context free grammar. Although simple, the grammar contains enough complexity and subtleties for the purpose of investigating Theory Completion. In these experiments `ALECTO` performed considerably better than `CProgol5.0`, particularly when two definitions were required to be completed "simultaneously".

In chapter 6 the first set of experiments studied single, top-level, deleted definitions and their regeneration from the context sensitive grammar. In the second experiment two deleted top-level definitions were regenerated. The third experiment attempted the recovery of a deleted definition from a lower level.

Two different experiments to complete Event Calculus programs were performed and the results were described in chapter 7. Event Calculus programs have a well defined theory structure which include a set of immutable core axioms, a narrative of events, an initial state, and a set of domain specific axioms. There are strict requirements when writing Event Calculus programs that the *domain specific axioms* are the components to be constructed. When finished, the domain specific axioms can be used, in conjunction with the other components, to generate the desired (hopefully) dynamic behaviour of the Event Calculus program. The structure of an Event Calculus program is particularly well suited to Theory Completion. The experiments in chapter 7 were designed to determine whether Theory Completion techniques could be used to produce the domain specific axioms from the core axioms – or *empty theory* – of the Event Calculus and observations of desired dynamic behaviour.

The work on learning Event Calculus programs was, in part, motivated from earlier work [Moyle and Muggleton, 1997; Moyle, 1999], in which `CProgol5.0` was used to generate domain specific axioms for the simple Blocks World Event Calculus program. Limitations in using `CProgol5.0` included the need to to pre-process the data to produce forms of negative observations. The most serious concern, however, was the need to re-code the two Event Calculus transition predicates *initiates* and *terminates* into a single *flips* predicate. `ALECTO`, as reported in section 7.4, was applied to a data set used in [Moyle and Muggleton, 1997] and achieved similar results to the earlier work, but without the need for re-coding, nor the need for the construction of negative information.

Section 7.5 described the application of `ALECTO` to learning a more substantial Event Calculus program – a program to control the navigation of

a real-life robot. The original control program is part of a suite of robot programs incorporating sensing, planning, and acting – all written in the Event Calculus. The robot navigation control program was provided by Murray Shanahan and is detailed in [Shanahan and Witkowski, 1999]. The robot navigation control program consists of three sub-components: 1) the domain specific axioms that describe the direct effects of actions in specific situations; 2) definitions for ramifications – or indirect effects of actions; and 3) background knowledge consisting, in part, of the layout of a model office (the robot's environment). The original program was manually adapted to a slightly different formulation of the Event Calculus, and also to account for a wider class of dynamic behaviour. With the alterations the robot navigation control program consisted of 13 pairs of initiates/terminates domain specific axioms.

The data for the experiments in learning the robot navigation control program was generated by executing all of the components together to produce time trace data of the dynamic behaviour of the robot. Theory Completion was then used to take the "existing" Event Calculus program (emptied of the domain specific axioms), and the time trace data to induce replacement domain specific axioms. `ALECTO` was applied to 30 different runs where each run consisted of observing the (simulation of the) robot navigate through the model office from a random start point to a random finishing point. The domain specific axioms produced from each run were evaluated on the remaining 29 data sets. The estimated average accuracy of the `ALECTO` generated domain specific axioms over the 30 runs was 70.2%, with a standard deviation of 6.8%. The average of the standard deviations within each run was 13.1% The average number of clauses in the 30 domain specific axiom sets generated was 14.

The development of the application of `ALECTO` to the learning of Event Calculus programs has produced a framework which should be applicable to other Event Calculus settings. The experimental results achieved by `ALECTO` in learning Event Calculus program suggest the possibility that discoveries of dynamic behaviour for some natural phenomenon may, in the future, be assisted by a machine.

8.2 Discussion

8.2.1 Theory Completion Techniques in ILP

This thesis has focused on extending ILP techniques based on Inverse Entailment for use in the non-OPL setting. Inverse Entailment is a technique for proposing a most specific hypothesis from a single observation and the existing background theory. Typically the head of this hypothesis contains the observation itself, while the body contains statements that are true about objects related to the observation. The "truth" of these related statements

are contained in the existing background theory. Only hypotheses that are more general than this most specific hypothesis are proposed as plausible explanations for the observations. It is hoped that "good" plausible hypotheses will also explain other observations. The limitation of this method is that all hypotheses considered are in the language – or predicate – of the observations.

The generation of a most specific hypothesis with a head different from the predicate of the observation requires the observation to be used as evidence for producing a related fact. This related fact, like the observation, is not explained directly by the existing background theory. In a sense the related fact is logically extracted – or abduced – from the unexplained observation and the existing background theory. This is known in Abductive Logic Programming as extraction-case abduction. In the case of inverse entailment, it requires negative information to be derived from the negation of the unexplained observation and the existing background theory. This thesis considered two techniques for extracting related facts from an observation and an existing background theory: Logical Back-propagation, and SOLD-resolution.

The first method investigated for performing Theory Completion by Inverse Entailment (TCIE) is Logical Back-propagation. To produce a related negated fact using Logical Back-propagation, first the existing background theory must be augmented with transformations. These transformations (based on techniques developed by Stickel [Stickel, 1992]) allow explicit negative information to be derived using standard techniques (SLD-resolution in PROLOG). Logical Back-propagation has been implemented in the TCIE system `CProgol5.0` (first reported in [Moyle and Muggleton, 1997] and presented afresh in [Muggleton and Bryant, 2000]). `CProgol5.0` is capable of producing most specific hypotheses with head predicates different from those predicates that are used to describe the observations. More general hypotheses than the most specific one are enumerated during a search process. The hypotheses returned by `CProgol5.0` are those which cover the given observation with the maximal information compression.

`CProgol5.0` contains a "Multi-predicate Search Algorithm" [Muggleton and Bryant, 2000] which performs serial searches for hypotheses in clauses, each potentially described using alternative head predicates. In effect, the algorithm performs a greedy search through each of the subsumption lattices having alternative most specific hypotheses. For each of these most specific hypotheses there must be a single negated fact, that can be derived via the observation and the augmented existing background theory, for use as its head. Although `CProgol5.0` generalises observations in multiple-predicates it does not perform multiple-predicate learning in the usual sense [De Raedt *et al.*, 1993].

`CProgol5.0` seems best suited to non-OPL where the predicates produced are independent in the original background theory. Previous applica-

tions of CProgol5.0 where predicates to be generated were closely related [Moyle and Muggleton, 1997] required the predicates to be re-coded into a single predicate. Other limitations in CProgol5.0 are: inflexible scoring; and difficulties in communicating domain knowledge about the acceptability of abduced facts from the seed observations[1].

The second method investigated for performing Theory Completion by Inverse Entailment (TCIE) is SOLD-resolution proposed by Yamamoto [Yamamoto, 1997]. To produce a related negated fact from an observation using SOLD-resolution *no* transformations to the existing background theory are required. Instead, this technique requires that the theorem proving mechanism be changed – that is SLD-resolution be replaced with SOLD-resolution. This is not a significant burden, as a SOLD-resolution interpreter can be easily programmed as a PROLOG meta-interpreter.

Logical Back-propagation generates *single atom* abductive explanations. SOLD-resolution, however, is able to generate *sets of atoms* to form abductive explanations. It is shown that all abductions produced by Logical Back-propagation can also be generated by SOLD-resolution; but the reverse does not hold.

SOLD-resolution is implemented in the Theory Completion system ALECTO – an extension of the Inverse Entailment based ILP system ALEPH. ALECTO uses SOLD-resolution and an existing background theory to transform the observations into sets of abductive explanations. Constraints must be provided by the inductive logic programmer to ensure that the explanations are "acceptable". It is the sets of abductive explanations that are later generalised to produce plausible hypotheses. The plausible hypotheses are guaranteed to explain the abductive explanations, and thus also explain the observations.

Multiple predicate learning is achieved in ALECTO by assuming, during the search of each hypothesis lattice, that all yet-to-be-generalised explanatory facts are true. In a sense, this ensures that some statements that need to be true in the generation of a hypothesis are available, regardless of whether the statements are derivable from the existing theory. A similar strategy is used in a scoring function of ranking hypotheses in the Abductive Concept Learning system ACL [Kakas and Riguzzi, 2000]. When ranking plausible hypotheses ALECTO uses a positive-data only scoring function whereas ACL needs negative examples.

The transformation of observations into abductive explanations has the following strengths. First, it is possible to obtain sets of negated atoms when there is no single negated atom that can explain the observation. These atoms, quite naturally, can be used to derive multiple most specific clauses. Second, it potentially provides a link between the different most specific

[1] In the latest release of CProgol5.0 examples are provided that illustrate its use in providing acceptability constraints that override the default constraints.

hypotheses through the shared variable bindings of the abduced atoms.

Aspects of the Theory Completion mechanisms of the systems `CProgol5.0` and `ALECTO` are similar to mechanisms used in the Abductive Concept Learning system `ACL`. For example `ACL` interleaves processes for abduction and induction in the presence of existing background theory and constraints. The objectives of `ACL` are much broader than those of the ILP systems investigated here. In particular, `ACL` is not constrained to learning Horn theories, but is capable of generating an abductive logic program. `ACL` was not applied to the data sets used in this thesis (see further work in section 8.4).

Abductive logic programming is typically concerned with selecting the *conditions* (i.e. a set of abducibles) under which one may reason with the background theory ([Flach and Kakas, 2000], page 20). These conditions form one explanation of the observed instance with respect to the background theory. In inductive logic programming *generalisations* are sought rather than explanations. ALP can be used to provide a service to ILP by utilising common background knowledge to form sets of explanations, that can then be generalised by ILP. This is the process that was performed in the work reported here. Some work in ALP is also interested in the generation of *abductive logic programs* [Kakas and Riguzzi, 2000], which is not the focus of this work.

SOLD-resolution [Yamamoto, 1997] is one computational model for extracting explanations for observations in the presence of a background theory. There are numerous alternatives including "resolution with residues" where explanations are collected in "dead-end style" clauses (as championed by Stickel [Stickel, 1991], e.g. [Cox and Pietrzykowski, 1986], [Cox and Pietrzykowski, 1987]); "unfolding" of the theory (e.g. [Console *et al.*, 1991]); and "constructive abduction" (e.g. SLDNFA [Denecker and Schreye, 1985], ACLP [Kakas and Michael, 1995]) (see further work in section 8.4).

8.2.2 Applications requiring Theory Completion

The form of applications requiring Theory Completion discussed here are restricted to the setting of non-OPL – where an existing background theory is present. Problems conforming to this setting are often well structured theories (or models) with the following properties: 1) the theory contains clearly separable immutable and mutable components, 2) the immutable theory relies on results that are provided by the mutable theory components (that is the predicates of the mutable theory are "called" by the immutable theory), and 3) observations can only be collected in the language of the immutable theory. Clearly the observations cannot be used as evidence for completions to the immutable theory. The observations are therefore used as evidence for alterations in the "lower" levels – or the mutable theory.

In ILP one commonly used domain for non-OPL is that of "extracting" grammars from examples of well formed sentences (see for example [Muggle-

ton and Bryant, 2000] and [Muggleton *et al.*, 1996]). The grammar problems are often framed with the three characteristics outlined above. First, there are some high level components that describe *sentences*. These broadly define some sub-components of sentences, and are taken to be fixed. New sentences, unable to be parsed by the grammar, are then used as evidence for additions to the sub-components. The existing theories for the grammars studied in both [Muggleton and Bryant, 2000] and [Muggleton *et al.*, 1996] have structures where *sentence* it the top level predicate, and also that the calling diagram for the existing theories are all *directed acyclic graphs*.

The non-OPL domain of Functional Genomics reported in [Muggleton and Bryant, 2000] has a structure in accordance with the characteristics mentioned at the beginning of the section. A model exists to predict the *phenotypic effects*[2] of particular enzyme genes on particular growth mediums, which clearly relates the gene codes of enzymes. What the scientist desires is a theory for how the codes of the genes operate, but only the phenotypic effects are observable. Again the background theory for this domain is represented in a logic program that is a directed acyclic graph with the observable predicate "above" the predicate of interest. Muggleton and Bryant reported the performance of `CProgol5.0` on recovering varying proportions of deleted definitions for the single predicate `codes/2`.

In [Parson *et al.*, 1999] a synthetic domain of a program that generates base-n strings was constructed to test Theory Recovery (a similar problem to Theory Completion). Here, too, `CProgol5.0` was used to recover varying proportions of deleted predicates. In this work, however, only single predicates were removed within any one experiment. The structure of the existing synthetic theory is also a directed acyclic graph.

In the applications mentioned above the results for completing existing background theories were poor when *no* definitions for the target predicate were contained in the existing background theories. Not only does the setting of learning Event Calculus programs have *no* definitions for the target predicates in the initial theory, but it also requires that *two* predicates be completed simultaneously. When learning Event Calculus programs the immutable component of any existing theory are the *core axioms*. The mutable components – the desired predicates for completion – are the domain specific axioms. The observations, however, are recorded in a single different predicate.

The simple Event Calculus program describing the Blocks World has been generated in a Theory Completion setting previously ([Moyle and Muggleton, 1997; Moyle, 1999; Wellner, 1999]). All three efforts were based on using `CProgol5.0`. The difficulties encountered reported in [Moyle, 1999] included: 1) having to significantly reformulate the Event Calculus so that domain specific axioms were encoded in a single predicate; 2) the need to

[2]The change in characteristics of a phenotype or an individual organism.

learn domain specific axioms for each state variable (or fluent) separately; 3) the lack of control over the bindings of abduced atoms; and 4) the use of an inappropriate cost function for scoring clauses. Further criticisms of the work, made in [Wellner, 1999], include that the experimental problem itself was too restricted; and that too much preprocessing of the data was required.

Wellner [Wellner, 1999] uses a similar abductive approach to the one taken in this thesis. However, as his final Theory Completion step is delegated to `CProgol5.0` his work requires that the Event Calculus be re-coded into a single predicate. His abductive extraction technique differs from the work here. Wellner uses an abductive "refinement policy" that generates a *union* of all possible abductive explanations to represent the set of observations, whereas in this thesis only one explanation per observation is used to construct the explanation set. In common with the other domains to which Theory Completion has been applied, the Event Calculus structure is a directed acyclic graph.

Other researchers have applied ILP techniques to learning action theories. Lorenzo and Otero ([Lorenzo and Otero, 1999] , [Lorenzo and Otero, 2000]) have investigated learning theories in the Situation Calculus [McCarthy and Hayes, 1969]. Their initial work also applied `CProgol5.0` to the Blocks World problem. Reddy and Tadepalli [Reddy and Tadepalli, 1997] used ILP to learn descriptions of the Blocks World in a planning system formulation. Sablon [Sablon, 1995] used ILP techniques in a planning system that performs some learning. The ILP approach taken in the work was based on *Iterative Version-spaces* [De Raedt, 1992; Mitchell, 1982] rather than Inverse Entailment.

This thesis, however, demonstrates that a more significant Event Calculus program than the Blocks World description can be generated by theory completion. The `ALECTO` TCIE system was able to achieve an average accuracy of 70.2% in the generation of domain specific axioms to control the navigation of a real-world robot. The theories produced contained clauses in two predicates (*initiates* and *terminates*) (while the observations were in a single predicate *holds_at*), and an average of 14 clauses. To produce these theories required both extraction-case abduction from observations, and multiple-predicate learning. Assistance was also provided by encoding appropriate domain knowledge as hypothesis structure constraints, and providing a customised scoring function. The tailoring of the Theory Completion system in this way to solve a specific class of problems is similar to the developments required in the learning of scientific theories in bio-chemical domains (see for example the machine assisted phamacophore discovery reported in [Finn *et al.*, 1998]). The "tuning" of the Theory Completion system for the robot navigation control problem has resulted in a setting likely to be suited to the learning of other Event Calculus programs.

The application of `ALECTO` to learning the robot navigation control pro-

gram answered the earlier criticisms thus: 1) there was no need to significantly reformulate the Event Calculus so that domain specific axioms were encoded in a single predicate; 2) the domain specific axioms referring to different fluents were produced in the one run, and not separately; 3) acceptability constraints provided appropriate control over the bindings of abduced atoms; and 4) a custom cost function was used for scoring hypotheses. Finally, the robot navigation control problem was clearly more significant than the Blocks World, and very little preprocessing of the data was required.

8.2.3 Logical Styles of reasoning and uses of the Event Calculus

In chapter 1 three modes of "logical" reasoning were discussed that followed the writings of C.S. Peirce (refer to section 1.3.1 on page 9). These modes were deduction, abduction, and induction. There are parallels to these modes in the different ways in which an Event Calculus program can be used. An Event Calculus setting can be used for 1) forecasting the state of a dynamically changing system, 2) planning the set of actions required to change the initial state of the system into some desired goal state, and 3) to learn the description of the dynamic behaviour of the system.

