1

PM-1 3½"x4" PHOTOGRAPHIC MICROCOPY TARGET
NBS 1010a ANSI/ISO #2 EQUIVALENT

PRECISIONSM RESOLUTION TARGETS

National Library
of Canada

Acquisitions and
Bibliographic Services Branch

395 Wellington Street
Ottawa, Ontario
K1A 0N4

Bibliothèque nationale
du Canada

Direction des acquisitions et
des services bibliographiques

395, rue Wellington
Ottawa (Ontario)
K1A 0N4

Your file - Votre référence

Our file - Notre référence

NOTICE

The quality of this microform is heavily dependent upon the quality of the original thesis submitted for microfilming. Every effort has been made to ensure the highest quality of reproduction possible.

If pages are missing, contact the university which granted the degree.

Some pages may have indistinct print especially if the original pages were typed with a poor typewriter ribbon or if the university sent us an inferior photocopy.

Reproduction in full or in part of this microform is governed by the Canadian Copyright Act, R.S.C. 1970, c. C-30, and subsequent amendments.

AVIS

La qualité de cette microforme dépend grandement de la qualité de la thèse soumise au microfilmage. Nous avons tout fait pour assurer une qualité supérieure de reproduction.

S'il manque des pages, veuillez communiquer avec l'université qui a conféré le grade.

La qualité d'impression de certaines pages peut laisser à désirer, surtout si les pages originales ont été dactylographiées à l'aide d'un ruban usé ou si l'université nous a fait parvenir une photocopie de qualité inférieure.

La reproduction, même partielle, de cette microforme est soumise à la Loi canadienne sur le droit d'auteur, SRC 1970, c. C-30, et ses amendements subséquents.

UNSUPERVISED LEARNING
FROM A
GOAL-DRIVEN AGENT

by

Bertrand Pelletier, B.Sc., M.Sc.

A thesis submitted to
the Faculty of Graduate Studies and Research
in partial fulfillment of
the requirements for the degree of
Doctor of Philosophy

Department of Systems and Computer Engineering
Carleton University
Ottawa, Ontario
July 12, 1993

© copyright

1993, Bertrand Pelletier

National Library
of Canada

Acquisitions and
Bibliographic Services Branch

395 Wellington Street
Ottawa, Ontario
K1A 0N4

Bibliothèque nationale
du Canada

Direction des acquisitions et
des services bibliographiques

395, rue Wellington
Ottawa (Ontario)
K1A 0N4

Your file Votre référence

Our file Notre référence

The author has granted an irrevocable non-exclusive licence allowing the National Library of Canada to reproduce, loan, distribute or sell copies of his/her thesis by any means and in any form or format, making this thesis available to interested persons.

L'auteur a accordé une licence irrévocable et non exclusive permettant à la Bibliothèque nationale du Canada de reproduire, prêter, distribuer ou vendre des copies de sa thèse de quelque manière et sous quelque forme que ce soit pour mettre des exemplaires de cette thèse à la disposition des personnes intéressées.

The author retains ownership of the copyright in his/her thesis. Neither the thesis nor substantial extracts from it may be printed or otherwise reproduced without his/her permission.

L'auteur conserve la propriété du droit d'auteur qui protège sa thèse. Ni la thèse ni des extraits substantiels de celle-ci ne doivent être imprimés ou autrement reproduits sans son autorisation.

ISBN 0-315-92942-1

Canada

Name

Sutton, P.H.

Dissertation Abstracts International is arranged by broad, general subject categories. Please select the one subject which most nearly describes the content of your dissertation. Enter the corresponding four-digit code in the spaces provided.

Sutton, P.H.

SUBJECT TERM

C I R A

SUBJECT CODE

U·M·I

Subject Categories

THE HUMANITIES AND SOCIAL SCIENCES

COMMUNICATIONS AND THE ARTS

Architecture 0729
Art History 0377
Cinema 0900
Dance 0378
Fine Arts 0357
Information Science 0723
Journalism 0391
Library Science 0399
Mass Communications 0708
Music 0413
Speech Communication 0459
Theater 0465

EDUCATION

General 0515
Administration 0514
Adult and Continuing 0516
Agricultural 0517
Art 0273
Bilingual and Multicultural 0282
Business 0688
Community College 0275
Curriculum and Instruction 0727
Early Childhood 0518
Elementary 0524
Finance 0277
Guidance and Counseling 0519
Health 0680
Higher 0745
History of 0520
Home Economics 0278
Industrial 0521
Language and Literature 0279
Mathematics 0280
Music 0522
Philosophy of 0998
Physical 0523

Psychology 0525
Reading 0535
Religious 0527
Sciences 0714
Secondary 0533
Social Sciences 0534
Sociology of 0340
Special 0529
Teacher Training 0530
Technology 0710
Tests and Measurements 0288
Vocational 0747

LANGUAGE, LITERATURE AND LINGUISTICS

Language
General 0679
Ancient 0289
Linguistics 0290
Modern 0291
Literature
General 0401
Classical 0294
Comparative 0295
Medieval 0297
Modern 0298
African 0316
American 0591
Asian 0305
Canadian (English) 0352
Canadian (French) 0355
English 0593
Germanic 0311
Latin American 0312
Middle Eastern 0315
Romance 0313
Slavic and East European 0314

PHILOSOPHY, RELIGION AND THEOLOGY

Philosophy 0422
Religion
General 0318
Biblical Studies 0321
Clergy 0319
History of 0320
Philosophy of 0322
Theology 0469

SOCIAL SCIENCES

American Studies 0323
Anthropology
Archaeology 0324
Cultural 0326
Physical 0327
Business Administration
General 0310
Accounting 0272
Banking 0770
Management 0454
Marketing 0338
Canadian Studies 0385
Economics
General 0501
Agricultural 0503
Commerce Business 0505
Finance 0508
History 0509
Labor 0510
Theory 0511
Folklore 0358
Geography 0366
Gerontology 0351
History
General 0578

Ancient 0579
Medieval 0581
Modern 0582
Black 0328
African 0331
Asia, Australia and Oceania 0332
Canadian 0334
European 0335
Latin American 0336
Middle Eastern 0333
United States 0337
History of Science 0585
Law 0398
Political Science
General 0615
International Law and Relations 0616
Public Administration 0617
Recreation 0814
Social Work 0452
Sociology
General 0626
Criminology and Penology 0627
Demography 0938
Ethnic and Racial Studies 0631
Individual and Family Studies 0628
Industrial and Labor Relations 0629
Public and Social Welfare 0630
Social Structure and Development 0700
Theory and Methods 0344
Transportation 0709
Urban and Regional Planning 0999
Women's Studies 0453

THE SCIENCES AND ENGINEERING

BIOLOGICAL SCIENCES

Agriculture
General 0473
Agronomy 0285
Animal Culture and Nutrition 0475
Animal Pathology 0476
Food Science and Technology 0359
Forestry and Wildlife 0478
Plant Culture 0479
Plant Pathology 0480
Plant Physiology 0817
Range Management 0777
Wood Technology 0746
Biology
General 0306
Anatomy 0287
Biostatistics 0308
Botany 0309
Cell 0379
Ecology 0329
Entomology 0353
Genetics 0369
Limnology 0793
Microbiology 0410
Molecular 0307
Neuroscience 0317
Oceanography 0416
Physiology 0433
Radiation 0821
Veterinary Science 0778
Zoology 0472
Biophysics
General 0786
Medical 0760

EARTH SCIENCES

Biogeochemistry 0425
Geochemistry 0996

Geodesy 0370
Geology 0372
Geophysics 0373
Hydrology 0388
Mineralogy 0411
Paleobotany 0345
Paleoecology 0426
Paleontology 0418
Paleozoology 0985
Palynology 0427
Physical Geography 0368
Physical Oceanography 0415

HEALTH AND ENVIRONMENTAL SCIENCES

Environmental Sciences 0768
Health Sciences
General 0566
Audiology 0300
Chemotherapy 0992
Dentistry 0567
Education 0350
Hospital Management 0769
Human Development 0758
Immunology 0982
Medicine and Surgery 0564
Mental Health 0347
Nursing 0569
Nutrition 0570
Obstetrics and Gynecology 0380
Occupational Health and Therapy 0354
Ophthalmology 0381
Pathology 0571
Pharmacology 0419
Pharmacy 0572
Physical Therapy 0382
Public Health 0573
Radiology 0574
Recreation 0575

Speech Pathology 0460
Toxicology 0383
Home Economics 0386

PHYSICAL SCIENCES

Pure Sciences
Chemistry
General 0485
Agricultural 0749
Analytical 0486
Biochemistry 0487
Inorganic 0488
Nuclear 0738
Organic 0490
Pharmaceutical 0491
Physical 0494
Polymer 0495
Radiation 0754
Mathematics 0405
Physics
General 0605
Acoustics 0986
Astronomy and Astrophysics 0606
Atmospheric Science 0608
Atomic 0748
Electronics and Electricity 0607
Elementary Particles and High Energy 0798
Fluid and Plasma 0759
Molecular 0609
Nuclear 0610
Optics 0752
Radiation 0756
Solid State 0611
Statistics 0463

Applied Sciences

Applied Mechanics 0346
Computer Science 0984

Engineering
General 0537
Aerospace 0538
Agricultural 0539
Automotive 0540
Biomedical 0541
Chemical 0542
Civil 0543
Electronics and Electrical 0544
Heat and Thermodynamics 0348
Hydraulic 0545
Industrial 0546
Marine 0547
Materials Science 0794
Mechanical 0548
Metallurgy 0743
Mining 0551
Nuclear 0552
Packaging 0549
Petroleum 0765
Sanitary and Municipal System Science 0790
Geotechnology 0428
Operations Research 0796
Plastics Technology 0795
Textile Technology 0994

PSYCHOLOGY

General 0621
Behavioral 0384
Clinical 0622
Developmental 0620
Experimental 0623
Industrial 0624
Personality 0625
Physiological 0989
Psychobiology 0349
Psychometrics 0632
Social 0451

The undersigned hereby recommend to
the Faculty of Graduate Studies and Research
acceptance of the thesis,

"Unsupervised Learning From a Goal-Driven Agent"

submitted by

Bertrand Pelletier, B.Sc., M.Sc.

in partial fulfillment of the requirements
for the degree of Doctor of Philosophy

Chair, Department of Systems and Computer Engineering

Thesis Supervisor

Thesis Supervisor

External Examiner

Carleton University

November 11, 1993

ABSTRACT

Knowledge acquisition, the process of incorporating expertise into computer systems, is one of the most difficult and challenging problems in machine learning. One way to address the problem is to extract the desired information by analyzing the behaviour of human experts while they are at their best, that is to say, while they solve problems in their domains of expertise – in contrast with situations where they try to explain how to solve these problems.

One approach to the knowledge acquisition problem consists in recording – without any perturbations – expert behaviours in order to extract pertinent information from them.

This thesis describes an approach and an implemented system to realize this form of unsupervised and non-obtrusive learning. The approach exploits techniques and representations from machine learning and planning to address the problem.

Starting from an incomplete theory of goals and plans used by the experts to solve problems, and using several instances of expert behaviours not fully explained by this theory, missing knowledge is inferred to augment the theory, making it adequate to explain a larger part of the behaviours.

The learning methods proposed in the thesis do the following: determine the goal structure used by the expert, that is to say, the decomposition of goals into subgoals, down to the level of primitive actions, infer candidate pertinent goals achieved by the expert that are left unexplained by the current theory, identify and generalize the plans used by the expert to achieve the goals, and determine the precondition and the postcondition and the generalized plans.

This type of learning is appropriate to acquire knowledge from designers using software tools (in databases and integrated circuits, for instance), and from planners achieving goals under given constraints such as in robotics or experimentation.

The learning methods were applied to two domains – entity-relationship model design and office tasks – where favorable empirical results have been obtained and are described.

ACKNOWLEDGEMENTS

I would like to thank my supervisors Bernard Pagurek and Stan Matwin for their guidance and support in the realization of this work. I would like to address special thanks to Stan for his support on both personal and professional levels.

The Department of Systems and Computer Engineering provided facilities for the early stage of the work. The Department of Computer Science of Ottawa University provided facilities for its kernel. In particular, I would like to thank Michel Racine and Jacques Sincennes for their help with computer hardware and software, and Elise Amyot for her help with the statistical analysis.

The Department of Systems and Computer Engineering, the Department of Computer Science of Ottawa University and NSERC provided financial assistance for the work.

Part of the work was done over the AMOS Project; thanks to Cognos Inc. for its contribution, along with Franz Oppacher, Jean Genest, Zbig Koperzak and Una-May O'Reilly.

The final and most grateful thanks go to my family and friends whose support and confidence were more than welcome more than once.