The process of forecasting the state of system from a complete Event Calculus program consisting of core axioms, domain specific axioms, a narrative of events, and an initial situation can be performed by *deduction*. To generate an ordered set of events – or a plan – that will transform the system from an initial situation into a desired goal state can be performed by *abduction*. A plan is an explanation as to what events are needed to transform the system from its initial state to the desired goal state (see also [Shanahan, 1989]). Finally, *induction* can be used to generate domain specific axioms when a trace of dynamic behaviour is combined with the narrative of events that produced the trace. The formulation of domain specific axioms is akin to programming. These processes are illustrated in table 8.1.

The ability to formulate domain specific axioms is crucial to using Event Calculus programs. Without them plans cannot be formed nor can predictions about how the system will respond to actions be made. With appropriate domain specific axioms in place Event Calculus planning systems (for example the one detailed in [Shanahan, 2000]) will have some "logical" basis with which to work. Machine assistants may, in the future, be a valuable tool in the inductive formulation of domain specific axioms.

8.2.4 Learning common sense knowledge

Philosophers and practitioners of artificial intelligence have argued that common sense is one of the core components of intelligence [McCarthy, 1968;

Table 8.1: A comparison of the modes of reasoning, as outlined by Peirce, to the different ways in which an Event Calculus program can be used. The Event Calculus components are labelled as follows: Core Axioms – T_{Core}, Narrative of events – $T_{Narrative}$, state of the system at time point i – T_i, domain specific axioms – T_{DSAs}, and the symbol $\Rightarrow$ is read as "producing". Other miscellaneous background theory is not shown.

Event Calculus program usage		*Mode of Reasoning*
Prediction (control)	$T_{Core} + T_{Narrative} + T_0 + T_{DSAs} \Rightarrow \quad T_1, \ldots, T_n$	Deduction
Planning	$T_{Core} + T_0, \ldots, T_n + T_{DSAs} \Rightarrow \quad T_{Narrative}$	Abduction
Learning Dynamic Behaviour	$T_{Core} + T_{Narrative} + T_0, \ldots, T_n \Rightarrow \quad T_{DSAs}$	Induction

McCarthy and Hayes, 1969; Ryle, 1949]. One of the central problems of AI, which has been recognised since the inception of the field [McCarthy, 1959], is how to endow a computer program with common sense. The importance of common sense techniques in AI is advocated by Davis [Davis, 1990] in the following passage.

> ... all parts of artificial intelligence - vision, natural language, robotics, planning, *learning*, expert systems, - must, sooner or later, make use of common sense knowledge and common sense reasoning techniques.[3] (page 21)

What then is *common sense* and can it be learned by machine? McCarthy considers a system to have common sense ([McCarthy, 1968] page 403)

> ... if it automatically deduces for itself a sufficiently wide class of immediate consequences of anything it is told and what it already knows.

It is possible to use the three components in McCarthy's definition as a test for success when attempting to learn common sense.

1. The system should be capable of *automatically deducing* consequences.

[3]Italics added for emphasis.

2. The system should use information that it is told, and information that it *already knows*. Any addition or alteration to this information will be considered as learning, providing that the third part is satisfied;

3. After learning has occurred the system should be able to derive a *sufficiently wide class of immediate consequences* that were not derivable prior to learning.

The first two components are satisfied by using the Event Calculus as the domain description language. The learning of Event Calculus domain specific axioms would satisfy the third component. The experiments reported in this thesis demonstrate that non trivial Event Calculus programs can be induced from dynamic observations. It is therefor argued that learning knowledge in a common sense formalism has been achieved.

8.3 Limitations

This section considers some limitations of the thesis. These limitations are discussed on different levels ranging from the theoretical and technical levels, to philosophical levels.

At the mundane technical level there are limitations in the use of SOLD-resolution as a method for Theory Completion. SOLD-resolution implements *relative subsumption* [Plotkin, 1971] between clauses which is weaker than relative-implication. Even so, this seems to be a reasonable compromise between tractability and efficiency. Furthermore, knowing what statements to *skip* – or abduce – based on the existing theory and an observation requires human intervention. Machine assistants that minimize human intervention, are preferred to those assistants that don't.

The efficiency of `ALECTO`, particularly in the extraction-case abduction of observations into explanations is particularly poor. The number of potential crosslists is exponential in the number of observations. This was also apparent in some of the experiments where the majority of the computation time was used in the generation of the crosslists. The actual generalisation time in some experiments was minor in comparison, typically being an order of magnitude less. These efficiency results were not unexpected, and are broadly in line with the analysis in section 4.5.1 on page 86. Indeed the relative proportion of effort spent on abduction and induction is accordance with comments in [Flach and Kakas, 2000] (page 24):

> ... the observed computational differences between generating abductive hypotheses and generating inductive hypotheses are likely to become smaller .... Much of the computational effort of ILP systems is spent on efficiently searching and pruning the apace of possible hypotheses, while ALP systems typically

> enumerate all possible abductive explanations. The latter approach becomes clearly infeasible when the abductive hypothesis space grows. ...

At an implementation level, ALECTO does not provide support for non-ground non-unit observations. The theory behind SOLD [Yamamoto, 1997], however, does account for that case.

The current implementation of ALECTO does not make use of all available domain knowledge when performing abductions. For example the underlying SOLD mechanism (and that of PROLOG in general) is not based on any type theory. By encoding a type theory and the functional dependencies of arguments in a manner similar to Constraint Logic Programming (CLP) [Jaffar and Lassez, 1987] and using this as part of the extraction-case abduction mechanism the search space for abductions would be reduced. It would also be expected that more appropriate abducibles could be produced with improved efficiency. Efforts to incorporate ALP and CLP are reported in [Kakas and Michael, 1995].

At a broader lever, Theory Completion, by definition, is limited to simply extending an existing theory. It is not able to modify the existing theory in light of new evidence provided by the new observations. A system that provides full support to a scientist should also be able to *revise* existing theories if necessary. A full Theory Revision system must be able to recognise that the existing theory is incorrect, diagnose which part(s) of the theory need to be replaced, and suggest what the replacement theory should be.

Using ILP as the basis for machine assisted scientific discovery has its own limitations. ILP is restricted to representing problems in a subset of first order logic (logic programs). Although a powerful description language, first order logic is not always suitable to describing every type of problem. For example problems that require reasoning with uncertainty, or forms of higher level reasoning (like reasoning by analogy). Present ILP systems suffer from very low levels of usability. A scientist aiming to use a machine to assist his discoveries should not be expected to become an expert inductive logic programmer (which itself requires him to become an expert in logic programming!). Although touted as a strength of ILP, the communicability of results is also a weakness. Humans not familiar with the interpretation of logic programs find it difficult to understand the results from ILP systems. The machine assistant should, therefore, be able to "speak the language" of the scientist. The scientist should *not* be forced to speak the language of the assistant[4].

Theory Completion as a method of machine assisted scientific theory formation has nothing to say about *how* and *when* to propose a complete change of *paradigm* [Kuhn, 1962]. At what point should a machine assis-

[4]Some effort has been put into translating the ILP system's results into the language of the scientist in [Finn *et al.*, 1998] for example.

tant *revolt* against an existing theory and discard it all together? How, too, should such an assistant formulate an alternative theory to the one discarded?

8.4 Future Directions

This section proposes possible directions for future research. These proposals are broadly classified into the following areas: 1) theory and methods; 2) application domains and settings; 3) other systems for scientific theory formation; and 4) evaluating the learned robot navigation control program.

8.4.1 Theory and methods

This section lists potential future work relating to the theory and implementation methods of Theory Completion.

Alternative methods for generating abductive explanations Investigate different methods for generating abductive explanations. There are numerous alternatives including "resolution with residues" (e.g. [Cox and Pietrzykowski, 1986], [Cox and Pietrzykowski, 1987]); "unfolding" of the theory (e.g. [Console *et al.*, 1991]); and "constructive abduction" (e.g. SLDNFA [Denecker and Schreye, 1985]). The choice should be governed by efficiency concerns. One promising area would be to combine Constraint Logic Programming so that domain knowledge can be utilised in the resolution process to constrain the solution space (e.g. ACLP [Kakas and Michael, 1995]).

Efficiency Improvements The first target for efficiency improvements would be methods for generating abductive explanations (see previous item). Two levels of improvement are sought: 1) generic methods that increase efficiency on every domain; and 2) specific methods or encodings that produce improvements on specific background knowledge. For example, the standard form of the Event Calculus is known to be extremely computationally inefficient due to much computational redundancy introduced by negation-as-failure ([Shanahan, 1997] page 239).

Incorporating type information Incorporate the type theory and the functional dependencies of arguments from the domains in a manner similar to Constraint Logic Programming (CLP [Jaffar and Lassez, 1987]). Utilise this mechanism for extraction-case abduction to produce more appropriate abducibles and also improve the efficiency by reducing the space of abducibles.

The structure of Existing background theories Investigate the structural topology of existing background theories that can be used in Theory Completion. This thesis has only considered existing background theories whose logic programs have calling diagrams that are directed acyclic graphs. What structural properties of the existing background theories are important when using extraction-case abduction to generate explanations?

Orderings over explanations The problems studied here have been restricted so that the abducible components are tightly specified. In the case that many alternative abductive explanations can be produced from a single observation, is there a preference ordering over the explanations? More specifically should the explanations that contain atoms from lower in the calling diagram be preferred to those above?

Further extensions from other areas What further simple extensions can be usefully adapted from other logic programming areas of research for Theory Completion? For example the areas of Theory Revision and Abductive Logic Programming. In particular are theory revision operators appropriate? Is the ability to generate constraints from the observations and the existing background theory (e.g. the `ACL` system described in [Kakas and Riguzzi, 2000]) useful in machine assisted scientific theory formation?

Hypothesis search stragies This thesis has used a sequential search for clauses when performing multi-predicate learning. Future work could investigate methods for "simultaneous" parallel searches across multiple lattices.

Noise handling The abductive explanations in this thesis have been generated using accurately classified observations. In what ways would the introduction of *noise* alter the setting? What noise handling techniques would be best suited to this problem?

Non-ground–non-unit observations Investigate the implementation and use of non-ground and non-unit observations. The theory has already been developed to utilise such observations by Yamamoto in [Yamamoto, 1997]. This theory should be easily implemented in `ALECTO`. However, finding real life applications to test these extensions may be difficult.

8.4.2 Application domains and settings

In this section, some obvious applications and settings are suggested for future work. Applications relating to Event Calculus program problems are also discussed.

Settings where definitions already exist `ALECTO` could be applied to the setting where some definitions of the target theory are already contained in the existing background theory. These results could then be compared to the results produced by other systems (in particular `CProgol5.0`). The possible data sets for use are 1) other grammars (from [Muggleton and Bryant, 2000] and [Muggleton *et al.*, 1996]), functional genomics (also from [Muggleton and Bryant, 2000]), and a synthetic domain of a program that generates base-n strings (from [Parson *et al.*, 1999]).

Medical domain The medical domain of *Endocrinology* may be suited to Theory Completion. Researchers have proposed detailed theoretical models with complicated metabolic pathways [Smythe *et al.*, 1982]. Theory Completion may be a useful way to expand these models.

Compare the performance of `CProgol5.0` on the robot data Future work could attempt to apply `CProgol5.0` to the Event Calculus robot navigation control data set. This will require some re-coding of the data.

Learning models to control ship damage Vadim Bulitko [Bulitko and Wilkins, 1999] produced an intelligent agent for real-time crisis decision making to reduce ship damage. The decision making task of ship damage control includes addressing problems such as fire spread, flooding, smoke, equipment failures, and personnel casualties. Components of the intelligent agent were learnt from real and simulated data. Much of the data was of a temporal nature. As well as the time traces of dynamic variables, static domain features were also present. For example the layout of the rooms, and decks on the vessels were available. Bulitko reported numerous difficulties when using `CProgol` on the problem, many of which were similar to the difficulties experienced on the application domains in the thesis. The ship damage control modelling problem is similar, in essence, to the robot navigation control problem, and may be able to be represented in the Event Calculus. Future work could include attempts to learn Event Calculus programs of ship damage control using the Theory Completion system `ALECTO`.

Learning ramifications in the Event Calculus In the experiments in learning Event Calculus programs, the target clauses were those that encoded the *direct effects* of actions. It may be possible to learn predicates to describe the *indirect effects* of actions, known as *ramifications*. Ramifications are fluents whose values are determined by the value of other fluents rather than being determined by the direct effect of actions. The ramifications can lie deeper than the domain specific axioms in the calling diagram of Event Calculus programs. For example

one ramification predicate in the robot navigation control program is `next_visible_corner`.

Noisy data in learning Event Calculus programs The data used in learning Event Calculus programs in this thesis contained no errors. How would `ALECTO` perform if it was provided with simulated noisy data?

8.4.3 Other systems

Some other systems that utilise observations for adapting existing theories have similar aims and features to those studied in this thesis. Further work could investigate and compare the features of such systems. For example `ACL` [Kakas and Riguzzi, 2000], and Yamamoto's own implementation of SOLD-resolution, `BORDA` [Ito and Yamamoto, 1998]. The systems could be applied to the problems in this thesis.

8.4.4 Evaluating the robot navigation control program

It would be interesting to test some of the `ALECTO` generated robot navigation control programs on Murray Shanahan's real life robots[5]. Perhaps dynamic observations of these "misbehaving" robots could be used by a machine assistant to improve these "initial" control programs.

8.4.5 Other

Further work might also be aimed at investigating the relationship between Theory Completion and "automatic programming" or "programming by example". Cypher [Cypher, 1993] states that programming by example is motivated by the following scenario.

> "... if a user knows how to perform a task on the computer, ... the user should be able to instruct the computer to "Watch what I do", and the computer should create the program that corresponds to the user's actions."

Simply recording a "macro" of the user's behaviour is not very useful, as the macro will only be able to be used under identical circumstances to those from which it was recorded. Of more use would be a generalised version of the macro that provided a wider range of applicability. In this thesis a robot was "watched" and a program for its behaviour was generated. Perhaps Theory Completion could be used as the basis for a tool for program synthesis, and even software reverse engineering.

[5] I hope it would not cause them too much harm!

8.5 Concluding Remarks

This thesis has been concerned with machine assisted theory formation and revision in scientific discovery. In the author's opinion it is evident that a machine assistant to aid scientific theory discovery should be able to:

- "Understand" existing background theories;
- Assimilate new observations with respect to the existing background theory;
- Propose useful hypotheses that explain the observations, with respect to the background theory;
- Reason with and improve its "common sense".

The features above have been investigated in this thesis, but the investigation has be confined to Theory Completion in Inductive Logic programming. These features, however, are only the first steps towards a machine assistant.

Furthermore a machine assistant should be:

- Easy for humans to interact with;
- Efficient at proposing theories;
- Capable of presenting its theories in a manner so that humans can understand what it has "discovered";
- Capable of suggesting alterations to the existing theory, and in the extreme case, suggest that the existing theory should be abandoned.

These areas require much more research. Efforts to produce better machine assistants for scientific theory formation should continue. Researchers should follow the advice given by Buddha in his final words and

> ... strive on with diligence.