TABLE OF CONTENTS

	Page
Abstract	iv
Acknowledgements	v
Table of Contents	vi
List of Tables	ix
List of Figures	x
List of Examples	xii
List of Appendices	xiii
1 Introduction	1
1.1 Presentation of the problem	1
1.2 Summary of contributions	13
1.3 Organization of the thesis	14
2 An environment for learning goal-oriented plans	15
2.1 General concepts in planning and modelling	16
2.2 Goals, interactions and hierarchical plans	22
2.3 Semantics of preconditions and postconditions	27
2.4 Definition of the semantics of hierarchical plans	37
2.4.1 Tasks' semantics is used for two different purposes	38
2.4.2 Negated literals can be avoided inside DEL/ADD lists	40
2.4.3 Inferential literals are useless inside DEL/ADD lists	41
2.4.4 Variable sharing between DEL/ADD lists is useful and dangerous	47
2.4.5 Common literals between DEL/ADD lists are useful and dangerous	54
2.4.6 Redundant literals inside DEL/ADD lists are useful	57
2.4.7 What is the meaning of "applying DEL/ADD lists to a state"	59
2.5 Explaining interactions	62
2.5.1 Goal graphs describe the structure of goals	65
2.5.2 Causal links determine contributors to the goals	68

3 Learning from agent interactions	72
3.1 Description of LEADER	72
3.1.1 The Interaction Observer	73
3.1.2 The Action Executor	74
3.1.3 The Interaction Analyzer	76
3.2 Learner: the learning component of LEADER	79
3.2.1 Identifying a candidate goal	80
3.2.2 Identifying the hierarchical plan realizing a goal	88
3.2.3 Determining the semantics of a hierarchical plan	93
3.2.3.1 The non-inferential propositional case	94
3.2.3.2 The inferential predicate calculus case	103
3.2.4 Constructing the resulting goal rule	122
3.3 Summary of the approach	127
 4 Experimentation with LEADER	 129
4.1 Experimental protocol	130
4.1.1 Choice of a testing domain	130
4.1.2 Choice of the training interactions	134
4.1.3 Choice of the testing interactions	137
4.1.4 Choice of the evaluation criteria	137
4.2 Application to the Office Tasks domain	139
4.2.1 The Office Tasks domain	139
4.2.2 Experimentation with the Office Tasks domain	143
4.2.3 Results for the Office Tasks domain	145
4.3 Application to an expert system for modelling	152
4.3.1 Description of the Modeller	153
4.3.2 Experimentation with Modeller	157

5 Comparison with related work	166
5.1 Comparison with program analysis techniques	166
5.2 Comparison with planners	168
5.2.1 Representation of goals and states	169
5.2.2 Representation of time	170
5.2.3 Representation of actions and plans	172
5.2.4 Plan construction	175
5.2.5 Plan verification	176
5.3 Comparison with learning apprentice systems	178
5.3.1 Nature of the initial domain theory	179
5.3.2 Nature of the interaction between the agent and the system	181
5.3.3 Nature of the knowledge used by the system	183
5.3.4 Methods used to acquire this knowledge	185
5.4 Concluding remarks	187
6 Conclusion	189
6.1 Conclusions	189
6.2 Future research	191
References	194
Appendices	198

LIST OF TABLES

Table	Description	Page
3-1	Programming concepts vs concepts used in LEADER	95
3-2	Available expressions for a program	96
4-1	Actions of the Office Tasks domain	140
4-2	General concepts vs concepts used in design	155
4-3	List of commands available to describe the designer interaction	156
5-1	Parameters for assessing the quality of a domain theory	180
5-2	Methods for transferring new tasks	186

LIST OF FIGURES

Figure	Description	Page
1-1	Representation of the approach	7
1-2	Architecture of the learning system	10
2-1	An interaction with its corresponding states	24
2-2	Partial relationships between hierarchical plan expansions	35
2-3	Complete relationships between hierarchical plan expansions	36
2-4	Goal graph built over an interaction	65
2-5	Invalid explanation built over an interaction	67
2-6	A goal graph with causal links	70
3-1	Architecture of LEADER	73
3-2	Illustration of the general algorithm	83
3-3	Illustration of Learning Heuristic 1	85
3-4	Illustration of Learning Heuristic 2	87
3-5	An establisher-clobberer pair of actions	91
3-6	Algorithm for computing the <i>in</i> and <i>out</i> sets of available expressions	98
3-7	Axiomatic theory used to illustrate the two approaches	106
3-8	An initial condition P and a task T	106
3-9	Global effect of a task T	107
3-10	Graphical representation of Approach 1	108
3-11	Graphical representation of Approach 2	110
4-1	Candidate artificial domains	133
4-2	The "redial_phone" action	141
4-3	A goal and related goal rules	141
4-4	An initial state for the Office Tasks domain	142
4-5	The second initial state for the Office Tasks domain	144
4-6	Goal rules produced by LEADER	146
4-7	Average number of rules produced by LEADER	147
4-8	Relative gain in coverage of interactions	148
4-9	Relative coverage of interactions for the first initial state	149
4-10	Variety (%) of actions used	150

4-11	Relative overage of interactions	151
4-12	Relative gain of coverage	152
4-13	Architecture of the Modeller	154
4-14	Semantics of some Modeller's actions	156
4-15	Part of the initial domain theory for Modeller	158
4-16	Definition of some Modeller's inferential literals	159
4-17	Graphical representation of a goal for Modeller	159
4-18	An interaction and its resulting state	160
4-19	Resulting states in internal format	161
4-20	Goal graph for the interaction of Figure 4-21	162
4-21	Proof trees for P_0 and Q_0	163
4-22	Goal rule produced from the interaction of Figure 4-18	163
4-23	Goal graph after learning for the interaction of Figure 4-18	164

LIST OF EXAMPLES

Example	Description	Page
2-1	Several hierarchical plans achieving the same goal	31
2-2	Expansions of a hierarchical plan	32
2-3	Need for inferential literals in tasks' semantics	42
2-4	Need for inferential literals inside the postcondition	45
3-1	Computing the precondition of a task	115
3-2	Computing the precondition of a non-executable task	116

LIST OF APPENDICES

Appendix	Description	Page
A	Terminology	198
B	Domain theory for actions of the Office Tasks domain	202
C	Domain theory for goals and plans of the Office Tasks domain	203
D	Domain theory for implemented actions of the Modeller domain	205
E	Domain theory for implemented goals and plans of the Modeller domain	206
F	Example of a session with LEADER	207

CHAPTER 1

Introduction

1.1 Presentation of the problem

Acquiring knowledge from human experts to incorporate it into computer systems is the bottleneck of knowledge acquisition and is a great challenge for machine learning. One way of addressing the knowledge acquisition problem is to obtain the knowledge automatically by non-obtrusive observations of a user interacting with a computer system. To extract from the entire record of the interaction its crucial parts, the knowledge acquisition system must be capable of at least partial comprehension of the interaction and its relationship to the goals being achieved by the user.

One of the existing methods to achieve such comprehension is to use explanation-based learning techniques which explain how the interaction realizes a class of similar goals in terms of an underlying domain theory.

This thesis addresses the knowledge acquisition problem by proposing a learning system that enhances a given domain theory about goals and plans. The system analyzes in an unsupervised manner the behaviour of users – preferably experts – that solve problems in the given domain.

There are many reasons to acquire knowledge from experts, for instance, to store and preserve knowledge and the heuristics learned by experts through years of experience, to duplicate that knowledge to create intelligent systems far more rapidly and easily than human expertise can be reproduced, and to build systems to assist human experts or to solve problems in situations where no human expert is available (for instance, in the developing regions of the world or in space exploration).

The automation of this transfer has several advantages.

First, even if a human expert is available for a given domain, it might be difficult for him or her to formalize his or her knowledge in terms of the data structures used by the system. For instance, an expert usually prefers to talk in terms of specific cases rather than to derive general deduction rules. Automating the learning (or semi-automating it with the help of a knowledge engineer) would ease the acquisition of information and increase the rate at which it is gained.

Second, it might be difficult for the expert to determine what knowledge must be used as the *kernel knowledge*, that is to say, the knowledge that is sufficient for the system to perform the intended task. So, the system may fail to perform its task, because its initial knowledge is incomplete, incorrect, or even inconsistent. In these cases, automated tools should be provided to the expert to refine the existing knowledge by generating new knowledge, testing the knowledge and modifying it.

Third, research done to automate the learning process provides greater insight into human behaviour, and allows the construction of systems that behave more intelligently when interacting with human users.

When addressing problems related to the automation of knowledge acquisition, one may list four classes of problems [Levi, Perschbacher et al. 1988]:

- construction of an initial knowledge base
- improvement of the existing knowledge
- automatic adaptation of the system to the style and to the level of expertise of the human user

- elaboration of principles and techniques for building intelligently behaving systems.

Each of these classes contains numerous difficult problems. The present thesis addresses problems of the second class, and indirectly, problems of the third and fourth classes. We present a learning system that acquires its knowledge without human supervision, and that does not depend on a particular domain. So, the first class of problems is not of interest to us because these problems usually heavily depend on the domain of application. The thesis addresses problems of the third class in that the learning system is able to understand (that is to say, explain) interactions from users having various styles or following different methodologies, and is also able to learn from these interactions. This class of problems is only partially addressed by the thesis because the system presented here does not learn by interacting with a human: learning occurs by analyzing off-line the interactions produced by the human.

The second class of problems, the improvement of the information used by a knowledge-based system, is known as the *knowledge base refinement problem*, a special case of the *theory revision problem* [Ginsberg 1989]:

"A theory revision problem exists for a theory T when T is known to yield incorrect results for given cases in its intended domain of application. The goal of theory revision is to find a revision T' of T which handles all known cases correctly, makes use of the theoretical terms used in T, and may, with a reasonable degree of confidence, be expected to handle future cases correctly. In contrast to pure inductive learning from experience, theory revision is not only guided by the information implicit in T, but also

attempts to preserve the language and, as much as possible, the structure of T."

The theory revision problem can be separated into three sequential sub-problems:

- detection of a deficiency such as¹
 - incorrectness: the domain theory misinterprets observations, or predicts non-existent observations
 - inconsistency: the domain theory provides inconsistent interpretations or predictions
 - incompleteness: the knowledge is insufficient to cover observations (missing rules, insufficient vocabulary...)
 - inefficiency: the presentation of the knowledge is inadequate for its intended use (information redundancy, inefficient ordering of rules, structure too flat or too deep...)
- suggestion of a repair for the deficiency
- validation of the repair (if several valid repairs are proposed, only the best one must be applied).

Existing learning systems deal with these problems with various degrees. For instance, ODYSSEUS [Wilkins 1990], a learning apprentice system designed to refine a domain theory, addresses the problems of incorrectness, inconsistency and incompleteness in an

¹ The following list of deficiencies take into account types of deficiencies not considered in Ginsberg's definition, such as information redundancy and inappropriate knowledge structure.

expert system knowledge base. The system detects a problem, then proposes and validates the repair. The validation is accomplished using additional underlying knowledge (assumed to be correct) about the domain of application.

Explanation-based learning (EBL) systems [DeJong and Mooney 1986] address the problem of inefficiency. They construct rules to explain the observations in a more efficient manner by flattening the knowledge structure; here, the efficiency is defined by the *operationality criterion*, indicating for instance which parameters are directly observable. Because the knowledge to be refined is correct and consistent, the proposed rules are also correct and consistent; no validation is thus required.

The representation language (classification rules, production rules, frames, plans, etc.), the nature of the information provided to the learning system and the nature of the knowledge to be refined have a direct impact on the detection, suggestion and validation tasks. For instance, in a rule-based system where the rules and facts possess a *belief factor* (degree of certainty), giving an inference tree and a set of cases (a case being a set of premises and deduced facts), determining the belief of the rules (the *synthesis problem*, a special case of the *knowledge base refinement problem*) can be NP-complete, NP-hard, or even solvable in polynomial time, depending on certain parametrizable conditions [Ling and Valtorta 1991].

The learning system we propose here is to acquire the plans used by some agent (typically a human expert) solving problems in a given domain, or more generally, simply interacting with the domain. The representation language for the knowledge base consists, among other things, of goals, plans, preconditions and postconditions. The initial knowledge base is assumed to be correct and consistent; in particular, each plan is guaranteed to succeed when executed in a state where its precondition is satisfied. So, the learning system focuses

primarily on the problems of incompleteness and inefficiency: acquiring new plans to perform tasks. However, because acquired plans can be overgeneralized, correctness and consistency are also of concern.

The approach proposed is pictured in Figure 1-1. It is assumed the agent interacts with a given world containing objects. Also, it is assumed that the agent has some goals to achieve. To do so, the agent uses a goal structure (by decomposing goals into subgoals) to select and apply on objects appropriate actions among those available (according to the leaves of the goal structure).

The learning system, called LEADER (for LEARNING DEsign Rules), observes the agent interaction (i.e., the ordered sequence of actions) in a non-obtrusive way and tries to explain the interaction using an incomplete domain theory about goals and plans in that domain. When no explanation can be found for some part of the interaction, LEADER tries to infer new knowledge to make the explanation possible. This is where the learning occurs: LEADER makes the domain theory more complete by identifying gaps in it and by filling them.

Figure 1-1. Representation of the approach. The learning system *LEADER* (LEARNING DESIGN RULES) observes, explains and learns from the actions performed by some agent.

Now, let us overview the architecture of the proposed system. Its learning capabilities rely on the following assumptions:

- the agent's behaviour is driven by goals
- the agent creates and executes hierarchical plans to achieve desired goals
- the system learns without interacting with the agent.

These assumptions are developed in the following.

Because the agent's behaviour is driven by goals that are later on translated into actions via hierarchical plans, the system can determine this goal structure specifying, for instance, how high-level goals are translated into lower level goals, how low-level goals are realized by operational commands and actions, and what are the precondition and postcondition of the plans used by the agent.

Using an incomplete theory of goals and hierarchical plans, as well as several instances of agent interactions not fully explained by this theory, the missing knowledge can be inferred to augment the theory, making it complete enough to explain a larger part of the interactions; the focus is thus on completeness.