Appendix A

List of symbols

Logic Programming

,	Disjunction (of literals)
$\vee$	Disjunction (of literals)
$\wedge$	Conjunction (of literals)
$\leftarrow$	Implication (within a clause)
$\cap$	The intersection (of clauses)
$\cup$	The union (of clauses)
$\emptyset$	The empty set (of clauses)
$\models$	Logical entailment
$\not\models$	Not logically entails
$\square$	The empty set of models (or false)
$\mathcal{M}_P$	The minimum model of program P
$\mathcal{M}^+(P)$	The minimum model of program P
$\mathcal{M}^-(P)$	The elements outside the minimum model of program P
$\overline{C}$	The negation of clause C
$\mathcal{P}_P$	The set of predicate symbols in program P
$\mathcal{F}_P$	The set of function symbols in program P
$\mathcal{V}_P$	The set of variable symbols in program P
$\mathcal{C}_P$	The set of constant symbols in program P

Inductive Logic Programming

B	The existing background theory, or background knowledge
C_i	A clause
E	A set of examples
E^+	A set of positive examples
E^-	A set of negative examples
I	A set of integrity Constraints
$\mathcal{H}$	The set of all hypotheses
$H_{(e,B)}$	An hypothesis clause that entails example e and relative to background knowledge B
$\bot$	Most specific hypothesis, bottom clause, or bottom
MSC_{e^+}	The most specific hypothesis produced from seed example e^+
$\overline{\bot}$	The negated most specific hypothesis
$\geq$	Quasi-order
$C\theta$	The result of applying the substitution θ to clause C
$\succeq$	θ-subsumption
$\geq_B$	generalised-subsumption with respect to background theory B
$\succeq_B$	relative-subsumption with respect to background theory B
$\models_B$	relative-implication with respect to background theory B
$\succ$	Proper θ-subsumption
$\sim$	Equivalence under θ-subsumption
$[C]$	The set of all clauses that are subsume-equivalent to C
$\sqcap$	The greatest lower bound (between two clauses in the θ-subsumption lattice is the most general instance)
$\sqcup$	The least upper bound (between two clauses θ-subsumption lattice is the least general generalisation)
ρ	A refinement operator
τ_t	A predicate for generating terms of type t
f	The pseudo information compression measure
p	The number of positive examples covered by an hypothesis
n	The number of negative examples covered by an hypothesis
l	The number of literals in an hypothesis
h	The optimistic estimate of the number of literals required to complete the input-output of an hypothesis
τ	The argument type for a moded predicate
c	The maximum clause length permitted during search
i	The maximum *i-depth* permitted during saturation
$I_{\mathcal{L}}$	The hypothesis structure constraints for language $\mathcal{L}$
$D_{\mathcal{H}}$	The probability distribution associated with all possible hypotheses
$D_X(H)$	The probability distribution over all observations X by hypothesis H
m	The number of (positive) examples
r	The number of random observations
$g(H)$	The generality of an hypothesis H
$sz(H)$	The size of an hypothesis H

Theory Completion

T_{fix}	The fixed (immutable) portion of an existing background theory
T_{mod}	The mutable portion of an existing background theory
O	The set of observations
$\neg l_j$	A Stickel literal of literal l_j
C'	A Stickelised clause
$\neg B$	A Stickelised version of theory B
p^*	The `CProgol5.0` "star" version of predicate p (for grounding literals)
$LBP(B, e^+)$	The negated atom produced by Logical Back-propagation from existing background theory B and example e^+
I_{acc}	Acceptability constraints on abductive explanations
Δ	A set of abduced atoms
$SOLD(B, \overline{e_i^+})$	The goal produced by SOLD-resolution from existing background theory B and goal $\overline{e_i^+}$
S	A skiplist
H	A h-list (ground skiplist)
χ	a crosslist

Event Calculus

T_{EC}	An Event Calculus program
T_{Core}	The Event Calculus core axioms
T_{DSAs}	The domain specific axioms of an Event Calculus program
$T_{Narrative}$	The narrative of events of an Event Calculus program
T_i	The state of the Event calculus program at time point i
Π	A set of time points
Δ	A set of events
Φ	A set of fluents
β	A fluent occurrence
α	An event occurrence

Appendix B

Appendix to Chapter 5

This appendix provides code listings and supporting material to chapter 5.

B.1 Observations available for completing the grammar

When performing positive-only learning, two types of "observations" are required to be input into `ALECTO` and `CProgol5.0`: 1) observations as *examples* of the concept to be learned, and 2) random observations selected from the distribution of all possible observations. For the grammar problem, these can be found in code snippets B.1 and B.2.

```
sentence([the, president, has, the, president] ,[]).
sentence([the, president, has, the, cat] ,[]).
sentence([the, president, has, a, president] ,[]).
sentence([the, president, has, a, cat] ,[]).
sentence([the, cat, has, the, president] ,[]).
sentence([the, cat, has, the, cat] ,[]).
sentence([the, cat, has, a, president] ,[]).
sentence([the, cat, has, a, cat] ,[]).
sentence([a, president, has, the, president] ,[]).
sentence([a, president, has, the, cat] ,[]).
sentence([a, president, has, a, president] ,[]).
sentence([a, president, has, a, cat] ,[]).
sentence([a, cat, has, the, president] ,[]).
sentence([a, cat, has, the, cat] ,[]).
sentence([a, cat, has, a, president] ,[]).
sentence([a, cat, has, a, cat] ,[]).
```

Code snippet B.1: The legal sentences generated by the simple context free grammar.

```
sentence([the],[]).
sentence([the, the, president],[]).
   :
sentence([cat, has, the, the],[]).
sentence([cat, the, has, the],[]).
   :
sentence([the, the, the, the, the],[]).
sentence([president, the, the, the, the],[]).
   :
sentence([the, the, a, president, president],[]).
sentence([the, has, has, the, president],[]).
   :
sentence([a, president, has, cat, the],[]).
sentence([a, president, has, the, cat],[]).
```

Code snippet B.2: A sample of random sentences of maximum length 5.

B.2 Background knowledge for completing the grammar

The following code snippets provide example contents of the '**.b**' background knowledge file as used when completing the grammar theory using `CProgol5.0` and `ALECTO`.

```
% CProgol5 Settings
:- observable(sentence/2)?
:- set(i,5)?
:- set(c,5)?
:- set(verbose,2)?
:- set(posonly)?
:- set(nodes,10000)?
```

Code snippet B.3: The settings used by `CProgol5.0` for the grammar problem.

```
% ALECTO Settings
:- set(evalfn,user). % for abductive scoring

:- set(max_skiplist_number,3).
:- set(max_hlist_number,2).
:- set(max_crosslist_number,5).
:- set(max_hlist_pred_symbol,3).
:- set(max_skiplist_pred_symbol,3).

:- set(i,4).
:- set(verbose,2).
:- set(nodes,2000).
:- set(proof_strategy,sld).
```

Code snippet B.4: The settings used by `ALECTO` for the grammar problem

```
% CProgol5 specific Language Bias
:- modeh(*,verb_phrase(+listofwords,-listofwords))?
:- modeh(*,noun_phrase(+listofwords,+listofwords))?
```

Code snippet B.5: Typical `CProgol5.0` specific language bias for the grammar problem.

```
% ALECTO specific Language Bias
:- abducible(noun_phrase/2).
:- abducible(verb_phrase/2).

:- modeh(1,verb_phrase(+listofwords,-listofwords)).
:- modeh(1,noun_phrase(+listofwords,-listofwords)).

:- determination(verb_phrase/2,noun/2).
:- determination(verb_phrase/2,determiner/2).
:- determination(verb_phrase/2,verb/2).
:- determination(verb_phrase/2,noun_phrase/2).
:- determination(verb_phrase/2,cons/3).

:- determination(noun_phrase/2,noun/2).
:- determination(noun_phrase/2,determiner/2).
:- determination(noun_phrase/2,verb_phrase/2).
:- determination(noun_phrase/2,verb/2).
:- determination(noun_phrase/2,cons/3).
```

Code snippet B.6: Typical `ALECTO` specific language bias for the grammar problem.

```
:- modeb(1,verb(+listofwords,-listofwords)).
:- modeb(1,noun_phrase(+listofwords,-listofwords)).
:- modeb(1,noun(+listofwords,-listofwords)).
:- modeb(1,determiner(+listofwords,-listofwords)).
:- modeb(1,verb_phrase(+listofwords,-listofwords)).

:- modeb(*,(+listofwords)=(-listofwords)).
:- modeb(*,(+listofwords)=(+listofwords)).
:- modeb(*,(#listofwords)=(#listofwords)).
:- modeb(*,(+listofwords)=(#listofwords)).
:- modeb(*,(+word)=(#word)).
:- modeb(*,cons(+listofwords,#word,-listofwords)).
```

Code snippet B.7: Common `CProgol5.0` and `ALECTO` *body modes* for the grammar problem.

```
% Typing
cons([H|T],H,T).

listofwords([]).
listofwords([H|T]):-
    word(H), listofwords(T).

word(the). word(president). word(has). word(a). word(cat).

suffixlist(T,[_H|T]).
suffixlist(T2,[_H|T1]):-
    suffixlist(T2,T1).
```

Code snippet B.8: Common `CProgol5.0` and `ALECTO` typing information for the grammar problem.

```
% CProgol5 Acceptability constraints
'*determiner'(A,B) :- suffix(A,B).
'*noun_phrase'(A,B) :- suffix(A,B).
'*verb_phrase'(A,B) :- suffix(A,B).
'*noun'(A,B) :- suffix(A,B).
'*verb'(A,B) :- suffix(A,B).
```

Code snippet B.9: `CProgol5.0` abductive explanation acceptability constraints for the grammar problem.

```
% ALECTO Acceptability constraints
ground_lit(noun(X,Y),noun(X,Y1)):-
    Y=Y1, suffixlist(Y1,X).
ground_lit(determiner(X,Y),determiner(X,Y1)):-
    Y=Y1, suffixlist(Y1,X).
ground_lit(verb_phrase(X,Y),verb_phrase(X,Y1)):-
    Y=Y1, suffixlist(Y1,X).
ground_lit(verb(X,Y),verb(X,Y1)):-
    Y=Y1, suffixlist(Y1,X).
ground_lit(noun_phrase(X,Y),noun_phrase(X,Y1)):-
    Y=Y1, suffixlist(Y1,X).
ground_lit(noun(X,Y),noun(X,Y1)):-
    Y=Y1, suffixlist(Y1,X).
```

Code snippet B.10: ALECTO abductive explanation acceptability constraints for the grammar problem.

```
% Grammar from Van Le p479
sentence(S,R):-
    noun_phrase(S,T), verb_phrase(T,R).

noun_phrase(S,T):-
    determiner(S,N), noun(N,T).

verb_phrase(T,R):-
verb(T,N), noun_phrase(N,R).

determiner([the|R],R).   determiner([a|R],R).

noun([president|R],R).   noun([cat|R],R).

verb([has|R],R).
```

Code snippet B.11: Common background clauses for the grammar problem.

B.3 Experimental results from completing the grammar

This section presents the theories generated by the Theory Completion systems `CProgol5.0` and `ALECTO`.

B.3.1 Theories generated by `CProgol5.0` for the grammar

Generated *verb_phrase* theory.

```
verb_phrase([A|B],C) :- noun_phrase(B,C).
```

Generated *noun_phrase* theory.

```
noun_phrase([A|B],C).
```

Generated *determiner* theory.

```
determiner([A,B|C],D) :- verb(C,[E|D]).
```

Generated *noun* theory.

```
noun([A,B,C,D|E],E).
```

Generated *verb* theory.

```
verb([A|B],B).
```

Figure B.1: The theories generated using `CProgol5.0` for the Group I Grammar problems.

Generated *noun_phrase* and *verb_phrase* theory.

```
noun_phrase([A|B],C).
verb_phrase(T,R) :- verb(T,N), noun_phrase(N,R).
```

Generated *noun_phrase* and *verb* theory.

```
noun_phrase([A|B],C).
```

Generated *noun_phrase* and *determiner* theory.

```
noun_phrase([A|B],C).
```

Generated *noun_phrase* and *noun* theory.

```
noun_phrase([A|B],C).
```

Generated *verb_phrase* and *noun* theory.

```
noun([A,B,C,D|E],E).
```

Generated *verb_phrase* and *verb* theory.

```
verb_phrase([A|B],C) :- noun_phrase(B,C).
```

Generated *verb_phrase* and *determiner* theory.

```
determiner([A,B|C],D) :- verb(C,[E|D]).
```

Generated *determiner* and *noun* theory.

```
determiner([A,B|C],D) :- cons(C,has,[E,F|D]).
```

Generated *determiner* and *verb* theory.

```
determiner([A,B|C],D) :- cons(C,has,[E|D]).
```

Generated *noun* and *verb* theory.

```
noun([A,B|C],D) :- determiner(C,[E|D]).
```

Figure B.2: The theories generated using `CProgol5.0` for the Group II Grammar problems.

Generated *sentence* theory.

```
sentence([A|B],C) :- determiner([A|B],B), noun_phrase([A|B],D),
    verb_phrase(D,C).
```

Figure B.3: The theories generated using `CProgol5.0` for the Group III Grammar problems.

B.3.2 Theories generated by `ALECTO` for the grammar

Generated *verb_phrase* theory.

```
verb_phrase(A,B) :-
   verb(A,C), noun_phrase(C,B).
```

Generated *noun_phrase* theory.

```
noun_phrase(A,B) :-
   determiner(A,C), noun(C,B).
```

Generated *determiner* theory.

```
determiner(A,B) :-
   A=[C|B].
determiner(A,B) :-
   cons(A,a,B).
```

Generated *noun* theory.

```
noun(A,B) :-
   cons(A,cat,B).
noun(A,B) :-
   cons(A,president,B).
```

Generated *verb* theory.

```
verb(A,B) :-
   cons(A,has,B).
```

Figure B.4: The theories generated using `ALECTO` for the Group I Grammar problems.

Generated *noun_phrase* and *verb_phrase* theory.

```
noun_phrase(A,B) :-
   determiner(A,C), noun(C,B).
noun_phrase(A,B) :-
   determiner(A,B).
verb_phrase(A,B) :-
   noun(A,C), verb(C,D), noun_phrase(D,B).
verb_phrase(A,B) :-
   verb(A,C), noun_phrase(C,B).
```

Generated *noun_phrase* and *verb* theory.

```
noun_phrase(A,B) :-
   determiner(A,C), noun(C,B).
noun_phrase(A,B) :-
   determiner(A,B).
noun_phrase(A,B) :-
   cons(A,has,C), determiner(C,D), noun(D,B).
verb(A,B) :-
   cons(C,has,B).
```

Generated *noun_phrase* and *determiner* theory.

```
noun_phrase(A,B) :-
   cons(A,a,C), noun(C,B).
noun_phrase(A,B) :-
   cons(A,the,C), noun(C,B).
```

Generated *noun_phrase* and *noun* theory.

```
noun_phrase(A,B) :-
   determiner(A,C), cons(C,cat,B).
noun_phrase(A,B) :-
   determiner(A,C), cons(C,president,B).
```

Generated *verb_phrase* and *noun* theory.

```
noun(A,B) :-
   B=[C|D].
verb_phrase(A,B) :-
   verb(A,C), noun_phrase(C,D), cons(D,cat,B).
verb_phrase(A,B) :-
   determiner(A,C), cons(C,cat,B).
```

Figure B.5: The theories generated using ALECTO for the Group II Grammar problems (part 1 of 2).

Generated *verb_phrase* and *verb* theory.

```
verb_phrase(A,B) :-
   cons(A,has,C), noun_phrase(C,B).
```

Generated *verb_phrase* and *determiner* theory.

```
determiner(A,B) :-
   A=[C|B].
verb_phrase(A,B) :-
   verb(A,C), noun_phrase(C,B).
```

Generated *determiner* and *noun* theory.

```
determiner(A,B) :-
   A=[C|B].
noun(A,B) :-
   determiner(A,B).
```

Generated *determiner* and *verb* theory.

```
determiner(A,B) :-
   A=[C|B].
verb(A,B) :-
   cons(A,has,B).
```

Generated *noun* and *verb* theory.

```
noun(A,B) :-
   A=[C|B].
verb(A,B) :-
   cons(A,has,B).
```

Figure B.6: The theories generated using ALECTO for the Group II Grammar problems (part 2 of 2).

Generated *sentence* theory.

```
sentence(A,B) :- noun_phrase(A,C), verb_phrase(C,B).
```

Figure B.7: The theories generated using ALECTO for the Group III Grammar problems.

Appendix C

Appendix to Chapter 6

This appendix provides code listings and supporting material to chapter 6.

C.1 Observations available for completing the grammar

824 observations of attacks from one intrusion strategy were produced. Code snippet C.1 presents 10 such examples.

10,000 mutated attacks strings from the positive examples were produced. Code snippet C.2 presents 2 such random examples.

C.2 Background knowledge

Sample background code for the grammar consumer clauses is shown in code snippet C.3.

Sample background code for the dis-assembler is shown in code snippet C.4.

C.3 Mode Declarations

Typical mode declarations and settings for the recovery of the clause `consume_load/4` are shown in code snippet C.5.

```
intrusion1([249,252,144,153,249,245,155,155,235,26,95,137,127,10,102,
            49,192,152,136,71,9,137,71,14,176,11,137,251,141,79,10,141,
            87,14,205,128,232,225,255,255,255,/,b,i,n,/,t,c,s,h],[]).
intrusion1([249,248,248,249,248,155,235,32,91,137,91,9,186,255,255,255,
            255,66,136,83,8,137,83,13,184,255,255,255,255,64,176,11,141,
            75,9,141,83,13,205,128,232,219,255,255,255,/,b,i,n,/,b,s,h],[]).
intrusion1([245,245,252,144,144,144,144,155,248,235,26,95,137,127,9,102,
            49,192,152,136,71,8,137,71,13,176,11,137,251,141,79,9,141,87,
            13,205,128,232,225,255,255,255,/,b,i,n,/,b,s,h],[]).
intrusion1([144,155,153,252,245,153,155,235,24,91,137,91,9,49,201,136,
            75,8,137,75,13,49,192,4,11,141,75,9,141,83,13,205,128,232,
            227,255,255,255,/,b,i,n,/,b,s,h],[]).
intrusion1([153,245,144,249,155,245,144,155,155,235,24,90,137,82,8,49,
            192,136,66,7,137,66,12,4,11,137,211,141,74,8,141,82,12,205,
            128,232,227,255,255,255,/,b,i,n,/,s,h],[]).
intrusion1([245,245,155,144,153,245,155,235,24,95,137,127,10,49,192,136,
            71,9,137,71,14,176,11,137,251,141,79,10,141,87,14,205,128,
            232,227,255,255,255,/,b,i,n,/,b,a,s,h],[]).
intrusion1([155,252,235,28,93,137,109,10,49,210,136,85,9,137,85,14,102,
            49,192,152,4,11,137,235,141,77,10,141,85,14,205,128,232,223,
            255,255,255,/,b,i,n,/,b,a,s,h],[]).
intrusion1([252,252,245,153,155,235,26,94,137,118,8,102,49,192,152,136,
            70,7,137,70,12,176,11,137,243,141,78,8,141,86,12,205,128,232,
            225,255,255,255,/,b,i,n,/,s,h],[]).
intrusion1([144,249,245,245,144,245,248,249,235,26,91,137,91,8,184,255,
            255,255,255,64,136,67,7,137,67,12,4,11,141,75,8,141,83,12,
            205,128,232,225,255,255,255,/,b,i,n,/,s,h],[]).
intrusion1([245,155,249,235,26,93,137,109,10,102,49,192,152,136,69,9,
            137,69,14,176,11,137,235,141,77,10,141,85,14,205,128,232,
            225,255,255,255,/,b,i,n,/,t,c,s,h],[]).
```

Code snippet C.1: Ten *examples* of malicious byte streams that conform to the intrusion attack strategy in code snippet 6.1.

```
intrusion1([144,248,252,235,24,94,136,118,8,49,192,136,70,7,137,70,12,
            176,11,137,243,141,78,8,141,86,12,205,128,232,227,255,255,
            255,47,98,105,110,47,115,104],[]).  % 1 bits flipped
intrusion1([252,245,249,252,252,155,245,245,235,34,95,137,127,10,187,
            255,255,255,255,67,136,95,9,137,95,14,184,255,255,255,255,
            64,5,11,137,251,141,79,10,141,87,14,205,128,232,217,255,
            255,255,47,98,105,110,47,98,97,115,104],[]).  % 1 bits flipped
```

Code snippet C.2: Two *random examples*.

```
%  consumers
consume_idles([B|OutList],[B|OutList]):-
    not(idle(B,_,_)).
consume_idles([Idle,B|OutList],[B|OutList]):-
    idle(Idle,_,_),
    not(idle(B,_,_)).
consume_idles([Idle|Outlist],Outlist2):-
    idle(Idle,_,_),
    consume_idles(Outlist,Outlist2).

consume_load([Jump,Offset1,Pop|Rest],Rest3,Reg,Len) :-
    jump(Jump,_), pop(Pop,Reg,_),
    remove_load_sequence(Offset1,[Pop|Rest],Rest2),
    detect_command(Rest,Command),
    append(Rest3,Command,Rest2), length(Command,Len).

consume_null_long([A,B|InList],OutList,Null,Addr):-
    null_long([A,B|NullLongList],Null),
    append([A,B|NullLongList],OutList,[A,B|InList]),
    valid_null_address(Null,Addr).

consume_null_byte([B1,B2|Rest],Rest,Null,Addr) :-
    null_byte(B1,B2,Null), decode(B2,Null),
    lessthan(Null,4), not(equals(Addr,Null)).

consume_place_addr([B1,B2,L1|Rest],Rest,RegSrc,L2) :-
    safeplus(L2,1,L1), movl(B1,_), bitop(B2,RegSrc,RegSrc,0x40).

consume_terminate([B1,B2,L|Rest],Rest,L,RegSrc,RegDest) :-
    movb(B1,_), bitop(B2,RegSrc,RegDest,0x40).

consume_place_null([B1,B2,L1|Rest],Rest,L2,RegSrc,RegDest) :-
    safeplus(L2,5,L1), movl(B1,_), bitop(B2,RegSrc,RegDest,0x40).

consume_null_operation(A,C,Len,Null,Addr) :-
    consume_terminate(A,B,Len,Null,Addr),
    consume_place_null(B,C,Len,Null,Addr).

consume_set_sys_call([0xb0,Call_Number|Rest],Rest,0,Call_Number,_).
consume_set_sys_call([0x04,Call_Number|Rest],Rest,0,Call_Number,_).
consume_set_sys_call(A,B,_,CN,Addr) :-
    consume_null_long(A,C,0,Addr), consume_set_sys_call(C,B,0,CN,Addr).

consume_set_sys_call([0xb0,Call_Number|Rest],Rest,0,Call_Number,_,0).
consume_set_sys_call([0x04,Call_Number|Rest],Rest,0,Call_Number,_,1).

consume_move_addr([B1,B2|Rest],Rest,Reg1,Reg2) :-
    movl(B1,_), bitop(B2,Reg1,Reg2,0xc0).
consume_move_addr(Rest,Rest,Reg1,Reg1).

consume_param(Rest,Rest,_,_,2'010,2'010).
consume_param([B1,B2,L1|Rest],Rest,L1,RegSrc,_Null,RegDest) :-
    leal(B1,_), bitop(B2,RegDest,RegSrc,0x40),!.

consume_params(A,D,Addr,Null,Len) :-
    calculate_offsets(Len,Offset1,Offset2),
    consume_move_addr(A,B,Addr),
    consume_param(B,C,Offset1,Addr,Null,2'001),
    consume_param(C,D,Offset2,Addr,Null,2'010).

consume_system_call([B1,B2|Rest],Rest) :-
    system_call(B1,B2,_).
```

Code snippet C.3: Sample grammar code from the consumer clauses.

```
% Typing predicates
byte(B):-integer(B).
:
listofbytes([]).
listofbytes([H|T]):-  byte(H),listofbytes(T).
mnemonic('nop').  mnemonic('cld').  ... mnemonic('leal').
null(R):- register(R).
nulllong(0). nulllong(1). nulllong(2). nulllong(3).
offset(1). offset(5).  register(2'001). register(2'010).
%  utility predicates
:
remove_load_sequence(Offset1,[Pop|Rest],Rest2):-
    pop(Pop,_Register,_Nmenonic),
    nth0(Offset1,[Pop|Rest],Call),
    call_(Call,_),
    safeplus(Offset1,1,Tmp1),
    :
    safeplus(Offset1,4,Tmp4),
    nth0(Tmp4,[Pop|Rest],0xFF),%,
    safeplus(Offset1,-1,Tmp5),
    remove_seq(Tmp5,5,Rest,Rest2).
detect_command(ListOfBytes,Command):-
    command(Command,_ID),
    sub_list(Command,ListOfBytes).
:
null_long([B1,B2],Null):-
    null_long(B1,B2,Null),
    decode(B2,Null).
:
decode(B,R1) :-
    nonvar(B), R1 is (B /\ 2'00111000) >> 3,
    R2 is B /\ 2'00000111, R1 =:= R2,!.
:
msbset_register_byte(Byte,RegSrc,RegDest):-
    bitop(Byte,RegSrc,RegDest,0x40).
bitop(A,B,C,D) :-
    nonvar(B), nonvar(C), nonvar(D), A is (B << 3) \/ C \/ D.
:
safeplus(A,B,C) :- nonvar(A), nonvar(B), nonvar(C), C is A + B.%, !.
:
%  idle generators
idle(0x90,'nop',0). idle(0xfc,'cld',1). idle(0xf9,'stc',2).
:
%  pop generators
pop(0x58,2'000,'popl %eax'). pop(0x59,2'001,'popl %ecx').
:
%  other instruction generators
jump(0xeb,'jmp').  call_(0xe8,'call').  cwtl(0x98,'cwtl').
cbtw(0x66,0x98,'cbtw').  system_call(0xcd,0x80,'int $0x80').
movl(0x89,'movl'). movb(0x88,'movb'). leal(0x8d,'leal').
%  null long generators
null_long(0x31,_,_). % xorl
:
null_long(0xb8,0xff,0xff,0xff,0xff,0x40,0). % movl f's into eax then inc
:
%  null word generators
null_word(0x66,0x31). null_word(0x66,0x29).
:
%  null byte generators
null_byte(0x30,_,_). null_byte(0x28,_,_).
%  command generators
command(['/','b','i','n','/','s','h'],0).
command(['/','b','i','n','/','b','a','s','h'],1).
:
```

Code snippet C.4: Sample code represent a dis-assembler for Intel x86 processors.

```
:- set(abinduce_init_from_file,yes).
:- set(evalfn,user). % for abductive scoring

:- set(max_skiplist_number,1).
:- set(max_hlist_number,1).
:- set(max_crosslist_number,1).
:- set(max_hlist_pred_symbol,3).
:- set(max_skiplist_pred_symbol,3).

:- set(i,2).
:- set(verbose,2).
:- set(nodes,2000).
:- set(proof_strategy,sld).
:- set(record,true).
:- set(recordfile,'consume_load.out').
:- set(clauselength,20).
:- set(depth,10000).  % for very long list manipulatons

:- abducible(consume_load/4).

ground_lit(consume_load(A,B,C,D),consume_load(A,B,C,D)):-
    addr(C), len(D),
    sublist(A,B).


:- modeh(1,consume_load([+jmpbyte,+offsetbyte,+popbyte|+listofbytes],
                        -listofbytes,+popaddr,+length)).

:- modeb(1,jump(+jmpbyte,-mnemonic)).
:- modeb(1, pop(+popbyte,-popaddr,-mnemonic)).
:- modeb(1, remove_load_sequence(+offsetbyte,[+popbyte|+listofbytes],-listofbytes)).
:- modeb(1, detect_command(+listofbytes,-listofbytes)).
:- modeb(1, append(-listofbytes,+listofbytes,+listofbytes)).
:- modeb(1, length(+listofbytes,-length)).
```

Code snippet C.5: Mode declarations and other settings for recovering `consume_load/4`.

C.4 Experimental Results

C.4.1 Learning Curves

This section presents the results of the experiments to regenerate the IDS clauses. For each of the steps in the number of training examples, there were 10 folds. The following notes pertain to the tables that follow.

Notes:

1. $|E^+|$ is the number of training examples.
2. *Average Accuracy* is the average accuracy of the clauses produced (with respect to the test folds).
3. *Accuracy Std. Dev.* is the standard deviation of the accuracy.
4. *Abduction CPU time (seconds)* is the average extraction case abduction time.
5. *Induction CPU time (seconds)* is the average generalisation time.
6. *Ave. Total CPU time (seconds)* is the average of the total run times.
7. *Total CPU time Std. Dev. (seconds)* is the standard deviation of the total run times.
8. The machine used for the experiments was a dual processor Intel Pentium III, 550MHz, 512 MB RAM.

Table C.1: Learning curve information for generating `consume_load/4` using `ALECTO`.

$\|E^+\|$	Average Accuracy	Accuracy Std. Dev.	Abduction CPU time (seconds)	Induction CPU time (seconds)	Ave. Total CPU time (seconds)	Total CPU time Std. Dev. (seconds)
2	0%	0%	0.03	1.19	1.22	0.04
5	0%	0%	0.09	7.22	7.31	1.4
10	0%	0%	0.15	32.8	33.0	1.9
20	0%	0%	0.32	188.5	188.8	51.7
50	0%	0%	0.82	209.4	210.3	0.04
100	0%	0%	1.81	209.4	211.2	0.04

Table C.2: Learning curve information for generating `consume_place_addr/4` using `ALECTO`.

$\|E^+\|$	Average Accuracy	Accuracy Std. Dev.	Abduction CPU time (seconds)	Induction CPU time (seconds)	Ave. Total CPU time (seconds)	Total CPU time Std. Dev. (seconds)
2	0%	0%	0.23	0.69	0.92	0.12
5	0%	0%	0.56	1.26	1.81	0.12
10	0%	0%	1.19	2.65	3.84	0.15
20	0%	0%	2.36	6.94	9.30	0.41
50	0%	0%	5.98	28.8	34.8	2.21
100	0%	0%	12.0	93.8	105.9	7.18

Table C.3: Learning curve information for generating `consume_null_long/4` using `ALECTO`.

$\|E^+\|$	Average Accuracy	Accuracy Std. Dev.	Abduction CPU time (seconds)	Induction CPU time (seconds)	Ave. Total CPU time (seconds)	Total CPU time Std. Dev. (seconds)
2	100%	0%	0.05	0.95	0.99	0.17
5	100%	0%	0.12	3.08	3.19	0.69
10	100%	0%	0.23	10.4	10.6	3.90
20	100%	0%	0.47	28.0	28.5	6.78
50	100%	0%	1.22	312.3	313.5	132.9
100	100%	0%	2.56	645.1	647.7	477.0

Table C.4: Learning curve information for generating `consume_null_operation/5` using `ALECTO`.

$\|E^+\|$	Average Accuracy	Accuracy Std. Dev.	Abduction CPU time (seconds)	Induction CPU time (seconds)	Ave. Total CPU time (seconds)	Total CPU time Std. Dev. (seconds)
2	89.8%	20.5%	0.04	1.86	1.89	0.43
5	90%	30%	0.11	5.09	5.20	1.42
10	100%	0%	0.22	10.2	10.4	2.57
20	100%	0%	0.42	34.8	35.2	11.9
50	100%	0%	1.10	113.3	114.4	26.4
100	70%	45.8%	2.31	340.0	342.3	20.4

Table C.5: Learning curve information for generating `consume_params/5` using `ALECTO`.

$\|E^+\|$	Average Accuracy	Accuracy Std. Dev.	Abduction CPU time (seconds)	Induction CPU time (seconds)	Ave. Total CPU time (seconds)	Total CPU time Std. Dev. (seconds)
2	4.08 %	8.2 %	0.04	0.67	0.71	0.46
5	22.5 %	6.1 %	0.09	2.4	2.5	1.06
10	30.8 %	10.0 %	0.18	5.5	5.7	2.98
20	37.0 %	8.3 %	0.37	25.2	25.6	4.90
50	33.0 %	9.4 %	0.95	146.2	147.1	17.0
100	30.9 %	9.7 %	1.92	558.2	560.1	343.1

Table C.6: Learning curve information for generating `consume_place_null/5` using `ALECTO`.

$\|E^+\|$	Average Accuracy	Accuracy Std. Dev.	Abduction CPU time (seconds)	Induction CPU time (seconds)	Ave. Total CPU time (seconds)	Total CPU time Std. Dev. (seconds)
2	0%	0%	0.16	1.35	1.51	0.74
5	0%	0%	0.40	2.56	2.95	2.30
10	0%	0%	0.77	9.80	10.6	3.04
20	0%	0%	1.49	26.4	27.9	1.71
50	0%	0%	3.78	125.8	129.6	16.2
100	0%	0%	7.89	410.1	418.0	102.6

Table C.7: Learning curve information for generating `consume_place_addr/4` and `consume_null_long/4` using `ALECTO`.

$\|E^+\|$	Average Accuracy	Accuracy Std. Dev.	Abduction CPU time (seconds)	Induction CPU time (seconds)	Ave. Total CPU time (seconds)	Total CPU time Std. Dev. (seconds)
2	0%	0%	0.05	2.29	2.34	0.40
5	0%	0%	0.13	5.03	5.16	0.63
10	0%	0%	0.26	19.8	20.1	10.7
20	0%	0%	0.54	57.8	58.4	28.3
50	0%	0%	1.40	157.1	158.5	103.1
100	0%	0%	3.01	238.5	241.5	0.08

C.4.2 Sample Theories

```
% no theory produced
```

Code snippet C.6: A sample recovered theory recovered clause for `consume_load/4`.

```
consume_place_addr([A, B, C|D], D, E, F) :-
   safeplus(F, 1, C), movl(A, G).
```

Code snippet C.7: A sample recovered theory recovered clause for `consume_place_addr/4`.

```
consume_null_long([A, B|C], D, E, F) :-
   null_long([A, B|G], E), append([A, A|G], D, [A, A|C]).
```

Code snippet C.8: A sample recovered theory recovered clause for `consume_null_long/4`.

```
consume_null_operation(A, B, C, D, E) :-
   consume_terminate(A, F, C, D, E), consume_place_null(F, B, C, D, E).
```

Code snippet C.9: A sample recovered theory recovered clause for `consume_null_operation/5`.

```
consume_params(A, B, C, D, E) :-
   register(F), register(G), calculate_offsets(E, H, I),
   consume_param(A, J, H, C, D, G), consume_param(J, B, I, C, D, F).

consume_params(A, B, C, D, E) :-
   register(F), calculate_offsets(E, G, H),
   consume_move_addr(A, I, C), consume_param(I, J, G, C, D, F),
   consume_param(J, B, H, C, D, C).
```

Code snippet C.10: A sample recovered theory recovered clauses for `consume_params/5`.

```
consume_place_null([A, B, C|D], D, E, F, G) :-
   safeplus(E, 5, C).
```

Code snippet C.11: A sample recovered theory recovered clause for `consume_place_null/5`.

```
consume_place_addr([A, B, C|D], D, E, F) :-
   safeplus(F, 1, C), movl(A, G).
```

Code snippet C.12: A sample recovered theory for `consume_place_addr/4` and `consume_null_long/4`.