Because the learning system acquires its knowledge by observing and recording only the agent's interactions, there are no interference with the agent during the interaction with the given domain. This contrasts with many learning systems, such as ARMS [Segre 1988], a system for robot control that requires that the teacher guides the robot arm through a series of movements that achieve a given goal, CLERK [Campbell 1990], a system that learns tasks accomplished by an electronic-mail system user by recording the task and the corresponding sequence of actions taught by the user, or LEAP [Mitchell and Mabadevan 1990], a system designed to acquire new knowledge in VLSI by observing design steps.

The result of learning is accumulated and later on presented to a human expert for verification, possible additional refinement, and incorporation into the existing knowledge.

For this type of learning from instances, we propose a framework based on the explanation-based learning (EBL) paradigm. This choice is influenced by the nature of the task involved: in order to learn, the interactions must be understood (at least partially), the relevant parts have to be identified, and the irrelevant ones discarded [DeJong and Mooney

1986] Also, EBL systems offer a good approach in the present learning context for additional reasons, as pointed out by [Levi, et al. 1988]:

"The most appropriate learning-system technologies for these applications [tactical planning and information requirements] were deductive learning systems such as EBL. The following conclusions supported our recommendation:

- EBL systems are the most capable of learning knowledge about procedures
- EBL systems are preferable given an adequate domain theory
- EBL systems make the best use of single positive learning experiences."

The thesis presents a framework as well as an implemented system that performs unsupervised learning, given a kernel knowledge for a given domain, along with some instances of interactions of a goal-driven agent with this domain. This unsupervised learning approach is appropriate for acquiring knowledge in any domain where an agent uses strings of commands to achieve goals. Examples of such circumstances are designers using software tools (such as for databases and integrated circuits), and planners achieving goals under given constraints (such as in robotics and experimentation).

Figure 1-2 presents the whole system.

Figure 1-2. Architecture of the learning system. Ovals represent external sources of information, parallelograms represent processes, rectangles represent data used or produced by processes, and arrows represent data flow. The broken line box represents *LEADER* (LEARNING DESIGN RULES), the component where all the learning is performed (described in detail in Chapter 3).

The *Interaction Observer* (described in Chapter 3) watches the agent interaction, with no disturbance, and produces the ordered sequence of actions corresponding to that interaction. Then, the *LEADER* system (also described in Chapter 3) analyzes the interaction, builds explanations for it, and produces new candidate knowledge to explain parts of the interaction that are left unexplained. As the figure shows, *LEADER* may use the approved existing knowledge as well as the candidate knowledge proposed so far to guide its learning.

The whole process is applied to different agent interactions (and possibly to different agents as well) and the resulting candidate knowledge for each of these interactions is accumulated.

After the system has been functioning for a while, a human expert in the world being modelled examines the accumulated proposed knowledge to verify it and add the filtered knowledge to the current knowledge about the world being modelled. The task of the expert is to eliminate incorrect or redundant information, and to refine the correct information, merging it with the already approved knowledge. This merging may further eliminate redundancies, refine knowledge, and even repair any incorrect information not previously detected.

Some existing learning systems, such as systems based on planning (STRIPS [Fikes and Nilsson 1971], TWEAK [Chapman 1987], PRIAR [Kambhampati and Hendler 1990]), learning apprentice systems (ARMS [Segre 1988], CAP [Dent, Boticario et al. 1992], CLERK [Campbell 1990], DISCIPLE [Tecuci and Kodratoff 1990], LEAP [Mitchell and Mabadevan 1990], ODYSSEUS [Wilkins 1990]), and systems learning classifications (AQ15 [Michalski, Mozetic et al. 1986], C4.5 [Quinlan 1993], CN2 [Clark and Niblett 1989], FOIL [Quinlan 1990]) address similar problems to the ones involved in LEADER. Their differences and similarities with LEADER are presented in detail in Chapter 5; the particularities of LEADER are summarized here:

- using examples of interactions, the system identifies the goals and it learns the plans that realize them. In typical systems, it is assumed that some teacher will provide the goals, one at a time.

For instance, in planning and learning apprentice systems, the learning system usually knows the goal pursued, even when provided only by a functional specification. In systems learning classification rules (where the class corresponds to the goal, the class description to a plan and the instance to a portion of the agent interaction), the intention is to learn the description

of a class, giving several instances with their classification. In LEADER, the class (i.e., the classification of the instances) is identified by the system, and the description of the class is learned, giving the instances.

- the instances are not delimited. In LEADER, an instance corresponds to a portion of an interaction, and it is the burden of the system to determine where the instance begins and where it ends. In particular, it has to find out how far back in time agent actions must be traced to derive explanations.
- the learning system does not rely on an abundance of instances. In some cases, only one instance might be available.
- the order of execution of agent actions and goals is important (it is not simply a conjunction of goals, as in typical systems that learn classification rules).
- the system may learn hierarchical plans as well as flat plans, and determine their precondition and postcondition.
- the initial domain theory is assumed to be incomplete. In contrast with typical EBL systems, in LEADER, learning occurs when the theory is not complete enough to explain the instances.
- finally, no information (such as instance classification, relevant features, reason of failure, judgement of repairs) is provided by some teacher during the learning. In fact, the learning system operates in a totally transparent manner with respect to the agent. In other words, all the learning is unsupervised.

1.2 Summary of contributions

The contributions of the present work are the following:

- an operational system implemented in Prolog capable of learning plans by analyzing agent's interactions in an unsupervised manner, in various domains of application ([Matwin, Oppacher et al. 1991] and Chapter 3)
- an appropriate representation for precondition and postcondition of hierarchical plans, along with an efficient algorithm for their computation, when conditions are restricted to the propositional case and no inferential literals are allowed ([Pelletier and Matwin 1992] and Section 3.2.3.1)
- for more complex situations, heuristic methods to compute precondition and postcondition for hierarchical plans (Section 3.2.3.2)
- the analysis of the performance of the system in two different domains: the execution of office tasks and the design of entity-relationship models.

The application to the first domain shows that LEADER can be used to enhance significantly a domain theory: it can increase the coverage of the theory (the number of situations taken into account by the theory) by approximately 10%.

The application to the second domain shows that LEADER can produce useful enhancements to the domain theory of a practical domain.

1.3 Organization of the thesis

The remainder of the thesis is organized as follows:

Chapter 2 presents concepts related to the context of goal-driven agents and used in LEADER. Standard concepts are presented, and new concepts are introduced.

Chapter 3 describes how the learning by observing the agent is done. In particular, it describes how LEADER identifies the goal pursued by the human expert and the hierarchical plan used, and how it builds proposals for enhancement.

Chapter 4 describes the experimental protocol used to evaluate LEADER and presents two examples of application. In the first example, the system is applied to the office tasks domain. The purpose is to show that LEADER can be used to enhance significantly a domain theory. In the second example, LEADER is used to enhance an expert system in database design by observing an expert designer at work. For this example, the purpose is to show that the system can propose useful enhancements. In both cases, learning is realized by analyzing the interaction, understanding goals and methods used, and inferring missing knowledge to make the domain theory more complete.

Chapter 5 establishes comparisons with LEADER and other systems such as program analyzers, planners and learning apprentice systems.

Chapter 6 presents the conclusions and indicates directions for future research.

Appendix A provides a glossary of main terms used in the thesis. Appendices B and C describe the office tasks domain. Appendices D and E describe the database design domain. Appendix F provides a complete example of a session with LEADER (the session is described in Section 4.3.2).

CHAPTER 2

An environment for learning goal-oriented plans

The specification of an environment for learning the goals and the plans used by a goal-driven agent requires the definition of appropriate concepts and the development of particular methods based on these concepts. Both the concepts and the methods depend on the learning task.

In the present context, the learning task considered is to infer new knowledge to explain a larger part of a partially explained interaction. More precisely, this task requires

- the identification of the goal being pursued;
- the determination of the hierarchical plan that contributes to the realization of this goal;
- the computation of the precondition and the postcondition of this hierarchical plan.

This chapter presents concepts and methods needed for learning goal-oriented plans according to the proposed approach, such as the notions of goal, interaction, hierarchical plan, precondition and postcondition. Section 2.1 presents basic concepts commonly used in the literature, and following sections describe adaptations and extensions made to solve the problem. Sections 2.2, 2.3 and 2.4 define the notions of goals and plans as used in the thesis. Section 2.5 introduces the concepts needed to build explanations for the agent's behaviour.

2.1 General concepts in planning and modelling

This section introduces concepts, representations and definitions inspired from standard terminology. For a discussion on the differences, see Chapter 5.

Assume that some *agent* interacts with a *world*² containing objects. For instance, the agent could be a robot and the world could consist of objects involved in an industrial assembly line process (the objects being built, the mechanism driving the assembly line, etc.). A representation is used to describe the following aspects of the world: which properties are true in a given state of the world, how the agent modifies the state, and how properties change when the state changes. Modelling is used to represent the world, the agent and the interactions.

An appropriate *context* [Guha 1990] is required to model this world, that is, the assumptions and approximations about the world that are sufficient for the representation to be correct according to the intentions of the representation. For instance, a fact stating that an object is in contact with another object has a meaning depending on the granularity (context) of the required description: either Newtonian mechanics or quantum mechanics.

In this work, we assume that the context is fixed. Consequently, there is no notion of assumption changes and context switching [Guha 1990]. For instance, if the world is about experiments in classical mechanics, then Newtonian mechanics would be the proper choice for the context (according to current scientific knowledge), rather than Einsteinian relativity principles or even a mix of both.

² "World" has several meanings in the literature. For instance, [Filman 1989] (p.108) defines it as a "set of related facts – for example, a situation, a simulation checkpoint, a belief set, or a hypothetical state of a problem solver". In our terminology (introduced next), this corresponds to a context, a world state and axioms.

Now, let us introduce some basic definitions about the representation we use for modelling.

Definition:

A *predicate* is composed of two parts: a name (called functor) and possibly some ordered arguments.

For instance, using the domain of entity-relationship model design, the predicate *related(A,B)* has the functor *related*, and arguments A and B corresponding to entities.

Definition:

A *literal* is a predicate (positive literal), or the negation of a predicate (negative literal). The negation is denoted by the connective " $\neg$ ".

Definition:

A *state* describes a world situation. It is represented by a set of fully instantiated literals. The empty state is denoted S^e , that is to say, $S^e = \{ \}$.

Some examples are:

positive literals: *entity(a)*, *entity(B)*.

negative literals: $\neg$ *entity(c)*, $\neg$ *related(b,_)*.

state: {*entity(a)*, *entity(b)*, *relation(a,b,role)*}.

Definition:

A *property* is a conjunction of literals. The conjunction is represented by the connective " $\wedge$ ".

For our purposes, a conjunction of literals may also be represented as the set of these literals. For instance, the one-element set $\{p\}$ represents the literal p , and $\{p,q,r,s\}$ represents the property $(p \wedge q \wedge r \wedge s)$. The empty set stands for the truth value *true*.

Definition:

The agent modifies the state via *commands*. A command C is represented by $C = \langle \text{precond}(C), \text{DEL}(C), \text{ADD}(C), \text{postcond}(C) \rangle$, where C specifies the name of the command and its arguments, where $\text{precond}(C)$ (the precondition of C) specifies the *weakest* condition that must hold in the current state for C to be executable, and where $\text{postcond}(C)$ (the postcondition of C) specifies the *strongest* condition obtained in the state after the execution of C . The components $\text{DEL}(C)$ and $\text{ADD}(C)$ provide, respectively, the literals to be deleted or added to the current state to produce the state resulting from the execution of the command.

The term *weakest* is used to characterize the precondition in order to discard any condition that is irrelevant to C . The terms *weakest* and *strongest* will take their full meaning when we discuss the precondition and the postcondition of hierarchical plans (Section 2.3).

The DEL/ADD mechanism required to compute the state obtained by the application of a command uses two lists of literals, as in STRIPS-like systems [Fikes and Nilsson 1971]: a list of literals to delete from the state (called the *DEL list*), and a list of literals to add (called the *ADD list*). If the two lists have no literals in common, one can do either the deletions or the additions first. Otherwise, it is assumed that the deletions are done first.

A command is a general specification of how the agent modifies the current state. So, its arguments are variables that are instantiated to appropriate state objects before the agent executes it.

Definition:

An *action* is a command whose arguments are fully instantiated.

To illustrate these concepts, let us take two examples: robot planning and database design.

In robot planning, the agent is usually a robot (one may consider machines such as a saw or a press as other agents). The world consists of objects manipulated by the robot, and agent actions consist mainly of moving objects (possibly including the robot itself). One of these actions might be *put box1 on block2* (put the box 1 on the block 2), which is a full instantiation of the command *put A on B*.³ One of the effects of this action is to achieve the following property in the resulting world state: *on(box1,block2)* (the box 1 is currently placed on the block 2).

In database design, the agent is usually a human expert. The world consists of objects manipulated by the expert, for instance, entities and relationships. Agent actions consist mainly of creating or deleting world objects and establishing new relationships among them.

There are two levels of modelling at work here. At the first level, a particular *enterprise* has to be *modelled* using a given *representation* (e.g., using entity-relationship representation). As no agent (in the sense previously described)⁴ is germane, the *enterprise* being *modelled* is not what was defined as a world.

³ Arguments beginning with a lowercase letter represent instances of objects; those beginning with an uppercase one represent variables.