Appendix D

Appendix to Chapter 7

This appendix provides code listings and supporting material to chapter 7. It is split into three sections: 1) material common to both the Blocks World and Robot Navigation Control problem; 2) material solely for the Blocks World problem; and 3) material solely for the Robot Navigation Control problem.

D.1 Supporting material, common to both Event Calculus problems

```
%%% Event Calculus Core Axioms

holds_at(F,T2):-
    happens(A,T1), lt(T1,T2), initiates(A,F,T1), not clipped(T1,F,T2).

holds_at(F,T):-
    initially(F), not clipped(0,F,T).

clipped(T1,F,T2):-
    happens(A,T), le(T1,T), lt(T,T2), terminates(A,F,T).

lt(T1,T2):-
    time_point(T1), time_point(T2), T1 < T2.
le(T1,T2):-
    time_point(T1), time_point(T2), T1 =< T2.
```

Code snippet D.1: The Event Calculus core axioms.

```
:- abducible(initiates/3).
:- abducible(terminates/3).
```

Code snippet D.2: The specification of abducible predicates for the Event Calculus in `ALECTO`.

```
ground_lit(initiates(X,Y,Z),initiates(X,Y,Z)):-
    ground_initiating_transition(Y,Z).
ground_lit(terminates(X,Y,Z),terminates(X,Y,Z)):-
    ground_terminating_transition(Y,Z).

ground_initiating_transition(Y,Z):-
    not_holds_at(Y,Z),
    Z1 is Z +1,
    example_holds_at(Y,Z1).

ground_terminating_transition(Y,Z):-
    example_holds_at(Y,Z),
    Z1 is Z +1,
    not_holds_at(Y,Z1).

example_holds_at(A,B):-
    example(_N,pos,holds_at(A,B)).
```

Code snippet D.3: The specification of acceptability constraints for the Event Calculus abducibles in ALECTO.

```
% constraints.pl

false:-   % Constraint 1
    hypothesis(_Head,Body,_),
    Body \== true,
    number_of_literals(Body,L),
    L==1,
    write('[Clause not long enough -- only one literal in body!]'),nl,
    set(constraint_violated,true),!.

false:-   % Constraint 2
    hypothesis(Head,Body,_),
    clause_not_generative((Head:-Body)),
    write('[Clause not strongly generative -- un-necessary variables!]'),nl,
    set(constraint_violated,true),!.

false:-   % Constraint 3
    hypothesis(terminates(_Event,Fluent,Time),Body,_),
    Body \== true,
    not in(Body,holds_at(Fluent,Time)),
    write('[Terminates clause does not refer to the fluent being terminated!]'),nl,
    set(constraint_violated,true),!.

false:-   % Constraint 4
    hypothesis(initiates(_Event,HeadFluent,Time),Body,_),
    in(Body,holds_at(BodyFluent,Time)),
    HeadFluent == BodyFluent,
    write('[Initiates clause refers to the fluent already holding!]'),nl,
    set(constraint_violated,true),!.

false:-   % Constraint 5
    hypothesis(initiates(_Event,Fluent,Time),Body,_),
    Body \== true,
    functor(Fluent,FSymb,Arity),
    functor(BodyFluent,FSymb,Arity),
    not in(Body,holds_at(BodyFluent,Time)),
    write('[Initiates clause does not refer to the fluent symbol in the body!]'),nl,
    set(constraint_violated,true),!.
```

Code snippet D.4: The specification of hypothesis structure constraints for the Event Calculus abducibles in ALECTO.

D.2 Supporting material for learning Domain Specific Axioms to describe the Blocks World

```
:- modeh(1,terminates(move(-object,-object),clear(-object),+time_point)).
:- modeh(1,initiates(move(-object,-object),clear(-object),+time_point)).
:- modeh(1,terminates(move(-object,-object),on(-object,-object),+time_point)).
:- modeh(1,initiates(move(-object,-object),on(-object,-object),+time_point)).

:- modeb(1,+event = move(-object,-object)).
:- modeb(1,+fluent = clear(-object)).
:- modeb(1,+fluent = on(-object,-object)).
:- modeb(100,diff(+object,+object)).
:- modeb(100,holds_at(clear(-object),+time_point)).
:- modeb(100,holds_at(on(-object,-object),+time_point)).

:- determination(initiates/3,'='/2).
:- determination(initiates/3,diff/2).
:- determination(initiates/3,holds_at/2).
:- determination(initiates/3,not_holds_at/2).
:- determination(terminates/3,'='/2).
:- determination(terminates/3,diff/2).
:- determination(terminates/3,holds_at/2).
:- determination(terminates/3,not_holds_at/2).

:- commutative(diff/2).
```

Code snippet D.5: The mode declarations used for learning domain specific axioms to describe the Blocks World.

```
% Initial situation

initially(on(block_c,table)).
initially(on(block_b,block_c)).
initially(on(block_a,block_b)).
initially(on(block_d,table)).

initially(clear(block_a)).
initially(clear(block_d)).
initially(clear(table)).
```

```
% Narrative

happens(move(block_a,block_d),5).
happens(move(block_b,block_a),10).
```

Code snippet D.6: Initial situation and narrative of events for the Blocks World.

```
time_point(0). time_point(1). time_point(2). time_point(3).
time_point(4). time_point(5). time_point(6). time_point(7).
time_point(8). time_point(9). time_point(10). time_point(11).
time_point(12). time_point(13). time_point(14). time_point(15).

object(block_a).          event(move(block_a,block_d)).
object(block_b).          event(move(block_b,block_a)).
object(block_c).
object(block_d).
object(table).

fluent(on(X,Y)):-object(X),object(Y).
fluent(clear(X)):-object(X).
```

Code snippet D.7: Other Background knowledge for the Blocks World.

```
not_holds_at(on(block_a,block_a),0).        not_holds_at(on(block_a,block_a),15).
not_holds_at(on(block_a,block_c),0).        not_holds_at(on(block_a,block_b),15).
not_holds_at(on(block_a,block_d),0).        not_holds_at(on(block_a,block_c),15).
not_holds_at(on(block_a,table),0).          not_holds_at(on(block_a,table),15).
not_holds_at(on(block_b,block_a),0).        not_holds_at(on(block_b,block_b),15).
   :                                           :
not_holds_at(on(table,block_d),0).          not_holds_at(on(table,block_d),15).
not_holds_at(on(table,table),0).            not_holds_at(on(table,table),15).
not_holds_at(clear(block_b),0).             not_holds_at(clear(block_a),15).
not_holds_at(clear(block_c),0).             not_holds_at(clear(block_d),15).
```

Code snippet D.8: Example negative fluent trace data for the Blocks World.

```
% Fluents that hold for time point 0
holds_at(on(block_a,block_b),0).
holds_at(on(block_b,block_c),0).
holds_at(on(block_c,table),0).
holds_at(on(block_d,table),0).
holds_at(clear(block_a),0).
holds_at(clear(block_d),0).
holds_at(clear(table),0).

% Fluents that hold for time point 1
holds_at(on(block_a,block_b),1).
holds_at(on(block_b,block_c),1).
holds_at(on(block_c,table),1).
holds_at(on(block_d,table),1).
holds_at(clear(block_a),1).
holds_at(clear(block_d),1).
holds_at(clear(table),1).

% Fluents that hold for time point 2
holds_at(on(block_a,block_b),2).
holds_at(on(block_b,block_c),2).
holds_at(on(block_c,table),2).
holds_at(on(block_d,table),2).
holds_at(clear(block_a),2).
holds_at(clear(block_d),2).
holds_at(clear(table),2).

% Fluents that hold for time point 3
holds_at(on(block_a,block_b),3).
holds_at(on(block_b,block_c),3).
holds_at(on(block_c,table),3).
holds_at(on(block_d,table),3).
holds_at(clear(block_a),3).
holds_at(clear(block_d),3).
holds_at(clear(table),3).

% Fluents that hold for time point 4
holds_at(on(block_a,block_b),4).
holds_at(on(block_b,block_c),4).
holds_at(on(block_c,table),4).
holds_at(on(block_d,table),4).
holds_at(clear(block_a),4).
holds_at(clear(block_d),4).
holds_at(clear(table),4).

% Fluents that hold for time point 5
holds_at(on(block_a,block_b),5).
holds_at(on(block_b,block_c),5).
holds_at(on(block_c,table),5).
holds_at(on(block_d,table),5).
holds_at(clear(block_a),5).
holds_at(clear(block_d),5).
holds_at(clear(table),5).

% Fluents that hold for time point 6
holds_at(on(block_a,block_d),6).
holds_at(on(block_b,block_c),6).
holds_at(on(block_c,table),6).
holds_at(on(block_d,table),6).
holds_at(clear(block_a),6).
holds_at(clear(block_b),6).
holds_at(clear(table),6).

% Fluents that hold for time point 7
holds_at(on(block_a,block_d),7).
holds_at(on(block_b,block_c),7).
holds_at(on(block_c,table),7).
holds_at(on(block_d,table),7).
holds_at(clear(block_a),7).
holds_at(clear(block_b),7).
holds_at(clear(table),7).

% Fluents that hold for time point 8
holds_at(on(block_a,block_d),8).
holds_at(on(block_b,block_c),8).
holds_at(on(block_c,table),8).
holds_at(on(block_d,table),8).
holds_at(clear(block_a),8).
holds_at(clear(block_b),8).
holds_at(clear(table),8).

% Fluents that hold for time point 9
holds_at(on(block_a,block_d),9).
holds_at(on(block_b,block_c),9).
holds_at(on(block_c,table),9).
holds_at(on(block_d,table),9).
holds_at(clear(block_a),9).
holds_at(clear(block_b),9).
holds_at(clear(table),9).

% Fluents that hold for time point 10
holds_at(on(block_a,block_d),10).
holds_at(on(block_b,block_c),10).
holds_at(on(block_c,table),10).
holds_at(on(block_d,table),10).
holds_at(clear(block_a),10).
holds_at(clear(block_b),10).
holds_at(clear(table),10).

% Fluents that hold for time point 11
holds_at(on(block_a,block_d),11).
holds_at(on(block_b,block_a),11).
holds_at(on(block_c,table),11).
holds_at(on(block_d,table),11).
holds_at(clear(block_b),11).
holds_at(clear(block_c),11).
holds_at(clear(table),11).

% Fluents that hold for time point 12
holds_at(on(block_a,block_d),12).
holds_at(on(block_b,block_a),12).
holds_at(on(block_c,table),12).
holds_at(on(block_d,table),12).
holds_at(clear(block_b),12).
holds_at(clear(block_c),12).
holds_at(clear(table),12).

% Fluents that hold for time point 13
holds_at(on(block_a,block_d),13).
holds_at(on(block_b,block_a),13).
holds_at(on(block_c,table),13).
holds_at(on(block_d,table),13).
holds_at(clear(block_b),13).
holds_at(clear(block_c),13).
holds_at(clear(table),13).

% Fluents that hold for time point 14
holds_at(on(block_a,block_d),14).
holds_at(on(block_b,block_a),14).
holds_at(on(block_c,table),14).
holds_at(on(block_d,table),14).
holds_at(clear(block_b),14).
holds_at(clear(block_c),14).
holds_at(clear(table),14).

% Fluents that hold for time point 15
holds_at(on(block_a,block_d),15).
holds_at(on(block_b,block_a),15).
holds_at(on(block_c,table),15).
holds_at(on(block_d,table),15).
holds_at(clear(block_b),15).
holds_at(clear(block_c),15).
holds_at(clear(table),15).
```

Code snippet D.9: Observations of dynamic behaviour for the Blocks World: Positive fluent traces

```
%  Theory learned - Run 5

initiates(move(A,B),on(A,B),C) :-
     holds_at(clear(A),C), holds_at(clear(B),C).
terminates(move(A,B),on(A,C),D) :-
     diff(B,block_b), holds_at(on(A,C),D), holds_at(clear(B),D).

initiates(move(A,B),clear(C),D) :-
     diff(A,B), holds_at(on(A,C),D), holds_at(clear(B),D).
terminates(move(A,B),clear(B),C) :-
     holds_at(clear(A),C), holds_at(clear(B),C).
```

Code snippet D.10: Domain specific axioms for describing the Blocks World induced using `CProgol5.0`.

Note: These clauses have been translated from the predicate `flips/4` in the original work ([Moyle, 1999] page 68) into the predicates `initiates/3` and `terminates/3`.

```
initiates(move(A,B),on(A,B),C) :-
     holds_at(clear(A),C), holds_at(clear(B),C).
terminates(move(A,B),on(A,C),D) :-
     holds_at(clear(A),D), holds_at(clear(C),D).

initiates(move(B,C),clear(B),D) :-
     holds_at(on(B,A),D), holds_at(clear(B),D).
terminates(move(B,A),clear(A),C) :-
     holds_at(clear(A),C), holds_at(clear(B),C).
```

Code snippet D.11: Domain specific axioms for describing the Blocks World induced by Wellner.

Note: These clauses have been translated from the predicate `flips/4` in the original work ([Wellner, 1999] page 116) into the predicates `initiates/3` and `terminates/3`.

D.3 Supporting material for learning Domain Specific Axioms for a Robot Navigation Control program

```
score_clause(circ_min_holds_at,Clause,Cost):-
    !,
    setting(use_crosslist,CrossKey),
    atoms_left_coverage_size(CrossKey,CovAtomsSize,_UncovAtomsSize),
    get_actual_unchanging_fluents(AUFTs,LenAUFTs),
    get_predicted_unchanging_fluents(PUFTs,LenPUFTs),
    intersect1(AUFTs,PUFTs,Intersect,_AUFTsSetDiffPUFTs),
    length(Intersect,LIntersect),
    size_of_potential_clause(Clause,ClauseSize),
    Utility is
        (LenPUFTs - LIntersect) % False Pos
        - (LenAUFTs - LIntersect) % False Neg
        + LIntersect  % True Pos
        - ClauseSize + CovAtomsSize, % Other stuff
    Cost is -1.0 * Utility,
    write('[CIRC Cost C= '),write(Cost),write(']'),
    !.
```

Code snippet D.12: The specification of a scoring function for the Event Calculus abducibles in ALECTO.

```
% Initial situation
initially(loc(corner(c7),ahead)).
initially(in(r2)).
initially(facing(north)).

% Narrative
happens(turn(right),4).          happens(go_straight,31).
happens(follow_wall,5).          happens(follow_wall,33).
happens(turn(right),9).          happens(turn(right),39).
happens(follow_wall,11).         happens(follow_wall,43).
happens(go_straight,13).         happens(turn(right),49).
happens(follow_wall,17).         happens(follow_wall,53).
happens(turn(right),19).         happens(turn(left), 55).
happens(follow_wall,23).         happens(turn(left), 57).
happens(turn(left), 25).         happens(follow_wall,59).
happens(turn(left), 27).         happens(turn(right),62).
happens(follow_wall,29).         happens(follow_wall,65).
```