⁴ If "enterprise agents", or in general, "enterprise processes", have to be "modelled", then these "agents" will become objects manipulated by the modelling expert. Consequently, the real agent manipulating the world will be the human expert interacting with a representation that allows these processes to be represented.

At the second level, the agent manipulating this world is the human expert interacting with a representation to model the first-level *world*.

Because the second level of modelling is of concern here, an example of agent actions may be *addE(student)* (according to the entity-relationship representation, create an entity called *student*).

This action is an instantiation of the command *addE(E)*. This command is described by $addE(A) = \langle \neg entity(A), \{ \}^5, \{ entity(A) \}, entity(A) \rangle$. When used with the substitution $\{ A/c \}$, it leads to the action $addE(c) = \langle \neg entity(c), \{ \}, \{ entity(c) \}, entity(c) \rangle$. When applied to the initial state

$$\{ entity(a), entity(b), relation(a,b,role) \}^6,$$

the resulting state is

$$\{ entity(a), entity(b), entity(c), relation(a,b,role) \}.$$

For simple cases, the DEL/ADD mechanism is sufficient to specify how the state changes when commands are executed. In real situations, the deletion or the addition of a literal in the state may have repercussions over many other literals. Also, one may be interested in a property that cannot be obtained by the execution of a single command. In these cases, the DEL/ADD mechanism either leads to large lists, making it inappropriate for a good understanding of what happens, or is inapplicable. It is then required to have a mechanism

⁵ As we will see later on, DEL/ADD lists do not contain negated literals, and the presence of a positive literal in the ADD list indicates that the corresponding negated literal (if any) will be retracted from the precondition; this is why the DEL list is empty here.

⁶ The closed-world assumption is used to verify that the fact $\neg entity(c)$ holds in the state, and thus, that the action is executable.

that specifies how some properties are modified as a consequence of other properties being modified.

Definition:

An *axiom* is an implication formula on the form $L \leq L_1 \wedge L_2 \wedge \dots \wedge L_n$, where L (called the *consequent*) is a positive literal, and L_i 's (called the *antecedents*) are positive or negative literals. The current set of axioms is denoted AXMS.

Thus, an axiom is a rule providing the definitions for literals that are inferred from other literals. For the axiom $related(A,B) \leq relation(A,B,R)$, for instance, the consequent is $related(A,B)$, the antecedent is $relation(A,B,R)$, and the axiom tells that if it is true that "there exists a relationship R between entities A and B ", then it is also true that "entities A and B are related". The set of axioms corresponds to what is called the *domain theory* in explanation-based systems [Mitchell, Keller et al. 1986]; in this case, it is the domain theory for properties.

The notion of axioms divides literals into two disjoint sets, as follows:

Definition:

A literal is called *inferential* if it has the same functor and the same number of arguments as the consequent of some axiom. Otherwise, it is called *primitive*.

For instance, if the current set of axioms contains only the above axiom, then $related(A,B)$ and $related(student, course)$ are inferential literals, whereas $relation(A,B,R)$ and $relation(student, course, related)$ are primitive ones.

This definition allows literals that appear in no axioms. According to the definition, such literals are primitive.

According to [Hammond 1989], properties (called *features*) can be divided into two disjoint groups: *observable properties* and *testable properties*. An observable property can be verified directly, without a test procedure (by accessing a database, observing the world or testing the membership in a set). In EBL terminology [Mitchell, et al. 1986], the verification of an observable property is said to be *operational*. On the other hand, the verification of a testable property requires a test procedure that has a cost. Thus, such a property is said to be non-operational. A state provides only observable properties, and testable properties are computed on request.

In some systems, the agent has a limited attention: only a subset of the world can be observed, depending of the agent's state. Here, we assume that the agent has an unlimited attention for all primitive properties, and that a primitive property never becomes an inferential property upon the agent actions, nor an inferential property becomes a primitive property.

Although disjunctions are not allowed inside properties or axioms (only conjunctions are), inferential literals can be used to express them: to represent the disjunction $p_1 \vee p_2$, a new inferential literal p can be created that replaces the disjunction, using the new axioms $p \leq p_1$ and $p \leq p_2$.

2.2 Goals, interactions and hierarchical plans

This section presents adaptations and extensions to notions commonly used in the planning literature. In particular, the following notions are described: goals (the properties the agent wants to achieve while interacting with the world), actions (the primitive operators available to the agent to achieve goals), tasks (the high-level operators available), interactions (the sequences of actions executed), achievement vectors (characterizations of the interactions in

terms of the feasibility and the achievement of goals), goal rule and hierarchical plan (the hierarchy of agent's goals and subgoals, along with the corresponding plans and sub-plans). The extensions we made here are needed in Chapter 3 for the elaboration of the learning system.

To begin, let us assume that the agent has a goal when interacting with the world. This goal is to obtain a state having some desired properties. The goal is decomposed into subgoals, the subgoals are themselves decomposed into subgoals, and so on, down to the level of commands. So, the initial goal is achieved via an ordered sequence of commands, and thus, via the corresponding ordered sequence of actions. Following definitions formalize these notions.

Definition:

If $T = (t_1, t_2, \dots, t_k)$ is an ordered sequence (ordered set) of elements, then

- (1) $\text{length}(T) = k$
- (2) $T(i)$ is the i^{th} element of T ($1 \leq i \leq k$)
- (3) $T(i..j)$ is the ordered sequence (ordered subset) $(t_i, t_{i+1}, \dots, t_j)$ ($1 \leq i \leq j \leq k$)
- (4) The *distance* between element t_i and element t_j is $\text{abs}(i-j)$,

where $\text{abs}()$ is the absolute value function

- (5) The empty ordered sequence is denoted $()$.

Definition:

An *interaction* is an ordered sequence of actions. The interaction is denoted

$I = (a_1, a_2, \dots, a_k)$, where a_i 's are the actions.

To each interaction corresponds an ordered sequence of states (Figure 2-1).

Figure 2-1. An interaction with its corresponding states. The figure also illustrates how the goal $\langle P_0, Q_0 \rangle$ is achieved by the execution of the action a_2 whose precondition and postcondition are P and Q , respectively.

Definition:

The ordered sequence of states corresponding to $I = (a_1, a_2, \dots, a_k)$ is $(S_0, S_1, S_2, \dots, S_k)$, where

- (1) S_0 is the state in which a_1 is executed. It is the state corresponding to the initial conditions; it may be empty.
- (2) S_i is the state resulting from the execution of a_i in state S_{i-1} ($1 \leq i \leq k$)
- (3) A special action, denoted a_0 , produces the state S_0 when executed in the empty state S^e . This action is not explicitly executed by the agent and is not shown in the enumeration of actions in I , although it is part of it.

Definition:

The membership of the action a_i in I is denoted $a_i \in I$ ($0 \leq i \leq k$).

Definition:

The ordering among actions and states is specified as follows ($0 \leq i, j \leq k$):

- (1) $a_i < a_j \iff i < j$ (similarly for $\leq$, $>$ and $\geq$)
- (2) $S_i < S_j \iff i < j$ (similarly for $\leq$, $>$ and $\geq$)
- (3) $a_i < S_j \iff i \leq j$ Note: states interleave with actions.
- (4) $S_i < a_j \iff i < j$

(5) If T contains either commands or states (but not both simultaneously), then

$\text{first}(T) = m$ such that $m \leq s$ for all $s \in T$

$\text{last}(T) = m$ such that $s \leq m$ for all $s \in T$.

In particular, the following relationships always hold:

$$a_0 < S_0 < a_1 < S_1 < \dots < a_k < S_k.$$

Now, we introduce notions required to analyze the interaction in terms of goals pursued by the agent.

Definition:

The semantics of a goal G is a pair $\langle P_0, Q_0 \rangle$ (denoted $G = \langle P_0, Q_0 \rangle$), where P_0 and Q_0 are properties. P_0 , called the *goal feasibility property*, represents the relevant facts about G allowing its satisfaction; Q_0 , called the *goal achievement property*, represents those facts that are true after its achievement.

Note that although a goal is feasible (achievable) in a given state, its achievement might not be needed or even desirable at this point.

For instance, the goal asking for the creation of an entity, expressed by

$$G = \langle \neg \text{entity}(A), \text{entity}(A) \rangle$$

is always achievable (the variable A can always be bound to a nonexistent entity), but is not always needed.

The way goals are achieved is expressed via hierarchical plans and goal rules, as defined next.

Definition:

A *task* is either a goal or a command.

Definition:

A *hierarchical plan* is an ordered sequence of tasks.

The semantics of a hierarchical plan H is described by the triple $\langle P, H, Q \rangle$, where P and Q are, respectively, the precondition and the postcondition of H . As for commands, the precondition of the hierarchical plan specifies the weakest properties that must hold in the current state for the hierarchical plan to be executable. The execution of the hierarchical plan leads (when the precondition is satisfied) to a new state where properties defined by the postcondition are satisfied. The postcondition of the hierarchical plan then describes the strongest properties obtainable. These concepts are discussed in detail in the next section.

Definition:

A *goal rule* is a rewrite rule whose left part is a goal and whose right part has three components: a precondition, a hierarchical plan and a postcondition. A goal rule for a goal $G = \langle P_0, Q_0 \rangle$ is denoted $G \Leftarrow \langle P, H, Q \rangle$ ⁷, where H , P , Q are, respectively, a hierarchical plan that achieves the goal G , the precondition of the goal rule, and the postcondition of the goal rule.

Given a goal rule $G \Leftarrow \langle P, H, Q \rangle$, P is not necessarily the precondition of H : although P is a property that guarantees the executability of H , it is not always the weakest one. Similarly, Q is not necessarily the postcondition of H ; it is the condition obtained when H is executed under P .

⁷ The notation $G \Leftarrow \langle P, S, Q \rangle$ (i.e., $\langle P_0, Q_0 \rangle \Leftarrow \langle P, S, Q \rangle$) expresses the fact that S is sufficient (and not necessarily required) to realize G : in order for G to be desirable, we have $P_0 \Leftarrow P$, and in order for G to be achieved, we have $Q_0 \Leftarrow Q$.

Making a difference between the right part of a goal rule and a hierarchical plan triple allows us to add conditions to the precondition of a hierarchical plan in order to build a goal rule achieving a goal.

For instance, for the goal $G = \langle P_0, Q_0 \rangle$, the hierarchical plan triple $\langle P, H, Q \rangle$ may give the goal rule $G \Leftarrow \langle P \wedge P_0, H, Q' \rangle$, where $P \wedge P_0$ is not the weakest condition for the executability of H (P is the weakest one), and where Q' is obtained by executing H under $P \wedge P_0$.

2.3 Semantics of preconditions and postconditions

This section defines formally what precondition and postcondition for hierarchical plans are (the problem of effectively determining the precondition and the postcondition for a given hierarchical plan is discussed in Section 3.2.3), and presents the concept of expansions and non-deterministic plans.

In what follows, we assume that the entire hierarchical plan H is required to achieve some desired goal. This is not a limiting assumption, because in order to state that not all H is required to achieve a goal G , one must know G , and it is thus possible to extract from H a sub-hierarchical plan H' that contains only the elements of H that are relevant to G , neglecting the part of H irrelevant to G ; the assumption then holds for H' , and only H' is of interest because it is sufficient to achieve the desired goal G .

The precondition of a hierarchical plan H specifies properties that must hold in the current state for H to be executable. A hierarchical plan H is not executable in a particular state where its precondition is not satisfied. This contrasts with some authors [Ginsberg and Smith 1988] where a non-executable hierarchical plan can be *executed*, producing no changes to the state.

Moreover, a hierarchical plan H might not be executable at all, regardless the current state, that is to say, that there exist no preconditions allowing H to be executed. This situation occurs when the precondition of a task T inside H is unsatisfied and it is impossible to satisfy it only by changing the initial conditions (if new tasks are added to H for providing properties that T requires, then a different hierarchical plan is obtained; H remains not executable). Usually, the problem comes from the fact that a task T' that precedes T in H , and thus is executed before T , has a postcondition that denies what T requires.

There may exist several initial conditions allowing a hierarchical plan H to be executed. For instance, if P is such an initial condition, then $P \wedge P_1$ also is, where P_1 is any predicate that is not *connected* to P nor to H (i.e., P_1 has no variable and no objects in common, it does not appear in P nor in H , and it is not related to predicates of P nor of H via axioms)⁸.

Because these artificial initial conditions are not very useful, a notion of *minimal initial condition* is required to discard them: determining the precondition of a hierarchical plan H should be formulated as *finding a minimal condition under which H is executable*.

In some domains, particularly in domains using axioms, determining if there is a unique minimal initial condition (assuming a formal definition of *minimal*), or even determining if such initial conditions exists is a difficult problem (it might be undecidable).

For instance, given an arbitrary plan in the blocks world, it might be difficult to determine if there exists an initial state in which the plan is guaranteed to be executable. As an example, consider a world in which there are two arms, and the following two commands:

⁸ One must also ensure that P_1 is not inconsistent (e.g., in the form $q \wedge \neg q$), and that it is not false in states where P is achieved.

pick_up(ARM, X): pick up the block X with the arm ARM

put_on(ARM, X): put the block held by the arm ARM on the block X.

To determine that there are no initial conditions allowing the entire execution of the sequence

(pick_up(ARM1,X), pick_up(ARM2,Y), put_on(ARM2,X)),

it is necessary to apply theorem proving techniques on the knowledge extracted from this particular blocks world, in order to prove that no matter the initial condition, the precondition of the third command is never achieved (at this point, **clear(X)** is always false, because **holding(ARM1,X)** is true).