Code snippet D.13: Initial situation and narrative of events for the Robot Navigation Control program.

```
:- modeh(1,initiates(#event,loc(corner(-corner),#position),+time_point)).
:- modeh(1,initiates(turn(#turn_direction),loc(door(-door),#position),+time_point)).
:- modeh(1,initiates(turn(#turn_direction),loc(corner(-corner),#position),+time_point)).
:- modeh(1,initiates(turn(#turn_direction),in(-room),+time_point)).
:- modeh(1,initiates(turn(+turn_direction),facing(+direction),+time_point)).

:- modeh(1,terminates(#event,loc(corner(-corner),#position),+time_point)).
:- modeh(1,terminates(turn(#turn_direction),loc(corner(-corner),#position),+time_point)).
:- modeh(1,terminates(turn(#turn_direction),loc(door(-door),#position),+time_point)).
:- modeh(1,terminates(turn(#turn_direction),in(-room),+time_point)).
:- modeh(1,terminates(turn(+turn_direction),facing(-direction),+time_point)).

:- modeb(100,holds_at(loc(corner(-corner),#position),+time_point)).
:- modeb(100,holds_at(door_open(-door),+time_point)).
:- modeb(100,holds_at(loc(door(-door),#position),+time_point)).
:- modeb(100,holds_at(in(-room),+time_point)).
:- modeb(100,holds_at(facing(-direction),+time_point)).

:- modeb(100,change_direction(+turn_direction,+direction,-direction)).
:- modeb(100,change_direction(+turn_direction,-direction,+direction)).
:- modeb(100,connects(+door,-room,-room)).
:- modeb(100,door(+door,-corner,-corner)).
:- modeb(100,next_visible_corner(+corner,-corner,+time_point)).
:- modeb(100,next_corner(+room,+corner,-corner)).
:- modeb(100,inner(+corner)).
:- modeb(100,corner(+corner)).
:- modeb(100,door(+door)).
:- modeb(100,room(+room)).
:- modeb(100,direction(+direction)).

:-determination(initiates/3,holds_at/2).
:-determination(initiates/3,change_direction/3).
:-determination(initiates/3,next_visible_corner/3).
:-determination(initiates/3,door/3).
:-determination(initiates/3,connects/3).
:-determination(initiates/3,next_corner/3).
:-determination(initiates/3,inner/1).
:-determination(initiates/3,corner/1).
:-determination(initiates/3,door/1).
:-determination(initiates/3,short_wall_direction/3).
:-determination(initiates/3,next_room_direction/3).
:-determination(initiates/3,direction/1).

:-determination(terminates/3,holds_at/2).
:-determination(terminates/3,door/3).
:-determination(terminates/3,connects/3).
:-determination(terminates/3,next_visible_corner/3).
:-determination(terminates/3,inner/1).
:-determination(terminates/3,corner/1).
:-determination(terminates/3,door/1).
:-determination(terminates/3,change_direction/3).
:-determination(terminates/3,short_wall_direction/3).
:-determination(terminates/3,next_room_direction/3).
:-determination(terminates/3,direction/1).

:- commutative(connects/3).
```

Code snippet D.14: Mode declarations for the Robot Navigation Control program.

```
not_holds_at(loc(corner(c9),behind),0).
not_holds_at(loc(corner(c9),ahead),0).
not_holds_at(loc(corner(c8),behind),0).
not_holds_at(loc(corner(c8),ahead),0).
not_holds_at(loc(corner(c7),behind),0).
   :
not_holds_at(in(r6),0).
not_holds_at(in(r5),0).
not_holds_at(in(r4),0).
not_holds_at(in(r3),0).
not_holds_at(in(r1),0).
   :
```

```
not_holds_at(loc(corner(c9),behind),54).
not_holds_at(loc(corner(c9),ahead),54).
not_holds_at(loc(corner(c8),behind),54).
not_holds_at(loc(corner(c8),ahead),54).
not_holds_at(loc(corner(c7),behind),54).
   :
not_holds_at(in(r6),54).
not_holds_at(in(r5),54).
not_holds_at(in(r4),54).
not_holds_at(in(r3),54).
not_holds_at(in(r2),54).
   :
```

Code snippet D.15: Example negative fluent trace data for the Robot Navigation Control program.

```
%%%% holds_at facts for time point 0
holds_at(loc(corner(c7),ahead),0).
holds_at(in(r2),0).
holds_at(facing(north),0).

%%%% holds_at facts for time point 1
holds_at(loc(corner(c7),ahead),1).
holds_at(in(r2),1).
holds_at(facing(north),1).

%%%% holds_at facts for time point 2
holds_at(loc(corner(c7),ahead),2).
holds_at(in(r2),2).
holds_at(facing(north),2).

%%%% holds_at facts for time point 3
holds_at(loc(corner(c7),ahead),3).
holds_at(in(r2),3).
holds_at(facing(north),3).

%%%% holds_at facts for time point 4
holds_at(loc(corner(c7),ahead),4).
holds_at(in(r2),4).
holds_at(facing(north),4).

%%%% holds_at facts for time point 5
holds_at(loc(corner(c7),behind),5).
holds_at(in(r2),5).
holds_at(facing(east),5).

%%%% holds_at facts for time point 6
holds_at(loc(corner(c8),ahead),6).
holds_at(in(r2),6).
holds_at(facing(east),6).

%%%% holds_at facts for time point 7
holds_at(loc(corner(c8),ahead),7).
holds_at(in(r2),7).
holds_at(facing(east),7).

%%%% holds_at facts for time point 8
holds_at(loc(corner(c8),ahead),8).
holds_at(in(r2),8).
holds_at(facing(east),8).

%%%% holds_at facts for time point 9
holds_at(loc(corner(c8),ahead),9).
holds_at(in(r2),9).
holds_at(facing(east),9).

%%%% holds_at facts for time point 10
holds_at(loc(corner(c8),behind),10).
holds_at(in(r2),10).
holds_at(facing(south),10).

%%%% holds_at facts for time point 11
holds_at(loc(corner(c8),behind),11).
holds_at(in(r2),11).
holds_at(facing(south),11).

%%%% holds_at facts for time point 12
holds_at(loc(corner(c9),ahead),12).
holds_at(in(r2),12).
holds_at(facing(south),12).

%%%% holds_at facts for time point 13
holds_at(loc(corner(c9),ahead),13).
holds_at(in(r2),13).
holds_at(facing(south),13).

%%%% holds_at facts for time point 14
holds_at(loc(corner(c10),behind),14).
holds_at(in(r2),14).
holds_at(facing(south),14).
    :
    :

%%%% holds_at facts for time point 20
holds_at(loc(corner(c11),behind),20).
holds_at(in(r2),20).
holds_at(facing(west),20).

%%%% holds_at facts for time point 21
holds_at(loc(corner(c11),behind),21).
holds_at(in(r2),21).
holds_at(facing(west),21).

%%%% holds_at facts for time point 22
holds_at(loc(corner(c11),behind),22).
holds_at(in(r2),22).
holds_at(facing(west),22).

%%%% holds_at facts for time point 23
holds_at(loc(corner(c11),behind),23).
holds_at(in(r2),23).
holds_at(facing(west),23).

%%%% holds_at facts for time point 24
holds_at(loc(corner(c12),ahead),24).
holds_at(in(r2),24).
holds_at(facing(west),24).

%%%% holds_at facts for time point 25
holds_at(loc(corner(c12),ahead),25).
holds_at(in(r2),25).
holds_at(facing(west),25).

%%%% holds_at facts for time point 26
holds_at(loc(door(d3),in),26).
holds_at(in(r2),26).
holds_at(facing(south),26).

%%%% holds_at facts for time point 27
holds_at(loc(door(d3),in),27).
holds_at(in(r2),27).
holds_at(facing(south),27).

%%%% holds_at facts for time point 28
holds_at(loc(corner(c27),behind),28).
holds_at(in(r4),28).
holds_at(facing(east),28).

%%%% holds_at facts for time point 29
holds_at(loc(corner(c27),behind),29).
holds_at(in(r4),29).
holds_at(facing(east),29).

%%%% holds_at facts for time point 30
holds_at(loc(corner(c28),ahead),30).
holds_at(in(r4),30).
holds_at(facing(east),30).

%%%% holds_at facts for time point 31
holds_at(loc(corner(c28),ahead),31).
holds_at(in(r4),31).
holds_at(facing(east),31).

%%%% holds_at facts for time point 32
holds_at(loc(corner(c29),behind),32).
holds_at(in(r4),32).
holds_at(facing(east),32).

%%%% holds_at facts for time point 33
holds_at(loc(corner(c29),behind),33).
holds_at(in(r4),33).
holds_at(facing(east),33).

%%%% holds_at facts for time point 34
holds_at(loc(corner(c30),ahead),34).
holds_at(in(r4),34).
holds_at(facing(east),34).
    :
    :

%%%% holds_at facts for time point 40
holds_at(loc(corner(c30),behind),40).
holds_at(in(r4),40).
holds_at(facing(south),40).

%%%% holds_at facts for time point 41
holds_at(loc(corner(c30),behind),41).
holds_at(in(r4),41).
holds_at(facing(south),41).

%%%% holds_at facts for time point 42
holds_at(loc(corner(c30),behind),42).
holds_at(in(r4),42).
holds_at(facing(south),42).

%%%% holds_at facts for time point 43
holds_at(loc(corner(c30),behind),43).
holds_at(in(r4),43).
holds_at(facing(south),43).

%%%% holds_at facts for time point 44
holds_at(loc(corner(c31),ahead),44).
holds_at(in(r4),44).
holds_at(facing(south),44).

%%%% holds_at facts for time point 45
holds_at(loc(corner(c31),ahead),45).
holds_at(in(r4),45).
holds_at(facing(south),45).

%%%% holds_at facts for time point 46
holds_at(loc(corner(c31),ahead),46).
holds_at(in(r4),46).
holds_at(facing(south),46).

%%%% holds_at facts for time point 47
holds_at(loc(corner(c31),ahead),47).
holds_at(in(r4),47).
holds_at(facing(south),47).

%%%% holds_at facts for time point 48
holds_at(loc(corner(c31),ahead),48).
holds_at(in(r4),48).
holds_at(facing(south),48).

%%%% holds_at facts for time point 49
holds_at(loc(corner(c31),ahead),49).
holds_at(in(r4),49).
holds_at(facing(south),49).

%%%% holds_at facts for time point 50
holds_at(loc(corner(c31),behind),50).
holds_at(in(r4),50).
holds_at(facing(west),50).

%%%% holds_at facts for time point 51
holds_at(loc(corner(c31),behind),51).
holds_at(in(r4),51).
holds_at(facing(west),51).

%%%% holds_at facts for time point 52
holds_at(loc(corner(c31),behind),52).
holds_at(in(r4),52).
holds_at(facing(west),52).

%%%% holds_at facts for time point 53
holds_at(loc(corner(c31),behind),53).
holds_at(in(r4),53).
holds_at(facing(west),53).

%%%% holds_at facts for time point 54
holds_at(loc(corner(c32),ahead),54).
holds_at(in(r4),54).
holds_at(facing(west),54).
    :
    :
```

Code snippet D.16: Observations of dynamic behaviour for the Robot Navigation Control program: Positive fluent traces

Table D.1: Accuracies for the 30 induced sets of DSAs for the Robot Navigation Control program using ALECTO. The average accuracy across the 30 trials is 70.2%.

DSA Theory	Pred. ⇑	Actual ⇑	Pred. ⇓	Actual ⇓	Total Pred. ⇳	Total Act. ⇳	Accuracy	σ
1	735	911	814	911	1549	1822	85.02%	10.89%
2	788	916	548	916	1336	1832	72.93%	20.80%
3	588	881	581	881	1169	1762	66.34%	2.90%
4	619	920	531	920	1150	1840	62.50%	14.13%
5	668	889	583	889	1251	1778	70.36%	11.36%
6	702	897	513	897	1215	1794	67.73%	17.56%
7	716	920	527	920	1243	1840	67.55%	16.66%
8	595	916	606	916	1201	1832	65.56%	7.76%
9	791	924	534	924	1325	1848	71.70%	18.57%
10	580	897	594	897	1174	1794	65.44%	8.39%
11	583	902	594	902	1177	1804	65.24%	8.24%
12	594	911	601	911	1195	1822	65.59%	8.29%
13	709	901	518	901	1227	1802	68.09%	17.26%
14	721	921	528	921	1249	1842	67.81%	17.14%
15	684	909	598	909	1282	1818	70.52%	9.50%
16	604	895	514	895	1118	1790	62.46%	14.94%
17	615	909	523	909	1138	1818	62.60%	14.87%
18	724	922	528	922	1252	1844	67.90%	16.57%
19	683	911	525	911	1208	1822	66.30%	14.08%
20	615	912	527	912	1142	1824	62.61%	14.21%
21	786	922	635	922	1421	1844	77.06%	12.53%
22	847	927	639	927	1486	1854	80.15%	11.89%
23	555	927	561	927	1116	1854	60.19%	10.94%
24	882	909	599	909	1481	1818	81.46%	17.00%
25	814	941	633	941	1447	1882	76.89%	11.13%
26	630	927	542	927	1172	1854	63.21%	13.55%
27	857	909	598	909	1455	1818	80.03%	15.65%
28	810	940	629	940	1439	1880	76.54%	11.30%
29	889	945	624	915	1513	1860	81.34%	14.93%
30	740	924	636	924	1376	1848	74.46%	9.17%

Notes:

1. *DSA Theory* is the index of the set of clauses induced.
2. *Pred.* ⇑ is the number of *initiates/3* atoms that the DSA theory predicts.
3. *Actual* ⇑ is the actual number of *initiates/3* atoms that are true.
4. *Pred.* ⇓ is the number of *terminates/3* atoms that the DSA theory predicts.
5. *Actual* ⇓ is the actual number of *terminates/3* atoms that are true.
6. *Total Pred.* ⇳ is the number of *initiates/3* and *terminates/3* transition atoms that the DSA theory predicts.
7. *Total Act.* ⇳ is the actual number of *initiates/3* and *terminates/3* transition atoms that are true.
8. *Accuracy* is the ratio of predicted transition atoms to actual transition atoms (item 6 divided by item 7).
9. σ is the sample standard deviation of the individual accuracies.

Table D.2: Miscellaneous statistics for generating the 30 induced sets of DSAs for the Robot Navigation Control program using `ALECTO`.

Run Number	Start set size (atoms)	Induction CPU time (seconds)	Abduction CPU time (seconds)	Number of events in narrative	Total CPU time (seconds)	Number of clauses induced
1	159	137	3806	22	3942	15
2	168	118	3666	18	3784	14
3	107	204	8770	40	8975	14
4	83	115	1090	16	1205	14
5	95	161	2498	34	2659	14
6	177	229	5700	29	5929	15
7	186	130	2986	17	3117	14
8	158	188	4005	20	4193	12
9	183	133	2457	13	2590	12
10	168	286	6971	30	7257	14
11	152	162	5113	29	5275	14
12	119	222	4329	22	4551	12
13	160	131	5339	27	5469	13
14	265	110	3096	16	3206	13
15	160	185	4109	22	4294	15
16	120	295	5715	32	6010	13
17	69	280	2349	24	2629	14
18	178	160	2640	16	2800	14
19	152	268	3814	21	4082	13
20	152	306	4368	23	4674	13
21	188	195	3380	16	3575	14
22	180	222	2616	10	2839	18
23	166	173	2240	10	2414	14
24	78	423	4016	24	4439	20
25	309	63	1872	3	1935	4
26	144	201	1460	11	1661	11
27	154	356	4469	26	4825	15
28	258	109	1343	4	1452	6
29	144	387	3154	19	3541	20
30	111	199	2175	17	2374	10