Fortunately, in theory, **precond(H)**, the precondition of the hierarchical plan H, can be defined using the set of states where H is executable:

Definition:

Let **Terminate(H)** be the set of states in which H is executable. Then **precond(H)** is the condition that is true for all states in **Terminate(H)**, and only for these.

So, **precond(H)** is the *weakest precondition under executability*.

We assume the *expressiveness* [Clarke (Jr) 1985] of our language, that is to say, that the language (conjunction of primitive and/or inferential literals) is powerful enough to express **precond(H)** for all executable hierarchical plans H. This is a realistic assumption because of the use of inferential literals that allows disjunctions inside goals.

Talking about the determination of the postcondition of a hierarchical plan H without talking about initial conditions in which H is executed makes no sense, because the

resulting conditions usually depend on the initial conditions used. A natural candidate for these initial conditions is $\text{precond}(H)$. So, a first attempt for determining the postcondition of a hierarchical plan H might be:

$\text{postcond}(H)$

= the set of conditions obtained by executing H on $\text{precond}(H)$.

However, as a hierarchical plan H may contain tasks that involve other tasks via goal rules, and because several goal rules with different postconditions may exist for a given task, the *set of conditions obtained by executing H on a given initial condition* is not well-defined.

To see the consequence of having several hierarchical plans for achieving a particular goal, let us consider the following example.

Consider the goal $G = \langle P_0, Q_0 \rangle$ and the goal rules

$H \Leftarrow \langle P_0, (C_1, T_1), Q_0 \wedge Q_1 \rangle$ (goal rule GR_1)

$H \Leftarrow \langle P_1, (C_2, T_2), Q_0 \wedge Q_2 \rangle$ (goal rule GR_2),

where C_i 's are commands and T_i 's are tasks. Note that the goal rules for H have different preconditions: P_0 for GR_1 , P_1 for GR_2 . Assuming that $P_0 \wedge P_1$ is consistent, the minimal condition that guarantees both P_0 and P_1 (i.e., that guarantees the executability of H) is $P_0 \wedge P_1$. So, $\text{precond}(H) = P_0 \wedge P_1$.

By considering the two ways of realizing H under $\text{precond}(H)$, the possible postconditions obtained by executing H are

$Q_0 \wedge Q_1$ (if the goal rule GR_1 is used),

$Q_0 \wedge Q_2$ (if the goal rule GR_2 is used).

So, the conditions obtained depend on the way tasks are realized. If there is only one way to expand each task (and derived sub-tasks) of H , then the non-determinism vanishes. In this example, assuming that Q_1 and Q_2 are not related, we have $\text{postcond}(H) = Q_0$ (strongest postcondition), and a goal rule for G is then

$$G \leq \langle P_0 \wedge P_1, H, Q_0 \rangle \quad (\text{goal rule GR})$$

(GR achieves G because $P_0 \wedge P_1 \Rightarrow P_0$ and $Q_0 \Rightarrow Q_0$).

Example 2-1. Several hierarchical plans achieving the same goal.

The need to consider all the ways a hierarchical plan can be expanded motivates the following definition:

Definition:

An *expansion of a hierarchical plan* H is a hierarchical plan obtained by expanding all tasks and derived sub-tasks of H down to the level of commands. An *expansion of a goal rule* $G \leq \langle P, H, Q \rangle$ is a goal rule where the hierarchical plan is an expansion of H .

To illustrate the definition, let us consider a complete example.

Let us we assume the following semantics for the commands C_i 's:

$$\begin{array}{ll} \text{precond}(C_1) = P_0, & \text{postcond}(C_1) = Q_1 \\ \text{precond}(C_2) = P_1, & \text{postcond}(C_2) = Q_2, \end{array}$$

and the following goal rules for the tasks T_i 's:

$$\begin{array}{ll} T_1 \leq \langle Q_1, C_{11}, Q_0 \wedge Q_1 \wedge Q_{11} \rangle \\ T_1 \leq \langle Q_1, C_{12}, Q_0 \wedge Q_1 \wedge Q_{12} \rangle \\ T_2 \leq \langle Q_2, C_{21}, Q_0 \wedge Q_2 \wedge Q_{21} \rangle \\ T_2 \leq \langle Q_2, C_{22}, Q_0 \wedge Q_2 \wedge Q_{22} \rangle. \end{array}$$

According to these, the hierarchical plan H has the following expansions:

(C₁, C₁₁)
 (C₁, C₁₂)
 (C₂, C₂₁)
 (C₂, C₂₂),

corresponding to the following expanded goal rules:

H ≤ <P₀, (C₁, C₁₁), Q₀ ∧ Q₁ ∧ Q₁₁>
 H ≤ <P₀, (C₁, C₁₂), Q₀ ∧ Q₁ ∧ Q₁₂>
 H ≤ <P₁, (C₂, C₂₁), Q₀ ∧ Q₂ ∧ Q₂₁>
 H ≤ <P₁, (C₂, C₂₂), Q₀ ∧ Q₂ ∧ Q₂₂>.

The expansions of the goal rule GR are then:

G ≤ <P₀, (C₁, C₁₁), Q₀ ∧ Q₁ ∧ Q₁₁>
 G ≤ <P₀, (C₁, C₁₂), Q₀ ∧ Q₁ ∧ Q₁₂>
 G ≤ <P₀ ∧ P₁, (C₂, C₂₁), Q₀ ∧ Q₂ ∧ Q₂₁>
 G ≤ <P₀ ∧ P₁, (C₂, C₂₂), Q₀ ∧ Q₂ ∧ Q₂₂>.

Example 2-2. Expansions of a hierarchical plan.

The above definition allows the introduction of the following:

Definition:

A goal rule (or a hierarchical plan) is *deterministic* if it has only one expansion.

A particular and interesting case of a deterministic goal rule is when its hierarchical plan contains only commands, that is to say, the goal rule is already an expansion:

Definition:

A *plan* is a hierarchical plan containing only commands.

Although dealing with hierarchical plans is more difficult than dealing only with plans, the former approach has the main advantage of allowing the organization of tasks into

hierarchies (instead of being restricted to linear plans). This capability in turn produces the following advantages:

- a sequence of tasks can be given a meaningful name for a later reference or use;
- the sequences of tasks are shorter (a name can be used instead of the actual sequence);
- the sequences of tasks are easier to understand (by human) because they are shorter and more meaningful;
- the hierarchy is helpful to understand the goal structure (goals and subgoals involved) used by the agent to create the interaction;
- it is syntactically easier to build new sequences from old ones (sequences are shorter);
- it is semantically easier to build new sequences from old ones (semantics is provided for each task).

Because the *set of conditions obtained by executing H on a given initial condition* is well-defined only for deterministic hierarchical plans, and because most hierarchical plans are not deterministic, including the most interesting ones, the concept of postcondition is defined as follows:

Definition:

postcond(H)

= the maximal condition that is true in every state obtainable by executing H
under its precondition **precond(H)**

In other words, **postcond(H)** is the *intersection* of **postcond(X)** for all expansions X of H

For instance, we obtain from the expansions of Example 2-2 (assuming that the domain contains no axioms and that Q_1 and Q_2 have nothing in common):

$$\text{postcond(H)} = (Q_1 \wedge Q_0) \cap (Q_2 \wedge Q_0) = Q_0.$$

Here, we assumed that the domain contains no axioms and so " $\cap$ " is the intersection operation applied over the sets of literals representing the conditions. For instance, if $Q_0 = \{a_0, b_0, c_0, \dots\}$, $Q_1 = \{a_1, b_1, c_1, \dots\}$, $Q_2 = \{a_2, b_2, c_2, \dots\}$, and " $\wedge$ " represents the conjuncted predicates, then

$$\begin{aligned} & (Q_1 \wedge Q_0) \cap (Q_2 \wedge Q_0) \\ &= \{a_1, a_0, b_1, b_0, c_1, c_0, \dots\} \cap \{a_2, a_0, b_2, b_0, c_2, c_0, \dots\} \\ &= \{a_0, b_0, c_0, \dots\} \\ &= Q_0. \end{aligned}$$

So, the postcondition of a hierarchical plan is the maximal condition that is guaranteed to be satisfied regardless the expansion used.

In particular, if X is an expansion of H, then $\text{postcond(X)} \Rightarrow \text{postcond(H)}$. The following schema illustrates the relations:

Figure 2-2. Partial relationships between hierarchical plan expansions.

If the domain uses axioms, then the intersection operation must be replaced with an operation " $\cap^*$ " that involves these axioms. For instance, if the axiom $a_1 \Rightarrow a_2$ (Q_1 is more specific than Q_2) is added to the previous example, the postcondition becomes:

$$\begin{aligned}
 \text{postcond}(H) &= (Q_1 \wedge Q_0) \cap^* (Q_2 \wedge Q_0) \\
 &= \{a_1, a_0, b_1, b_0, c_1, c_0, \dots\} \cap^* \{a_2, a_0, b_2, b_0, c_2, c_0, \dots\} \\
 &= (\text{after replacing } a_1 \text{ with } a_2) \\
 &\quad \{a_2, a_0, b_1, b_0, c_1, c_0, \dots\} \cap^* \{a_2, a_0, b_2, b_0, c_2, c_0, \dots\} \\
 &= (\text{after replacing } \cap^* \text{ with } \cap \text{ because no more axioms are used}) \\
 &\quad \{a_2, a_0, b_1, b_0, c_1, c_0, \dots\} \cap \{a_2, a_0, b_2, b_0, c_2, c_0, \dots\} \\
 &= \{a_2, a_0, b_0, c_0, \dots\} \\
 &= a_2 \wedge Q_0.
 \end{aligned}$$

So, $\text{postcond}(H)$ represents the *strongest condition for the execution of H under $\text{precond}(H)$* . In terms of sets of states, $\text{postcond}(H)$ can be defined as the condition that is true for all states in $\text{Image}(H, \text{Terminate}(H))$ and only for these, where $\text{Image}(H, X)$ is the set of states obtainable by executing H on states in X .

Fortunately, it is not necessary to find all the expansions of the entire hierarchical plan H to compute $\text{postcond}(H)$. Indeed, if $H = (t_1, t_2, \dots, t_k)$ is a hierarchical plan, then $\text{postcond}(H)$ can be obtained by using $\text{precond}(t_i)$ and $\text{postcond}(t_i)$ of subtasks t_i 's, where for a given t_i , the values of $\text{precond}(t_i)$ and $\text{postcond}(t_i)$ are either directly provided (t_i is a command or a

task whose semantics is already known) or recursively computed using their own subtasks (algorithms are described in Section 3.2.3).

It is possible to extend the previous schema by applying previous definitions of $\text{precond}()$ and $\text{postcond}()$ to interactions. The complete schema is:

Figure 2-3. Complete relationships between hierarchical plan expansions.

where I is the interaction derived from X under the substitution σ (instantiation of variables in X with objects of the world), or, equivalently, where X is derived from I under σ^{-1} (*de-instantiation*).

The use of the schema is as follows: in general, the interaction I is known, and X and $\text{precond}(X)$ must be determined. In this case, X is determined by generalizing I to X by applying σ^{-1} , and the precondition of X is obtained by applying the function $\text{precond}()$ to the result:

$$X = \sigma^{-1}(I), \text{ precondition of } X = \text{precond}(\sigma^{-1}(I)).$$

For instance, consider the following interaction:

$$I = (\text{addE}(\text{student}), \text{addE}(\text{course})),$$

with the substitution $\sigma = \{E1/\text{student}, E2/\text{course}\}$ and the semantics

$$\text{precond}(\text{addE}(E)) = \neg \text{entity}(E)$$

$$\text{postcond}(\text{addE}(E)) = \text{entity}(E).$$

We have:

$$\begin{aligned} X &= \sigma^{-1}(I) \\ &= \sigma^{-1}(\text{addE}(\text{student}), \text{addE}(\text{course})) \\ &= (\text{addE}(E1), \text{addE}(E2)), \end{aligned}$$

$$\begin{aligned} \text{and } \text{precond}(X) &= \text{precond}(\text{addE}(E1), \text{addE}(E2)) \\ &= \neg \text{entity}(E1) \wedge \neg \text{entity}(E2). \end{aligned}$$

The schema also illustrates that the precondition of I can be obtained by computing $\sigma(\text{precond}(X))$, when the formula for $\text{precond}()$ is known:

$$\text{precondition of } I = \sigma(\text{precond}(X)).$$

In this case, however, it would be simpler to directly apply the function $\text{precond}()$ to I . The same mechanisms also apply to $\text{postcond}()$.

2.4 Definition of the semantics of hierarchical plans

In previous sections, we saw that four components are used to represent the semantics of hierarchical plans and tasks: a precondition, a DEL list, an ADD list, and a postcondition.

For instance, we may have:

$$\text{precond}(\text{addE}(A)) = \{\neg \text{entity}(A)\}$$

$$\text{DEL}(\text{addE}(A)) = \{ \}$$

$$\text{ADD}(\text{addE}(A)) = \{\text{entity}(A)\}$$

$$\text{postcond}(\text{addE}(A)) = \{\text{entity}(A)\}.$$

In this section, we present a detailed analysis of this representation. In particular, following considerations are discussed:

- (1) tasks' semantics is used for two different purposes
- (2) negated literals can be avoided inside DEL/ADD lists
- (3) inferential literals are useless inside DEL/ADD lists
- (4) variable sharing between DEL/ADD lists is useful and dangerous
- (5) common literals between DEL/ADD lists are useful and dangerous
- (6) redundant literals inside DEL/ADD lists are useful
- (7) what is the meaning of "applying DEL/ADD lists to a state."

2.4.1 Tasks' semantics is used for two different purposes

The previous sections introduced the semantics of tasks. This section presents the contexts where this representation is used in LEADER.

The definition of tasks are required for two different purposes in LEADER: the first one is to build the sequence of states corresponding to a given interaction. This sequence of states is then analyzed using the current domain theory, and new knowledge is proposed to explain a larger part of the interaction. Producing this knowledge requires the creation of hierarchical tasks and the determination of their precondition and postcondition. The determination of these conditions constitutes the second purpose.

In brief, tasks' semantics are used for the following reasons:

Purpose 1: application of a task

given a state S and a task T whose precondition is satisfied in S , determine the state S' resulting when T is executed on S ;⁹

Purpose 2: composition of tasks

given two tasks $T1$ and $T2$ along with their semantics (precondition, DEL/ADD lists and postcondition), determine the semantics of the hierarchical plan $(T1, T2)$ (with left-to-right execution: $T1$ is executed before $T2$).

For the first purpose, the semantics traditionally is used to delete from the state any literal that unifies with an element of the DEL list, and adding the literals of the ADD list to it.

Because the result of the application of lists may be sensitive to the order of application (for instance, if the DEL and ADD lists have common elements), it is usually assumed that the DEL list is applied before the ADD list is, as follows:

- verify if the precondition holds in the state;
- if so, delete from the state any literal that unifies with an element of the DEL list, and add the literals of the ADD list to the resulting state.

More details about the need for both purposes are given through the remaining subsections, in relationship with the discussion of topics mentioned above.

⁹ Note that the purpose is to compute only S' , and not the set of all literals (primitive and inferential ones) that are true in the resulting state. The latter is computed via axioms, once S' is known.

2.4.2 Negated literals can be avoided inside DEL/ADD lists

In general, the presence of a negated literal $\neg p$ (where p is positive) inside the DEL list can be interpreted either as *$\neg p$ ceases to be true after the execution of the task* or *it is no more possible to infer the truth value of $\neg p$* . In contrast, the presence of a negated literal $\neg p$ inside the ADD list would represent that *$\neg p$ becomes true after the execution of the task*.

For the purpose of applying a task (Purpose 1), it is not required to have negated literals inside DEL/ADD lists, because states contain only positive literals. Indeed, negated literals in the DEL cannot unify with an element of the state, and a negated literal of the ADD list cannot be added to the state. In other words, negated literals in DEL/ADD lists are useless.

For the purpose of composition of tasks (Purpose 2), the above argument does not apply: although negated literals are not used to compute states, they might be useful to correctly express the semantics of the task. However, under the closed-world assumption, any DEL/ADD lists containing negated literals can be replaced with an equivalently expressive pair of DEL/ADD lists containing only positive literals. For instance, the following lists containing negated literals (p, q, r, s are positive literals):

DEL = $\{p, \neg q\}$, ADD = $\{r, \neg s\}$
 (meaning: p and $\neg q$ are not provable, r and $\neg s$ are provable)

can be converted into the following equivalent lists:

DEL = $\{p, s\}$, ADD = $\{r\}$
 (meaning: p and s are not provable, r is provable).

In both cases, under the closed-world assumption, p, s and q are false, and r is true.

In summary, it is assumed in LEADER that DEL/ADD lists contain only positive literals.

2.4.3 Inferential literals are useless inside DEL/ADD lists

In what follows, we show that

- (1) inferential literals are not required inside DEL/ADD lists
- (2) inferential literals are not always sufficient to describe the semantics when used inside DEL/ADD lists.

This allows us to reject the use of inferential literals inside DEL/ADD lists.

Firstly, let us consider Purpose 1: finding the state obtained by executing an action.

A surprising fact is that inferential literals are not used to compute states. Indeed, when a task is executed on a state S , what is finally applied is a particular expansion of the task, that is to say, an ordered sequence of (deterministic) actions whose semantics is given in terms of primitive literals (because states contain only primitive literals). So, inferential literals are never required to compute a state S ; they are only used to compute preconditions and postconditions.¹⁰

Thus, inferential literals are not required inside DEL/ADD lists, and their presence in these lists (as for the presence of negative literals) does not interfere with the computation of states because states contain only primitive literals¹¹.

Now, let us consider Purpose 2: determining the semantics of a hierarchical plan.

¹⁰ By changing the mechanism that applies DEL/ADD lists to states to take into account inferential literals (e.g., by removing from the state all literals that can prove via axioms a literal in the DEL list, and those that can prove the negation of a literal in the ADD list), then having inferential literals inside the lists may have consequences on the computation of states. Although the mechanism we proposed will be later on refined, it will not be modified to accept inferential literals.

¹¹ However, if inferential literals are present in the ADD list, then before adding to a state a literal from the ADD list, it must be checked that the literal is primitive.

Having inferential literals in DEL/ADD lists may help to determine the semantics of the hierarchical plan. For instance, a condition expressed with inferential literals usually is easier to read and understand than the equivalent one expressed in terms of primitive literals.

In fact, because inferential literals are used to represent disjunctions inside conditions (recall that disjunctions are not directly allowed inside conditions), the semantics (i.e., the four components) of a non-deterministic hierarchical plan usually cannot be expressed without inferential literals. To see this, let us consider the following example.

The semantics of the task $T = \text{"remove_many_to_many_rel}(A,B) \text{"}$ (remove a many-to-many relationship between entities A and B in an entity-relationship model) can be expressed by inferential literals as:

```
precond(T) = many_to_many(A,B)
DEL(T) = {many_to_many(A,B)}
ADD(T) = { }
postcond(T) =  $\neg$ many_to_many(A,B),
```

using the axioms

```
many_to_many(A,B) <= 1_to_many(A,B)  $\wedge$  1_to_many(B,A)
1_to_many(A,B) <= card(max,A,B,n) 12
1_to_many(A,B) <= card(max,A,B,C)  $\wedge$  C > 1.
```

The semantics of T cannot be expressed only by primitive literals because it requires disjunctions, as in

```
precond(T)
= (card(max,A,B,n)  $\vee$  (card(max,A,B,C1)  $\wedge$  C1 > 1))
 $\wedge$  (card(max,B,A,n)  $\vee$  (card(max,B,A,C2)  $\wedge$  C2 > 1)).
```

Example 2-3. Need for inferential literals in tasks' semantics.

¹² The predicate "card(max,A,B,C)" specifies that the maximal cardinality from entity A to entity B is C , which is either n or a non-negative number.

The same reasoning applies to the postcondition. Also, there are many ways to remove a many-to-many relationship (i.e., the task has several expansions), and the different ways correspond to different DEL lists:

if T was achieved by removing "card(max,A,B,n)" from the state,

$$\text{DEL}(T) = \{\text{card}(\text{max}, \text{A}, \text{B}, n)\}$$

if T was achieved by removing "card(max,A,B,2)" from the state,

$$\text{DEL}(T) = \{\text{card}(\text{max}, \text{A}, \text{B}, 2)\}$$

if T was achieved by removing "card(max,B,A,5)" from the state,

$$\text{DEL}(T) = \{\text{card}(\text{max}, \text{B}, \text{A}, 5)\}$$

if T was achieved by deleting the entity A from the state,

$$\text{DEL}(T) = \{\text{entity}(\text{A}), \text{relation}(\text{A}, _, _), \text{card}(_, \text{A}, _, _), \dots\}$$

...

It is impossible to express all these inside DEL(T) using only primitive literals. However, it is not necessary to use inferential literals inside DEL/ADD lists, if the only interest is the formulation of the semantics. If it is the case, the semantics can be formulated by providing the precondition and the postcondition of each task.

The previous example shows that literals (and thus inferential literals) are not required inside DEL/ADD lists for the only purpose of the formulation of the semantics. The next example shows that in some cases, having inferential literals inside DEL/ADD lists is not sufficient: they must appear inside the postcondition.

To see that the DEL/ADD mechanism alone (with or without inferential literals) is inadequate in presence of non-deterministic tasks, we build two tasks (T_1 and T_3) having different postconditions, although their preconditions and DEL/ADD lists are identical. Consider following tasks:

$T_1 = \text{delR}(A,B)$

{Delete the relationship between entities A and B}

$\text{precond}(T_1) = \text{entity}(A) \wedge \text{entity}(B) \wedge \text{related}(A,B)$

$\text{DEL}(T_1) = \{\text{related}(A,B)\}$

$\text{ADD}(T_1) = \{ \}$

$\text{postcond}(T_1) = \text{entity}(A) \wedge \text{entity}(B) \wedge \neg \text{related}(A,B)$

$T_2 = (\text{delR}(A,B), \text{delE}(A), \text{delE}(B))$

{Delete the relationship between entities A and B, and the entities themselves}

$\text{precond}(T_2) = \text{entity}(A) \wedge \text{entity}(B) \wedge \text{related}(A,B)$

$\text{DEL}(T_2) = \{\text{entity}(A), \text{entity}(B), \text{related}(A, _),$
 $\text{related}(_, B), \text{related}(B, _), \text{related}(_, A)\}$

$\text{ADD}(T_2) = \{ \}$

$\text{postcond}(T_2)$

$= \neg \text{DEL}(T_2)$

$= \neg \text{entity}(A) \wedge \neg \text{entity}(B) \wedge \neg \text{related}(A, _)$

$\wedge \neg \text{related}(_, B) \wedge \neg \text{related}(B, _) \wedge \neg \text{related}(_, A).$

Consider the task T_3 that is the execution of either T_1 or T_2 , that is to say, T_1 and T_2 are two hierarchical plans for achieving the task T_3 .

Because $\text{precond}(T_1) = \text{precond}(T_2)$, $\text{precond}(T_3)$ is also $\text{precond}(T_1)$, that is to say,

$\text{precond}(T_3) = \text{entity}(A) \wedge \text{entity}(B) \wedge \text{related}(A,B).$

Because T_3 can be achieved either by T_1 or by T_2 , it is not deterministic, and $\text{postcond}(T_3)$ must indicate the strongest

condition that is true (guaranteed) for all expansions of T_3 . By extracting the maximal common part of $\text{postcond}(T_1)$ and $\text{postcond}(T_2)$, we get

$$\text{postcond}(T_3) = \neg \text{related}(A, B).$$

Similarly, $\text{DEL}(T_3)$ indicates the maximal common part of $\text{DEL}(T_1)$ and $\text{DEL}(T_2)$ (that is also the maximal common part of negated literals of $\text{postcond}(T_1)$ and $\text{postcond}(T_2)$):

$$\text{DEL}(T_3) = \text{related}(A, B).$$

Finally, $\text{ADD}(T_3) = \{ \}$, and we have:

$$\text{precond}(T_3) = \text{entity}(A) \wedge \text{entity}(B) \wedge \text{related}(A, B)$$

$$\text{DEL}(T_3) = \{ \text{related}(A, B) \}$$

$$\text{ADD}(T_3) = \{ \}$$

$$\text{postcond}(T_3) = \neg \text{related}(A, B).$$

We observe that although T_1 and T_3 have the same precondition and the same DEL/ADD lists, they have different postconditions.

Example 2-4. Need for inferential literals inside the postcondition.

This example shows that, if the domain allows non-deterministic tasks, $\text{postcond}(T)$ is not a (mathematical) function of $\text{precond}(T)$, $\text{DEL}(T)$ and $\text{ADD}(T)$: several $\text{postcond}(T)$ are possible. Thus, there exist no algorithms for computing $\text{postcond}(T)$ given only the other three components of the semantics of T .

This is a big limitation in our case because non-deterministic tasks appear (usually) as soon as hierarchical tasks are introduced, and hierarchical tasks (i.e., goal structures) are the basis of LEADER. This does not mean that the postcondition of hierarchical tasks cannot be computed, but simply that the DEL and ADD lists alone cannot be used for that purpose.

For instance, if T is a non-deterministic task, and the index E provides all expansions of T , then we have (neglecting the handling of inferential literals):

$$\text{postcond}(T) = (\text{precond}(T) - \text{ANY_DEL}(T)) \cup \neg\text{DEL}(T) \cup \text{ADD}(T)$$

where " $\text{ANY_DEL}(T)$ " is the set of literals deleted by any of the expansions of T , where " $\text{DEL}(T)$ " (respectively " $\text{ADD}(T)$ ") is the set of literals deleted (respectively added) by each of the expansions of T , where " $\text{precond}(T) - \text{ANY_DEL}(T)$ " is the set of literals that are guaranteed to remain from the postcondition, where " $\neg\text{DEL}(T) \cup \text{ADD}(T)$ " is the set of literals that is guaranteed to be added, and where " $-$ " and " $\cup$ " are adjusted to take into account negative literals, as expressed by the following:

$$\begin{aligned}\text{ADD}(T) &= \bigcap_E \text{ADD}(E) \\ \text{DEL}(T) &= \bigcap_E \text{DEL}(E) \\ \text{ANY_DEL}(T) &= \bigcup_E \text{DEL}(E).\end{aligned}$$

The removal of negated literals appearing in any expansion ensures that the semantics is conservative (that is to say, any literal appearing in $\text{postcond}(T)$ is guaranteed to appear in the resulting state).