Notes:
1. *Run Number* is the training set index.
2. *Start set size* is the number of atoms produced by extraction-case abduction.
3. *Induction CPU time* is the CPU time (in seconds) for induction.
4. *Abduction CPU time* is the CPU time (in seconds) for performing the extraction-case abduction.
5. *Number of events in narrative* is the number of *happens/2* events in the narrative of the run.
6. *Total CPU time* is the sum of the Induction and Abduction CPU times (in seconds).
7. *Number of Clauses induced* As it says.
8. The machine used for the experiments was a dual processor Intel Pentium III, 550MHz, 512 MB RAM.

```
initiates(follow_wall,loc(corner(C2),ahead),T):-
    holds_at(loc(corner(C1),behind),T),
    next_visible_corner(C1,C2,T).
terminates(follow_wall,loc(corner(C1),behind),T):-
    holds_at(loc(corner(C1),behind),T).

initiates(go_straight,loc(corner(C2),behind),T):-
    holds_at(loc(corner(C1),ahead),T), door(_D,C1,C2).
terminates(go_straight,loc(corner(C1),ahead),T):-
    holds_at(loc(corner(C1),ahead),T),door(_D,C1,_C2).

initiates(go_straight,loc(corner(C2),behind),T):-
    holds_at(loc(corner(C1),ahead),T), door(_D,C2,C1).
terminates(go_straight,loc(corner(C1),ahead),T):-
    holds_at(loc(corner(C1),ahead),T),door(_D,_C2,C1).

initiates(turn(left),loc(corner(C2),behind),T):-
    holds_at(loc(door(D),in),T), holds_at(in(R1),T), connects(D,R1,R2),
    door(D,C1,C2), nearest_corner(R2,C1,C2).
terminates(turn(left),loc(corner(C1),ahead),T):-
    holds_at(loc(corner(C1),ahead),T), door(D,C1,_C2), holds_at(door_open(D),T).

initiates(turn(right),loc(corner(C2),behind),T):-
    holds_at(loc(door(D),in),T), holds_at(in(R1),T), connects(D,R1,R2), door(D,C2,C1),
    nearest_corner(R2,C2,C1).
terminates(turn(right),loc(corner(C1),ahead),T):-
    holds_at(loc(corner(C1),ahead),T), door(D,_C2,C1), holds_at(door_open(D),T).

initiates(turn(left),loc(door(D),in),T):-
    holds_at(loc(corner(C1),ahead),T), door(D,C1,_C2), holds_at(door_open(D),T).
terminates(turn(left),loc(door(D),in),T):-
    holds_at(loc(door(D),in),T).

initiates(turn(right),loc(door(D),in),T):-
    holds_at(loc(corner(C1),ahead),T),
    door(D,_C2,C1),
    holds_at(door_open(D),T).
terminates(turn(right),loc(door(D),in),T):-
    holds_at(loc(door(D),in),T).

initiates(turn(left),in(R2),T):-
    holds_at(loc(door(D),in),T), holds_at(in(R1),T), connects(D,R1,R2).
terminates(turn(left),in(R1),T):-
    holds_at(loc(door(_D),in),T), holds_at(in(R1),T).
initiates(turn(right),in(R2),T):-
    holds_at(loc(door(D),in),T), holds_at(in(R1),T), connects(D,R1,R2).
terminates(turn(right),in(R1),T):-
    holds_at(loc(door(_D),in),T), holds_at(in(R1),T).

initiates(turn(LorR),loc(corner(C),behind),T):-
    holds_at(loc(corner(C),ahead),T), inner(C), holds_at(in(Room),T),
    nearest_corner(Room,C,CNext), short_wall_direction(C,CNext,Direction1),
    holds_at(facing(Direction0),T), change_direction(LorR,Direction0,Direction1).
terminates(turn(LorR),loc(corner(C),ahead),T):-
    holds_at(loc(corner(C),ahead),T), inner(C), holds_at(in(Room),T),
    nearest_corner(Room,C,CNext), short_wall_direction(C,CNext,Direction1),
    holds_at(facing(Direction0),T), change_direction(LorR,Direction0,Direction1).

initiates(turn(LorR),facing(Direction1),T):-
    holds_at(loc(corner(C),ahead),T), inner(C), holds_at(in(Room),T),
    nearest_corner(Room,C,CNext), short_wall_direction(C,CNext,Direction1),
    holds_at(facing(Direction0),T), change_direction(LorR,Direction0,Direction1).
terminates(turn(LorR),facing(Direction0),T):-
    holds_at(loc(corner(C),ahead),T), inner(C), holds_at(in(Room),T),
    nearest_corner(Room,C,CNext), short_wall_direction(C,CNext,Direction1),
    holds_at(facing(Direction0),T), change_direction(LorR,Direction0,Direction1).

initiates(turn(LorR),facing(Direction1),T):-
    holds_at(facing(Direction0),T), holds_at(loc(corner(C1),ahead),T),
    (door(D,C1,_C2);door(D,_C2,C1)),!,
    holds_at(door_open(D),T), change_direction(LorR,Direction0,Direction1),
    holds_at(in(R1),T), connects(D,R1,R2), next_room_direction(R1,R2,Direction1).
terminates(turn(LorR),facing(Direction0),T):-
    holds_at(facing(Direction0),T), holds_at(loc(corner(C1),ahead),T),
    (door(D,C1,_C2);door(D,_C2,C1)),!, holds_at(door_open(D),T),
    change_direction(LorR,Direction0,Direction1), holds_at(in(R1),T),
    connects(D,R1,R2), next_room_direction(R1,R2,Direction1).

initiates(turn(LorR),facing(Dir1),T) :-
   holds_at(loc(door(_D),in),T), holds_at(facing(Dir0),T),
   change_direction(LorR,Dir0,Dir1).
terminates(turn(_LorR),facing(Direction0),T):-
    holds_at(facing(Direction0),T), holds_at(loc(door(_D),in),T).
```

Code snippet D.17: Modified Robot Navigation Control DSAs used to generate fluent traces for the experiments.

Bibliography

[Abeysinghe, 1993] Geetha Kalyani Abeysinghe. *Event Calculus to Support Temporal Reasoning in a Clinical Domain*. Ph.D. thesis, University of Southampton, 1993.

[Aleph-One, 1996] Aleph-One, Smashing The Stack For Fun And Profit 1996, Phrack 49.

[Aliseda, 2000] Atocha Aliseda, Abduction as epistemic change: Charles S.Peirce and epistemic theories in artificial intelligence 2000, `http://www.rz.uni-frankfurt.de/~wirth/tt.htm`.

[Apt *et al.*, 1987] K. R. Apt, A. Blair, and A. Walker. Towards a Theory of Declarative Knowledge. In J. Minker, editor, *Foundations of Deductive Databases and Lgic Programming*, p. 89-148. Morgan Kaufmann, 1987.

[Bacon, 1994] Francis Bacon. *Novum Organum*. eds. P. Urbach and J. Gibson. Open Court Publishing Company, 1994.

[Bain and Sammut, 1999] Michael Bain and Claude Sammut. A framework for behavioural cloning. In K. Furukawa, D. Michie, andS. Muggleton. editors, *Machine Intelligence 15: machine intelligence and inductive learning*, Oxford, 1999. Oxford University Press.

[Blockeel, 1998] Hendrik Blockeel. *Top-down Induction of First Order Logical decision Trees*. PhD Thesis, Katholieke Universiteit Leuven, 1998.

[Blum and Blum, 1975] L. Blum and M. Blum. Towards a mathematical theory of inductive inference. *Information and Control*, 28:125-155, 1975.

[Bratko, 1994] I. Bratko. Machine Learning and Qualitative Reasoning. *Machine Learning*, 14(3):305-312, 1994.

[Bratko *et al.*, 1992] I. Bratko, S. Muggleton, and A. Varsek. Learning Qualitative Models of Dynamic Systems. In S. Muggleton, editor, *Inductive Logic Programming*, p. 437-452. Academic Press, London, 1992.

[Bryant *et al.*, 1999] Christopher H. Bryant, Stephen H. Muggleton, C. David Page, and Micheal J.E. Sternberg. Combining active learning with inductive logic programming to close the loop in machine learning. In S. Colton editor, *Proceedings of AISB'99 Symposium on AI and Scientific Creativity*, p. 59-64. 1999. The Society for the Study of Artificial Intelligence and Simulation of Behaviour (AISB).

[Bulitko and Wilkins, 1999] V.V. Bulitko and D.C. Wilkins. Learning to Envision: An Intelligent Agent for Ship Damage Control. In *Proceedings of the ACAI'99 conference*, Crete, Greece, 1999.

[Buntine, 1988] W. Buntine. Generalized subsumption and its application to induction and redundancy. *Artificial Intelligence*, 36:375-399, 1988.

[Clark, 1978] K. L. Clark. Negation as Failure. In H. Gallaire and J. Minker, editors, *Logic and Databases*, p. 293-322. Plenum Press, 1978.

[Clocksin and Mellish, 1987] William F. Clocksin and Christopher S. Mellish. *Programming in Prolog.* 3rd ed. Springer-Verlag, Berlin, 1987.

[Console *et al.*, 1991] L. Console, D.T. Dupre, and P. Torasso. On the Relationship Between Abduction and Deduction. *Journal of Logic and Computation*, 1(5):661–690, 1991.

[Costa *et al.*, 2000] Vitor Santos Costa, Luis Damas, Rogerio Reis, and Ruben Azevedo, YAP Prolog User's Manual 2000, `http://www.ncc.up.pt/~vsc/Yap/`.

[Cox and Pietrzykowski, 1986] P.T. Cox and T. Pietrzykowski. Causes for Events: Their Computation and Applications. In *8th Conference on Automated Deduction*, p. 608–621. Oxford, England, 1986.

[Cox and Pietrzykowski, 1987] P.T. Cox and T. Pietrzykowski. General Diagnosis by Abductive Inference. In *1987 Symposium on Logic Programming*, p. 183–189. San Francisco, California, 1987.

[Cussens, 1997] James Cussens. Part-of-speech tagging using Progol. In *Inductive Logic Programming: Proceedings of the 7th International Workshop (ILP-97)*, p. 93-108. Prague, 1997. Springer.

[Cussens *et al.*, 1997] James Cussens, David Page, Stephen Muggleton, and Ashwin Srinivasan. Using Inductive Logic Programming for Natural Language Processing. In W. Daelemans, T. Weijters, andA. van der Bosch editors, *ECML'97 – Workshop Notes on Empirical Learning of Natural Language Tasks*, p. 25-34. University of Economics, Prague. Invited keynote paper, 1997.

[Cypher, 1993] Allen Cypher ed. *Watch What I Do: Programming by Demonstration*. 1993, MIT Press.

[Davey and Priestley, 1990] B. A. Davey and H. A. Priestley. *Introduction to Lattices and Order*. Cambridge University Press, 1990.

[Davis, 1990] Ernest Davis. *Representations of Commonsense Knowledge*. Morgan Kaufmann, 1990.

[De Raedt, 1992] Luc De Raedt. *Interactive Theory Revision: An Inductive Logic Programming Approach*. Academic Press, London, 1992.

[De Raedt and Dzeroski, 1994] Luc De Raedt and Saso Dzeroski. First order jk-clausal theories are PAC-learnable. *Artificial Intelligence*, 70:375–392, 1994.

[De Raedt *et al.*, 1993] L. De Raedt, N. Lavrac, and S. Dzeroski. Multiple predicate learning. In S. Muggleton editor, *Proceedings of the 3rd International Workshop on Inductive Logic Programming*, p. 221–240. J. Stefan Institute, Slovenia, 1993.

[Denecker, 1995] Marc Denecker. *Personal communication*, 1995.

[Denecker *et al.*, 1996] Marc Denecker, Kristof Van Belleghem, Guy Duchatelet, Frank Piessens, and Danny De Schreye. A Realistic Experiment in Knowledge Representation in Open Event Calculus: Protocol Specification. In M. Maher editor, *Proceedings of JICSLP'96, the Joint International Conference and Symposium on Logic Programming*, p. 170-184. Bonn, Germany, 1996. MIT Press.

[Denecker and Kakas, 2000] Marc Denecker and Antonis C. Kakas. Special issue: abductive logic programming. *Journal of Logic Programming*, 44(1-4), 2000.

[Denecker and Schreye, 1985] Marc Denecker and Danny De Schreye. Representing Incomplete knowledge in Abductive Logic Programming. *Journal of Logic and Computation*, 5(5):553 - 577, 1985.

[Deransart *et al.*, 1996] P. Deransart, A. Ed-Dbali, and L. Cervoni. *Prolog: The Standard*. Springer-Verlag, Heidleberg, 1996.

[Dimopoulos and Kakas, 1996] Y. Dimopoulos and A. Kakas. Abduction and Inductive Learning. In L. De Raedt editor, *5th International Workshop on Inductive Logic Programming*, p. 25-28. Leuven, Belgium, 1996.

[Dimopoulos and Kakas, 1996] Y. Dimopoulos and A. Kakas. Abduction and Inductive Learning. In L. De Raedt, editor, *Advances in Inductive Logic Programming*, p. 144-171. 1996.

[Dryllerakis, 1992] K. J. Dryllerakis. Results from Implementing the Event Calculus in CLP(R). Technical report Department of Computing, Imperial College, London, 1992.

[Eastmond, 1996] Matthew Eastmond. *ILP applied to traffic flow problems on the M25*. M.Sc. thesis, Oxford University, 1996.

[Eshghi, 1988] K Eshghi. Abductive Planning with Event Calculus. In *Proceedings of the Fifth International Conference on Logic Programming*, p. 562 - 579. 1988.

[Fayyad *et al.*, 1996] Usama M. Fayyad, Gregory Piatetsky-Shapiro, Padhraic Smyth, and Ramasamy Uthurusamy eds. *Advances in Knowledge Discovery and Data Mining*. 1996, MIT Press.

[Feng, 1992] C. Feng. Inducing Temporal Fault Diagnostic Rules from a Qualitative Model. In S. Muggleton, editor, *Inductive Logic Programming*, p. 473-494. Academic Press, London, 1992.

[Finger and Genesereth, 1985] J. J. Finger and M. R. Genesereth. RESIDUE: a deductive approach to the design synthesis. Technical report CS-85-1035, Stanford University, 1985.

[Finn *et al.*, 1998] Paul Finn, Stephen Muggleton, C. David Page, and Ashwin Srinivasan. Pharmacophore discovery using the inductive logic programming system Progol. *Machine Learning*, 30:241-271, 1998.

[Flach and Kakas, 2000] Peter A. Flach and Antonis C. Kakas eds. *Abduction and Induction: Essays on their Relation and Integration*. Applied Logic Series, D. M. Gabbay. Vol. 18. 2000, Kluwer Academic Publishers.

[Frisch, 1999] Alan M. Frisch. Sorted Downward Refinement: Building Background Knowledge into a Refinement Operator for Inductive Logic Programming. In S. Dzeroski and P. Flach editors, *9th International Workshop on Inductive Logic Programming (ILP-99)*, p. 104-115. Bled, Slovenia, 1999. Springer-Verlag.

[Garfinkel and Spafford, 1996] S. Garfinkel and G. Spafford. *Practical UNIX and Internet Security*. O'Reilly and Associates, Sebastopol, 1996.

[Gillies, 1996] Donald Gillies. *Artificial intelligence and scientific method*. Oxford University Press, Oxford, 1996.

[Hamilton, 1988] A. G. Hamilton. *Logic for Mathematicians*. Revised ed. Cambridge University Press, Cambridge, 1988.

[Hume and Sammut, 1992] D. Hume and C. Sammut. Applying Inductive Logic Programming in Reactive Environments. In S. Muggleton, editor, *Inductive Logic Programming*, p. 539-549. Academic Press, London, 1992.

[Inoue, 1992] Katsumi Inoue. Linear Resolution for Consequence Finding. *Artificial Intelligence*, 56:301 - 353, 1992.

[Ito and Yamamoto, 1998] K. Ito and A. Yamamoto. Finding hypotheses from examples by computing the least generlisation of bottom clauses. In S. Arikawa and H. Motoda editors, *Proceedings of Discovery Science '98*, p. 303–314. 1998. Springer, Berlin.

[Jaffar and Lassez, 1987] J. Jaffar and J-L. Lassez. Constraint Logic Programming. In *14th ACM Symposium on Principles of Programming Languages*, p. 111-119. 1987.

[Jaffar and Lassez, 1987] J. Jaffar and J.L. Lassez. Constraint Logic Programming. In *the Proceedings of the 14th ACM Symposium on Principles of Programmming Languages*, p. 111-119. Munich, 1987.

[Jevons, 1877] W.Stanley Jevons. *The Principles of Science: A treatise on Logic and Scientific Method*. Second ed. Macmillan and Co., London, 1877.

[Kakas *et al.*, 1993] A. C. Kakas, R. A. Kowalski, and F. Toni. Abductive Logic Programming. *Journal of Logic and Computation*, 2(6):719-770, 1993.

[Kakas and Michael, 1995] A. C. Kakas and A. Michael. Integrating Abductive and Constraint Logic Programming. In *Proceedings of the 12th International Conference on Logic Programming*, Japan, 1995.

[Kakas and Riguzzi, 2000] Antonis C. Kakas and Fabrizio Riguzzi. Abductive Concept Learning. *New Generation Computing*, 18:243-294, 2000.

[King *et al.*, 1992] R. King, S. Muggleton, R. Lewis, and M. Sternberg. Drug design by machine learning: The use of inductive logic programming to model the structure-activity relationships of trimethoprim analogues binding to dihydrofolate reductase. In *Proceedings of the National Academy of Sciences*, p. 11322-11326. 1992.

[King and Srinivasan, 1997] R. King and A. Srinivasan. The discovery of indicator variables for QSAR using inductive logic programming. *Journal of Computer-Aided Molecular Design*, 11:571-580, 1997.

[King *et al.*, 1996] R.D. King, S. Muggleton, A. Srinivasan, and M.J.E. Sternberg. Structure-activity relationships derived by machine learning:

The use of atoms and their bond connectivities to predict mutagenicity by inductive logic programming. *Proceedings of the National Academy of Sciences*, 93:438-442, 1996.

[King and Srinivasan, 1996] R.D. King and A. Srinivasan. Prediction of rodent carcinogenicity bioassays from molecular structure using inductive logic programming. *Environmental Health Perspectives*, 104(5):1031-1040, 1996.

[Kowalski, 1979] Robert Kowalski. *Logic for Problem Solving*. Elsevier, North Holland, 1979.

[Kowalski and Sergot, 1986] Robert Kowalski and Marek Sergot. A Logic-based Calculus of Events. *New Generation Computing*, 4:67 - 95, 1986.

[Kruijff, 1997] Geert-Jan M. Kruijff. Concerning logics of abduction - The relation between abduction and induction. In P. Flach and A. Kakas editors, *Proceedings of the IJCAI'97 workshop on Abduction and Induction in AI*, Nagoya, Japan, 1997.