For the previous example, we have:

$$\begin{aligned}\text{ADD}(T) &= \bigcap_E \text{ADD}(E) = \{ \} \\ \text{DEL}(T) &= \bigcap_E \text{DEL}(E) = \{\text{related}(A,B)\} \\ \text{ANY_DEL}(T) &= \bigcup_E \text{DEL}(E) \\ &= \{\text{entity}(A), \text{entity}(B), \text{related}(A, _), \text{related}(_, B), \text{related}(B, _), \text{related}(_, A)\}\end{aligned}$$

$$\begin{aligned}
\text{precond}(T) &= \text{ANY_DEL}(T) \\
&= \{ \text{entity}(A), \text{entity}(B), \text{related}(A,B) \} - \\
&\quad \{ \text{entity}(A), \text{entity}(B), \text{related}(A, _), \text{related}(_, B), \text{related}(B, _), \text{related}(_, A) \} \\
&= \{ \} \quad (\text{because } \neg \text{related}(A, _) \Rightarrow \neg \text{related}(A, B)) \\
\\
\text{postcond}(T) &= (\text{precond}(T) - \text{ANY_DEL}(T)) \cup \neg \text{DEL}(T) \cup \text{ADD}(T) \\
&= \{ \} \cup \neg \{ \text{related}(A, B) \} \cup \{ \} \\
&= \{ \neg \text{related}(A, B) \}.
\end{aligned}$$

This shows that the postcondition of a task T can be computed if more information is available than provided solely by the DEL/ADD lists of T (for instance, by using as above information on each of T 's expansion).

In summary, for the second purpose (computing the semantics of a hierarchical plan), inferential literals are not needed inside DEL/ADD lists to express the semantics of a task if the purpose is to obtain a semantics that is easy to understand by humans. It is not always helpful to have them inside DEL/ADD lists for the purpose of computing the semantics of a hierarchical plan. Moreover, they are sometimes needed (inside at least one of the semantics components) to express disjunctions.

Putting results for both purposes together, we can assume (in LEADER), without loss of expressiveness or computational power, that DEL/ADD lists contain no inferential literals.

2.4.4 Variable sharing between DEL/ADD lists is useful and dangerous

This section analyzes the various consequences of having a variable appearing in both a literal of the ADD list and a literal of the DEL list. In particular, the section presents

problems arising when standard unification is used for applying ADD and DEL lists (solution approaches are presented for these problems).

First, notice that universal quantification is assumed over variables appearing inside DEL and ADD lists. Now, consider the following example illustrating that the DEL list can share common variables with the ADD list, via their literals:

```
postcond( paint_object(Obj,Color) ) = color(Obj,Color)
DEL = { color(Obj,_) }
ADD = { color(Obj,Color) }.
```

Because the semantics of a task is a general schema involving variables, and because variables appearing in the name of the command (here, `paint_object(Obj,Color)`) are susceptible to appear both in the DEL and ADD lists, these lists are likely to have common variables.

However, a more interesting observation is that the DEL list can share common variables with the ADD list, via their literals, even when the name of the command is fully instantiated.

For instance, using the substitution $\{Obj/box1, Color/red\}$, the previous semantics becomes:

```
postcond( paint_object(box1,red) ) = color(box1,red)
DEL = { color(box1,_) }
ADD = { color(box1,red) }.
```

Although the substitution leaves uninstantiated some variables inside DEL/ADD lists, the lists share no common variables. However, this is not always the case. For instance, to

update the rank of an employee in a database, using the employee number to extract the database record, the task semantics is:

```
postcond( update_employee_rank(NoEmpl, Rank) ) =  
          empl(NoEmpl, NameEmpl, Rank)  
DEL = {empl(NoEmpl, NameEmpl, _)}  
ADD = {empl(NoEmpl, NameEmpl, Rank)}.
```

After using the substitution {NoEmpl/234, Rank/cs2}, the DEL/ADD lists still contain common uninstantiated variables. These variables are used to transmit the information from the extracted record to the new record:

```
postcond( update_employee_rank(234, cs2) ) =  
          empl(234, NameEmpl, Addr, cs2)  
DEL = {empl(234, NameEmpl, Addr, _)}  
ADD = {empl(234, NameEmpl, Addr, cs2)}.
```

To correctly handle such situations, we make the following assumption:

because the DEL list is applied before the ADD list, the DEL list is used to instantiate additional variables of the command when it is applied to the state.

Because states contain only fully instantiated literals, no variables of the DEL list are left uninstantiated after its application. However, the ADD list may still contain variables, as in the following hypothetical semantics:

$$\text{postcond}(\text{command1}(A,B)) = p2(A,B,C,_)$$

$$\text{DEL} = \{p1(A,B,C)\}$$

$$\text{ADD} = \{p2(A,B,C,_)\}.$$

Because what is added to states must be fully instantiated, no variable in the ADD list must remain uninstantiated after the application of the DEL list. To eliminate such situations when computing states, the following assumption is added:

after the application of the DEL list to any state, all literals in the ADD list are fully instantiated.

We can observe that this is not a problem for the purpose of the formal computation of semantics of hierarchical plans, because no states are used.

Another problem may occur when a literal in DEL unifies with several literals in the state. First, notice that this phenomenon is required in some cases, for instance, to express that the deletion of an entity implies the deletion of all links involving this entity:

$$\text{precond}(\text{delE}(E)) = \text{entity}(E)$$

$$\text{DEL} = \{\text{entity}(E), \text{relation}(E,_,_)\}$$

$$\text{ADD} = \{ \}$$

$$\text{postcond}(\text{delE}(E)) = \neg \text{entity}(E) \wedge \neg \text{relation}(E,_,_).$$

Here, the substitution $\{E/e\}$ provides the semantics:

$$\text{precond}(\text{delE}(e)) = \text{entity}(e)$$

$$\text{DEL} = \{\text{entity}(e), \text{relation}(e,_,_)\}$$

$$\text{ADD} = \{ \}$$

$$\text{postcond}(\text{delE}(e)) = \neg \text{entity}(e) \wedge \neg \text{relation}(e,_,_)$$

and the literal $relation(e, _, _)$ unifies with several elements of the state, each of them being deleted, producing the desired resulting state:

Before: $S = \{entity(e), entity(a), entity(b), relation(e,a,role), relation(e,b,role)\}$

After: $S' = \{entity(a), entity(b)\}$

However, observe that a problem arises in the following similar case:

$S = \{p3(a,b,c), p3(d,e,f)\}$

$DEL = \{p3(A,B,C)\}$

$ADD = \{p4(C)\}.$

Here, the literal in DEL unifies with two literals of the state, and each one provides a different substitution. Because the value of the variable C depends on the substitution used, and because C appears inside the ADD list, the meaning of *add literals of the ADD list to the state* is not clear; one of the following resulting states might be obtained with standard unification:

(1) substitution = $\{A/a, B/b, C/c\}$, $S' = \{p4(c)\}$

(2) substitution = $\{A/d, B/e, C/f\}$, $S' = \{p4(f)\}$

(3) substitution = $\{A/a, B/b, C/ c \text{ or } f\}$, $S' = \{p4(c), p4(f)\}.$

The possibility (3) is the least restrictive (so, the most interesting). However, because this situation is complex and is not needed in the context we defined (goals, agent, state...), we eliminate it with the following convention:

if a literal in the DEL list unifies with several literals in the state, then all unified literals in the state are removed, and the substitution applied to the ADD list is the substitution of the last unified literal.

The substitution returned is well-defined (unique) and depends on the order in which the literals of the state are considered. This order depends on the particular implementation. This convention generalizes naturally the case where there is only one unifying literal in the state, as for the database update example.

Now, let us point out another potential problem with the application of DEL/ADD lists, when standard unification is performed. Consider the following semantics:

$$\begin{aligned} S &= \{p5(a,b), p6(a,c)\} \\ DEL &= \{p5(A,B), p6(A,B)\} \\ ADD &= \{p7(B)\}, \end{aligned}$$

that gives, with the substitution $\{A/a\}$,

$$\begin{aligned} S &= \{p5(a,b), p6(a,c)\} \\ DEL &= \{p5(a,B), p6(a,B)\} \\ ADD &= \{p7(B)\}. \end{aligned}$$

Although each literal of the DEL list unifies exactly once with an element of the state, each one provides a different substitution for the variable B (either b or c). So, the substitution applied to the ADD list is not well-defined. Even the meaning of DEL is not clear: should we assume that because the DEL list uses the variable B inside both of its literals, in contrast with the list

$$DEL = \{p5(a,_), p6(a,_)\},$$

then it is important that literals sharing the variable B are unified in parallel (either none or all of them unify with a given substitution)? It makes sense to answer this question by the affirmative. Furthermore, the shared-variables mechanism provides an elegant way to express such requirements.

Again, we propose a more appropriate unification process that generalizes the non-problematic cases, assuming that literals having variables in common must not be unified independently of each other:

- build an undirected graph whose nodes are the literals of DEL and such that for each pair of literals in DEL that share the same variables, there is a link in the graph between the corresponding nodes; each link is labelled with the corresponding name of the variables
- identify the connected components of the graph
- for each connected component
 - try to unify in parallel all literals in the component with literals of the state (either all of them unify, or none of them do)
 - if the unification is successful, delete from the state all unified literals, and try another unification
 - store the substitution corresponding to the last unification (if there is one).

Note that unifying the connected components and returning only the last substitution to apply to the ADD list guarantees that each variable of the ADD list will have at most one substitution¹³.

For the previous example, the graph built for the application of the DEL list is

The graph has only one connected component, the component does not unify with the state, and no substitution is applied to the ADD list. So, the variable **B** remains uninstantiated thus violating the above assumption about variables of the ADD list.

For DEL lists whose literals have no variable sharing, each node of the corresponding graph is a connected component, and the unification process is as usual.

2.4.5 Common literals between DEL and ADD lists are useful and dangerous

Similarly to the previous section, this one analyzes the various consequences of allowing the same literal to appear in both the ADD list and the DEL list of a given command.

If DEL/ADD lists contain only propositions (predicates without arguments), then it can be assumed that no proposition appears in both lists. To see this, observe that the list pair

$$\text{DEL} = \{p, q, r\}, \quad \text{ADD} = \{p, s, t\}$$

can be replaced with the equivalent pair

$$\text{DEL}' = \text{DEL} - \text{ADD} = \{q, r\}, \quad \text{ADD}' = \text{ADD} = \{p, s, t\}$$

¹³ Note that the assumption stating that at least one substitution is provided for each variable of the ADD list is still required.

where $DEL' \cap ADD' = \{ \}$. They are equivalent because they both specify that the resulting state contains p , s and t , but contains neither q nor r .

If DEL/ADD lists contain literals (with arguments), the concept of *common literals* can no more be expressed in terms of set intersections as above, and the validity of the previous transformation will depend on the way it is defined. First, consider the following convention:

Tentative Definition 1:

The sets of literals A and B have literals in common if at least one literal in A unifies with at least one literal in B . If it is the case, to each pair of unifying literals corresponds (after the unification) a common literal.

For instance, $\{entity(a), \neg entity(b)\}$ and $\{entity(X)\}$ have the literal $entity(a)$ in common.

According to this definition, the previous transformation is not always valid. Indeed, consider the semantics used to update a database record (presented above):

$postcond(update_employee_rank(NoEmpl, Rank)) =$

$empl(NoEmpl, NameEmpl, Rank)$

$DEL = \{ empl(NoEmpl, NameEmpl, _) \}$

$ADD = \{ empl(NoEmpl, NameEmpl, Rank) \}.$

Because $empl(NoEmpl, NameEmpl, _)$ unifies with $empl(NoEmpl, NameEmpl, Rank)$, the previous transformation gives the semantics


```
postcond( update_employee_rank(NoEmpl, Rank) ) =  
          empl(NoEmpl, NameEmpl, Rank)  
DEL = { }  
ADD = { empl(NoEmpl, NameEmpl, Rank) }
```

which is not equivalent to the previous one: DEL lists are different, and a different state results.

Now, consider the following definition:

Tentative Definition 2:

The sets of literals A and B have literals in common if at least one literal in A is syntactically identical to a literal in B. In particular, variables occurring at the same relative position in the literals must be identical (that is, literals must unify with the empty substitution). If it is the case, each common identical literal is a common literal.

For instance, {entity(A),domain(B)} and {entity(A),domain(C)} have only the literal entity(A) in common; domain(B) is not syntactically identical to domain(C).

With this definition, the above problem disappears because DEL and ADD no more have a common literal (in the current sense). However, the following problem remains: it is possible that DEL and ADD lists have a common literal that is used to unify and delete several literals in the state. Consider for instance:

$$S = \{p8(a,b), p8(a,c)\}$$

$$DEL = \{p8(A,B)\}$$

$$ADD = \{p8(A,B)\}.$$

According to the previous discussion, the resulting state is (assuming that the last unifying literal returned by the application of the DEL list is the right-most one of the state):

$$S' = \{p8(a,c)\}.$$

This result is not compatible with the application of the above transformation that leads to the semantics:

$$DEL = \{ \}$$

$$ADD = \{p8(A,B)\}$$

and so, to the resulting state is (nothing is deleted, and nothing is added because $p8(A,B)$ is not fully instantiated when applied to the state):

$$S' = S = \{p8(a,b), p8(a,c)\}.$$

Instead of contorting the definition of common literal in an unnatural way to solve the problem (for instance, by restricting common literals to fully ground literals), it is preferable to take the Tentative Definition 1 as the actual definition, and simply allow common literals within DEL and ADD lists.