[Kuhn, 1962] Thomas S. Kuhn. *The structure of scientific revolutions*. 2nd, enlarged ed. University of Chicago Press, 1970, Chicago, 1962.

[Kuipers, 1986] B Kuipers. Qualitative simulation. *Artificial Intelligence*, 24:169-203, 1986.

[Kunen, 1989] K. Kunen. Signed data Dependiencies in Logic Programs. *Journal of Logic Programmng*, 7:231-245, 1989.

[Lavrac and Dzeroski, 1994] Nada Lavrac and Saso Dzeroski. *Inductive logic programming : techniques and applications*. Ellis Horwood series in artificial intelligence, Ellis Horwood, New York, London, 1994.

[Lloyd, 1993] John W. Lloyd. *Foundations of Logic Programming*. Springer-Verlag, 1993.

[Lorenzo and Otero, 1999] David Lorenzo and Ramon P. Otero. Learning Action Theories with Causality. In S. Dzeroski and P. Flach editors, *Inductive Logic Programming: Late-Breaking Papers*, p. 36- 41. Bled, Slovenia, 1999.

[Lorenzo and Otero, 2000] David Lorenzo and Ramon P. Otero. Using a ILP algorithm to learn Logic Programs for Reasoning about Actions. In J. Cussens and A. Frisch editors, *Inductive Logic Programming: Work-in-Progress Reports*, p. 163 - 171. London, 2000.

[McCarthy, 1959] John McCarthy. Programs with Common Sense. In *Proceedings of the Symposium on Mechanisation of Thought Processes*, London, 1959. Her Majesty's Stationery Office.

[McCarthy, 1968] John McCarthy. Programs with Common Sense. In M. Minsky, editor, *Semantic Information Processing*, p. 403-418. The MIT Press, 1968.

[McCarthy and Hayes, 1969] J. McCarthy and P.J. Hayes. Some philosophical problems from the standpoint of Artificial Intelligence. In B. Meltzer and D. Michie, editors, *Machine Intelligence 4*, p. 463 - 502. Edinburgh University Press, 1969.

[Medawar, 1979] Peter Medawar. *Advice to a Young Scientist*. Basic Books, 1979.

[Medawar, 1982] Peter Medawar. *Pluto's Republic (incorporating 'The art of the soluble' and 'Induction and intuition in scientific thought')*. Oxford University Press, Oxford, 1982.

[Michie, 1988] Donald Michie. Machine Learning in the Next Five Years. In D. Sleeman and J. Richmond editors, *Third European Workshop Session on Learning (EWSL '88)*, p. 107 - 122. University of Strathclyde, 1988. Pitman, London.

[Mill, 1872] John Stuart Mill. *A System of Logic*. 8 ed. London, 1872.

[Mitchell, 1982] Thomas Mitchell. Generalization as search. *Artificial Intelligence*, 18(2):203 - 226, 1982.

[Mitchell, 1997] Thomas Mitchell. *Machine Learning*. McGraw-Hill Book Company, Department of Computer Science, Carnegie Mellon University, Pittsburgh, 1997.

[Moyle and Heasman, 2002] S. Moyle and J. Heasman. Applying ILP to the learning of intrusion strategies. Technical report PRG-RR-02-03, Oxford University Computing Laboratory, Oxford, 2002.

[Moyle and Muggleton, 1997] S. Moyle and S. Muggleton. Learning Programs in the Event Calculus. In N. Lavrac and S. Dzeroski editors, *7th International Workshop, ILP-97*, p. 205–212. Prague, 1997. Springer.

[Moyle, 1999] S. A. Moyle. Learning about Action and Change: An Inductive Logic Programming approach. Technical report PRG-TR-6-99, Oxford University Computing Laboratory, University of Oxford, Oxford, 1999.

[Muggleton, 1987] S. Muggleton. Duce, an oracle based approach to constructive induction. In *IJCAI-87*, p. 287-292. 1987. Kaufmann.

[Muggleton, 1988] S. Muggleton. A strategy for constructing new predicates in first order logic. In *Proceedings of the Third European Working Session on Learning*, p. 123-130. 1988. Pitman.

[Muggleton, 1992] Stephen Muggleton ed. *Inductive logic programming.* The A.P.I.C. series, Vol. no. 38. 1992, Academic Press: London.

[Muggleton, 1995] S. Muggleton. Inverse Entailment and Progol. *New Generation Computing*, 13(3 and 4):245–286, 1995.

[Muggleton, 1996] S. Muggleton. Learning from positive data. In S. Muggleton, editor, *Proceedings of the 6th International Workshop on Inductive Logic Programming*, p. 225-244. 1996.

[Muggleton, 1998] Stephen Muggleton. Completing inverse entailment. In C. D. Page editor, *Proceedings of the Eighth International Workshop on Inductive Logic Programming (ILP-98), LNAI 1446*, p. 245-249. 1998. Springer-Verlag, Berlin.

[Muggleton, 1999] Stephen Muggleton. Scientific knowledge discovery using inductive logic programming. *Communications of the ACM*, 42(11):42-46, 1999.

[Muggleton and Bryant, 2000] Stephen Muggleton and Christopher Bryant. Theory Completion Using Inverse Entailment. In J. Cussens and A. Frisch editors, *Inductive Logic Programming: 10th International Conference*, p. 130 - 146. London, 2000. Springer.

[Muggleton and de Raedt, 1994] Stephen Muggleton and Luc de Raedt. Inductive Logic Programming: Theory and Methods. *Journal of Logic Programming*, 19, 20:629 - 679, 1994.

[Muggleton and Feng, 1992] S. Muggleton and C. Feng. Efficient Induction in Logic Programs. In S. Muggleton, editor, *Inductive Logic Programming*, p. 281-298. Academic Press, 1992.

[Muggleton *et al.*, 1992] S. Muggleton, R. King, and M. Sternberg. Protein secondary structure prediction using logic-based machine learning. *Protein Engineering*, 5(7):647-657, 1992.

[Muggleton and Page, 1994] S. Muggleton and C.D. Page. A learnability model for universal representations. Technical report PRG-TR-3-94, Oxford University Computing Laboratory, Oxford, 1994.

[Muggleton *et al.*, 1996] S. Muggleton, C.D. Page, and A. Srinivasan. Learning to read by Theory Revision. Technical report PRG-TR-26-96, Oxford University Computing Laboratory, Oxford, 1996.

[Nienhuys-Cheng and de Wolf, 1997] S-H Nienhuys-Cheng and R de Wolf. *Foundations of inductive logic programming.* Lecture Notes in Artificial Intelligence, eds. J. G. Carbonell and J. Siekmann. Vol. 1228. Springer, Berlin, 1997.

[Page, 2000] C. David Page. ILP: Just Do It. In J. Cussens and A. Frisch editors, *Inductive Logic Programming: 10th International Conference*, p. 130 - 146. London, 2000. Springer.

[Parson *et al.*, 1999] R. Parson, K. Khan, and S. Muggleton. Theory recovery. In S. Dzeroski and P. Flach editors, *9th International Workshop on Inductive Logic Programming (ILP-99)*, p. 257 - 267. Bled, Slovenia, 1999. Springer-Verlag.

[Peirce, 1931] Charles Sanders Peirce. Elements of logic. In C. Hartshorne and P. Weiss, editors, *Collected papers of Charles Sanders Peirce*, Harvard University Press, Cambridge, Massachusetts, 1931.

[Plotkin, 1969] Gordon D. Plotkin. A Note on Inductive Generalization. In B. Meltzer and D. Michie editors, *Machine intelligence 4*, Edinburgh, 1969. Edinburgh University Press, Edinburgh.

[Plotkin, 1971] Gordon D. Plotkin. *Automatic Methods of Inductive Inference*. Ph.D. thesis, Edinburgh University, 1971.

[Potter *et al.*, 1991] Ben Potter, Jane Sinclair, and David Till. *An Introduction to Formal Specification and Z*. Prentice Hall Series in Computer Science, ed. C. A. R. Hoare. Prentice Hall, 1991.

[Puget, 1997] J.-F. Puget. Inductive Logic Programming and Constraint Logic Programming (Abstract). In S. Dzeroski and N. Lavrac editors, *Proceedings of the 7th International Workshop on Inductive Logic Programming*, p. 33–36. 1997. Springer-Verlag.

[Reddy and Tadepalli, 1997] Chandra Reddy and Prasad Tadepalli. Learning Horn Definitioins with Equivalence and Membership Queries. In N. Lavrac and S. Dzeroski editors, *7th International Workshop, ILP-97*, p. 205–212. Prague, 1997. Springer.

[Robinson, 1965] J. A. Robinson. A Machine-Oriented Logic Based on the Resolution Principle. *Journal of the Association for Computing Machinery*, 12(1):23 - 41, 1965.

[Rouveirol, 1992] Celine Rouveirol. Extensions of Inversion of Resolution Applied to Theory Completion. In S. Muggleton, editor, *Inductive Logic Programming*, p. 63-92. Academic Press, 1992.

[Ryle, 1949] Gilbert Ryle. *The Concept of Mind*. Hutchison, London, 1949.

[Sablon, 1995] Gunther Sablon. *Iterative Versionspaces with an application in Inductive Logic Programming*. Ph.D. thesis, Katholieke Universiteit, 1995.

[Sadri and Kowalski, 1995] Fariba Sadri and Robert Kowalski. Variants of the Event Calculus. In *Proceedings of the 12th International Conference on Logic Programming*, Japan, 1995.

[Saith *et al.*, 1998] R.R. Saith, A. Srinivasan, D. Michie, and I.L. Sargent. The relationship between embryo, oocyte and follicular features and the developmental potential of human IVF embryos. *Human Reproduction Update*, 4(2):121-134, 1998.

[Sammut and Banerji, 1984] Claude Sammut and Ranan B. Banerji. Learning Concepts by Asking Questions. In R. S. Michalski, J. G. Carbonell, andT. M. Mitchell, editors, *Machine learning: an artificial intelligence approach*, Springer, Berlin, 1984.

[Schmidt-Schauss, 1988] M. Schmidt-Schauss. Implication of Clauses is Undecidable. *Theoretical Computer Science*, 59:287-296, 1988.

[Shanahan, 1989] Murray Shanahan. Prediction is Deduction but Explanation is Abduction. In *Preceedings of IJCAI 89*, p. 1055 - 1060. 1989.

[Shanahan, 1991] Murray Shanahan. Computational Aspects of the Event Calculus. Technical report Imperial College Department of Computing, London, 1991.

[Shanahan, 1996] Murray Shanahan. *Personal communication*, 1996.

[Shanahan, 1997] Murray Shanahan. *Solving the Frame Problem: A Mathematical Investigation of the Common Sense Law of Inertia*. MIT Press, 1997.

[Shanahan, 2000] Murray Shanahan. An Abductive Event Calculus Planner. *The Journal of Logic Programming*, 44:207– 239, 2000.

[Shanahan, 2000] Murray Shanahan. A Logical Account of the Common Sense Informatic Situation for a Mobile Robot. *Electronic Transactions on Artificial Intelligence*, , 2000.

[Shanahan and Witkowski, 1999] Murray Shanahan and Mark Witkowski. Robot Navigation and Map Building with the Event Calculus. In *Working Notes of the IJCAI 99 Workshop on Robot Action Planning*, 1999.

[Shanahan and Witkowski, 2000] Murray Shanahan and Mark Witkowski. High-Level Robot Control Through Logic. In *ATAL 2000*, 2000.

[Shapiro, 1983] E.Y. Shapiro. *Algorithmic Program Debugging*. MIT Press, 1983.

[Simon, 1984] Herbert A. Simon. Why should machines learn? In R. S. Michalski, J. G. Carbonell, andT. M. Mitchell, editors, *Machine learning: an artificial intelligence approach*, Springer, Berlin, 1984.

[Smythe *et al.*, 1982] G Smythe, M Duncan, J Bradshaw, W Cai, and R Symons. Serotoninergic control of growth hormone secretion: hypothalamic dopamine, norepinephrine and serotonin levels and metabolism in three hyposomatropic rat models and in normal rats. *Endocrinology*, 110(2):376–383, 1982.

[Srinivasan, 1999] Ashwin Srinivasan. A study of two sampling methods for analysing large datasets with ILP. *Data Mining and Knowledge Discovery*, 3(1):95-123, 1999.

[Srinivasan, 2000] Ashwin Srinivasan, The Aleph Manual 2000, `http://web.comlab.ox.ac.uk/oucl/research/areas/machlearn/Aleph/`.

[Srinivasan, 2000] Ashwin Srinivasan. A study of two probabilistic methods for searching large spaces with ILP. *Data Mining and Knowledge Discovery (under review)*, `ftp://ftp.comlab.ox.ac.uk/pub/Packages/ILP/Papers/AS/dami99.ps.gz`, 2000.

[Srinivasan and Camacho, 1999] Ashwin Srinivasan and Rui Camacho. Numerical reasoning with an ILP system capable of lazy evaluation and customised search. *Journal of Logic Programming*, 40(2/3):185-214, 1999.

[Srinivasan and King, 1999] A. Srinivasan and R.D. King. Using Inductive Logic Programming to construct Structure-Activity Relationships. In G. C. Gini and A. R. Katrizsky editors, *Predictive toxicology of chemicals: experiences and impact of AI tools (Papers from the 1999 AAAI Spring Symposium)*, p. 64-73. 1999. AAAI Press, Menlo Park, CA.

[Srinivasan *et al.*, 1994] Ashwin Srinivasan, Stephen Muggleton, and Michael Bain. The justification of logical theories based on data compression. In K. Furukawa, D. Michie, andS. Muggleton editors, *Machine Intelligence 13 - Machine Intelligence and Inductive Learning*, p. 87 - 122. Strathclyde University, 1994. Clarendon Press, Oxford.

[Sripada, 1995] Suryanarayana M. Sripada. Efficient Implementation of the Event Calculus for Temporal Database Applications. In *Proceedings of the 12th International Conference on Logic Programming*, Japan, 1995.

[Sternberg *et al.*, 1994] M. Sternberg, R. King, R. Lewis, and S. Muggleton. Application of machine learning to structural molecular biology. *Philosophical Transactions of the Royal Society B*, 344:365-371, 1994.

[Sternberg *et al.*, 1992] M. Sternberg, R. Lewis, R. King, and S. Muggleton. Modelling the structure and function of enzymes by machine learning. In *Proceedings of the Royal Society of Chemistry: Faraday Discussions*, p. 269-280. 1992.

[Sternberg *et al.*, 1994] M.J.E. Sternberg, J.D. Hirst, R.A. Lewis, R.D. King, A. Srinivasan, and S. Muggleton. Application of Machine Learning to Protein Structure Prediction and Drug Design. In S. Schulze-Kremer, editor, *Advances in Molecular Bioinformatics*, p. 1-8. IOS Press, 1994.

[Stickel, 1991] M.E. Stickel. A Prolog-like inference system for computing minimum-cost abductive explanations in natural-language interpretation. *Annals of Mathematics and Artificial Intelligence*, 4:89-106, 1991.

[Stickel, 1992] Mark E. Stickel. A Prolog technology theorem prover: a new exposition and implementation in Prolog. *Theoretical Computer Science*, 104(1):109–128, 1992.

[Taylor, 1996] Kerry Lea Taylor. *Autonomous Learning by Incremental Induction and Revision*. PhD Thesis, The Australian National University, 1996.

[Toni, 1995] F. Toni. *Abductive Logic Programming*. PhD thesis, Imperial College, 1995.

[Van Belleghem *et al.*, 1994] Kristof Van Belleghem, Marc Denecker, and Danny De Schreye. The Abductive Event Calculus as a General Framework for Temporal Databases. In *Proceedings of the International Conference on Temporal Logic*, 1994.

[van der Laag, 1995] Patrick van der Laag. *An analysis of refinement operators in inductive logic programming*. Ph. D. thesis, Erasmus Universiteit, Rotterdam, the Netherlands, 1995.

[Van Le, 1993] Tu Van Le. *Techniques of Prolog programming with implementation of logical negation and quantified goals*. John Wiley and Sons, Singapore, 1993.

[Wellner, 1999] Benjamin R. Wellner. *An Abductive-Inductive Learning Framework for Logic-Based Agents*. MSc Thesis, Imperial College of Science Technology and Medicine, 1999.

[Whewell, 1847] William Whewell. *The philosophy of the inductive sciences*. 2 ed. John W. Parker, London, 1847.

[Yamamoto, 1997] Akihiro Yamamoto. Representing inductive inference with SOLD-resolution. In P. Flach and A. Kakas editors, *Proceedings of the IJCAI'97 workshop on Abduction and Induction in AI*, Nagoya, Japan, 1997.