2.4.6 Redundant literals inside DEL/ADD lists are useful

Using conventions established so far, let us now discuss the possibility of using useless literals inside the DEL and ADD lists. More precisely, the question is whether the DEL list can contain literals that are not in the state where it is applied, and, similarly, if the ADD list

can contain literals that are already in the state. As explained next, allowing these redundancies may be useful because it leads to more general semantics of actions, as used in the domain theory of systems used to analyze and evaluate LEADER (Chapter 4).

First, note that from the definition of applying a DEL list to a state, if a literal L from the DEL list is not present in the state, then no unification with L can occur, L cannot be retracted from the state, and no unification can be propagated to the ADD list. So, L does not affect the resulting state.

However, having such apparently useless literals in the DEL list may be helpful. For instance, consider the semantics of the command $delE(E)$. The DEL list contains two literals,

$$\text{precond}(delE(E)) = \text{entity}(E)$$
$$\text{DEL} = \{\text{entity}(E), \text{relation}(E, _, _)\}$$

and only one of them, $\text{entity}(E)$, is guaranteed to be present in the state when the precondition is satisfied. Having the literal $\text{relation}(E, _, _)$ inside the DEL list (and inside the postcondition) is helpful because the semantics is more general. In fact, if only totally unifying DEL lists (each literal unifies with a literal in the state) are allowed, then several sub-commands are usually required to describe a command. For instance, for the command $delE(E)$ used in Modeller (a software tool to assist designers of entity-relationship model – described in Chapter 4), two sub-commands must be defined.

$$\text{precond}(\text{delE}_1(E)) = \text{entity}(E) \wedge \neg \text{relation}(E, _, _)$$

$$\text{DEL}_1 = \{\text{entity}(E)\}$$

$$\text{ADD}_1 = \{ \}$$

$$\text{postcond}(\text{delE}_1(E)) = \neg \text{entity}(E) \wedge \neg \text{relation}(E, _, _).$$

$$\text{precond}(\text{delE}_2(E)) = \text{entity}(E) \wedge \text{relation}(E, _, _)$$

$$\text{DEL}_2 = \{\text{entity}(E), \text{relation}(E, _, _)\}$$

$$\text{ADD}_2 = \{ \}$$

$$\text{postcond}(\text{delE}_2(E)) = \neg \text{entity}(E) \wedge \neg \text{relation}(E, _, _).$$

Although they both have totally unifying DEL lists, they are not more readable than the original one; to the contrary. As the example illustrates, having non-unifying literals inside the DEL list is sometimes helpful, and never harmful. We thus make use of this mechanism.

A similar reasoning reveals that allowing literals in the ADD list that are also in the state or in the precondition is always harmless and sometimes useful.

2.4.7 What is the meaning of "applying DEL/ADD lists to a state"

The following conventions and methods summarize the above discussion:

Assumptions:

- the DEL and ADD lists contain no negative literals
- the DEL and ADD lists contain no inferential literals
- the DEL and ADD lists may have literals in common
- the DEL list may contain literals that are not in the state nor in the precondition
- the ADD list may contain literals that are in the state or in the precondition
- after the application of the DEL list to any state, all literals in the ADD list are fully instantiated.

To apply a DEL/ADD list to a state S , given a substitution σ used to convert commands into actions¹⁴:

(1) apply the substitution σ to the lists DEL and ADD

(2) apply the resulting DEL list to S as follows:

- build an undirected graph whose nodes are the literals of DEL and such that for each pair of literals in DEL that share the same variables, there is a link

¹⁴ The current version of LEADER does not implement all the possibilities handled by the proposed method; only those present in the domain used to evaluate LEADER are supported.

in the graph between the corresponding nodes; the link is labelled with the name of the variables

- identify the connected components of the graph
- for each connected component
 - try to do the unification in parallel for all literals in the component, with literals of the state (either all of them unify, or none of them does)
 - if the unification is successful, delete from S all unified literals, and try another unification
 - store the substitution corresponding to the last unification (if there is one)

(3) apply the resulting ADD list to the resulting S as follows:

- apply the substitutions stored while applying the DEL list to the resulting ADD list to obtain the final ADD list
- if some literals of the final ADD list contain variables, the whole application of the DEL/ADD lists to S fails (violation of the above assumption)
- otherwise, add all literals of the final ADD list to S.

Finally, because

- the DEL/ADD mechanism can be used to compute the state resulting in the execution of a task (when only deterministic tasks are present in the theory)

- it can be used to express the semantics of tasks
- it may lead to inconsistencies when used for computing the semantics of non-deterministic tasks,

we propose the following conventions for the use of the four components describing the semantics of a task (precondition, DEL/ADD lists, postcondition),

- the DEL/ADD lists appear only in the semantics of commands and are used only to compute states
- the precondition and the postcondition appear in the semantics of both commands and hierarchical tasks, and are used only to compute formal conditions, not to compute states.

Algorithms for computing formal conditions are described in detail in Section 3.2.

2.5 Explaining interactions

As mentioned in Chapter 1, LEADER learns by building explanations for the hierarchical plans and goals used in the interaction, by identifying what is left unexplained, and by inferring what is missing in the current existing knowledge to explain the interaction.

This section presents the representation needed to construct explanations of the interaction. In particular, the following notions will be introduced: direct and indirect actions (actions used in goal rules as part of the plans or as contributing to achieve their precondition), achievement vector (a binary tuple describing the degree of achievement of a goal for a given position of the interaction), goal tree and goal graph (structures describing how the domain theory currently explains parts of the interaction), partial realization (a measure of the proportion of a goal rule achieved through the interaction), proof tree and causal link

(structures describing how actions contribute to achieve the precondition of goal rules), and supporting actions (actions whose effect contribute to satisfy a goal).

Because an interaction is composed of actions, explaining a part of an interaction means justifying why corresponding actions are used. An action may be justified because:

- the action is used within a hierarchical plan;
- the action achieves a property needed as a precondition of a task;

The first type of justification is the most straightforward: an action that realizes a hierarchical plan, that in turn achieves a goal, is justified by the desire to satisfy this goal. Justifications of this type are identified by building the goal graph (Section 2.5.1), a structure for explaining the goal structure used by the agent.

It may appear that justifications of the second type form a subset of those of the first type. Indeed, in planning, to achieve a property required as a precondition of some task, a subgoal is usually fired. Then, this subgoal fires an appropriate sub-hierarchical plan later decomposed into actions. These actions would be explained by the sub-hierarchical plan used and are first-type justifications.

However, if the existing knowledge contains no information about any sub-hierarchical plan achieving the desired property, corresponding actions cannot be explained by a goal rule, but only by achieving that property. Consequently, this second type of justification must be taken into account. However, it would be required later to infer new hierarchical plans (along with goal rules, goals, etc.) to entirely understand related actions. Justifications of the second type are obtained by the analysis of causal links (Section 2.5.2).

The above discussion justifies the following definitions (in particular, for the concepts that are used in Chapter 4 to analyze the learning capability of LEADER):

Definition:

An action is *direct* if it appears in the expansion of a goal rule used to achieve a goal. The action is *indirect* if it is required to achieve the precondition of a goal rule used to achieve a goal (some of the postcondition of the action appears in the goal rule precondition).

Before introducing the notions of goal graph structure and causal links in the next two sub-sections, let us present a complementary notion useful to characterize the explanations:

Definition:

For any goal $G = \langle P_0, Q_0 \rangle$, any conditions P and Q , and any states S and S' , the corresponding *achievement vector* is the 4-tuple

$$AV = (P(S), P_0(S), Q(S'), Q_0(S')).$$

At first glance, sixteen different achievement vectors seem possible. However, because $P(S) \Rightarrow P_0(S)$ and $Q(S') \Rightarrow Q_0(S')$, only nine possibilities exist, as the seven achievement vectors in the form $(true, false, _, _)$ or $(_, _, true, false)$ cannot be achieved. Each one of these nine possibilities provides a particular characterization of various parts of an interaction. These characterizations are described in detail in Chapter 3 when the learning algorithms are presented.

The following two sub-sections present the goal graph structure and the causal links, respectively.

2.5.1 Goal graphs describe the structure of goals

Let $I = (a_1, a_2, \dots, a_k)$ be an interaction and GR be a set of goal rules.

Definition:

A *goal graph* GG over the interaction I is a pair $\langle N, PC \rangle$, where N is a set of tasks (nodes), and PC defines the parent-child relations between elements of N according to the goal rules in GR.

The actions of I are the leaves of the goal graph GG (thus, $I \subseteq N$), each internal node T of GG corresponds to a goal rule R in GR, and the children of T in GG are the subtasks of the hierarchical plan of R.

Figure 2-4 shows a goal graph built over the interaction $I = (a_1, a_2, \dots, a_7)$, using the following goal rules (conditions are not shown, node names represent task names):

$$n_8 \leq a_1, a_2$$

$$n_9 \leq n_8, a_3, a_4, a_5$$

$$n_{10} \leq a_3, a_4, a_6$$

Figure 2-4. Goal graph built over an interaction.

As an illustration of the contents of GG and PC, we have for this goal graph:

$$N = \{n_1 = a_1, n_2 = a_2, \dots, n_7 = a_7, n_8, n_9, n_{10}\}$$

$$PC = \{ \langle n_8, a_1 \rangle, \langle n_8, a_2 \rangle, \langle n_9, n_8 \rangle, \langle n_9, a_3 \rangle, \langle n_9, a_4 \rangle, \langle n_9, a_5 \rangle, \\ \langle n_{10}, a_3 \rangle, \langle n_{10}, a_4 \rangle, \langle n_{10}, a_6 \rangle \}$$

Definition:

A *goal tree* for an interaction I and a goal rule R is a tree contained in the goal graph of I, and whose root is (unifies with) the left part of R. Tasks, goal rules, hierarchical plans, preconditions and postconditions used in that goal tree are said to be realized by the sub-interaction formed by the ordered leaves of the goal tree; these leaves are then explained by the goal tree. A goal graph for an interaction I is said to explain I if each action of I (excluding a_0) is explained by a goal tree.

Goal rules correspond to the domain theory for goals as used in EBL systems [Mitchell, et al. 1986]. In particular, *goal regression* [ibid] of the goal tree would compute the generic goal tree for G.

Because the tree structure used, using the same node more than once is not allowed. For instance, using the goal rules

$$\begin{aligned} n_{13} &\leq n_{11}, n_{12} \\ n_{11} &\leq a_1, a_2 \\ n_{12} &\leq a_2, a_3, \end{aligned}$$

the dag (directed acyclic graph) shown in Figure 2-5 does not constitute a valid explanation.

Figure 2-5. Invalid explanation built over an interaction.

To build a valid explanation using these goal rules, the interaction should contain at least two occurrences of the action a_2 , as in (a_1, a_2, a_2, a_3) .

The preference for the tree representation over the dag representation stems from following considerations:

- trees are simpler structures than dags, and consequently are simpler to manipulate
- short circuit hierarchical plans consisting of avoiding repeated tasks are learnable with trees. Firstly, notice that learning such shorter hierarchical plans is worthwhile. Also, notice that if dags are allowed for explanations, a hierarchical plan using several occurrences of the same node will be explained. Because the learning methods used focus primarily on the unexplained parts of the interaction, the explanatory dags offer few¹⁵ opportunities for learning
- some high level goals where the repeated execution of the tasks is important must not be realized through a dag structure, and consequently, the

¹⁵ It is possible to learn from a dag that explains an interaction, for instance, by noticing that it is a dag (instead of some expected tree) and by building the corresponding goal rule(s). However, a special learning method must be designed to this end.

underlying interaction must not be explained by such a structure. For instance, the goal *apply 3 layers of paint on the chair* must not be realized by executing only once the action *paint the chair*.

As explained before, the learning system proposes new knowledge by completing partial explanations; the following notions are used to identify those parts of the interaction that are not explained by the current domain theory.

Definition:

An action that is not part of a goal graph is said to be *unexplained* by the goal graph. A goal rule R is *partially realized* by an interaction if only parts of its precondition, hierarchical plan or postcondition are satisfied.

2.5.2 Causal links determine contributors to the goals

Building goal rules requires identifying which actions (and which tasks, in general) participate in the realization of desired properties, these properties being either preconditions of other tasks, or desirable properties of some state. To identify these tasks, it is necessary to specify the relationship between their effects and the conditions they achieve. These relationships are represented through causal links, as described in the next definitions.

Definition:

A *proof tree* for a literal c is a tree whose root is c and showing how c is proven in terms of other literals, using the current set of axioms AXMS.

Definition:

Given a literal c and a proof tree PT for c , a literal u *contributes* to c , denoted $u \rightarrow c$, if u is a node in PT .

Definition:

Let $GG = \langle N = \{n_1, n_2, \dots, n_k\}, PC \rangle$ be a goal graph. A *causal link* for GG is a 4-tuple $\langle n_i, e, c, n_j \rangle$, where

- (1) $n_i < n_j$
- (2) $e \in \text{postcond}(n_i)$
- (3) $e \rightarrow c$
- (4) $c \in \text{precond}(n_j)$.

Furthermore, for each (negative) literal u needed for the precondition of the task n_j and assumed under the closed-world assumption, the causal link $\langle a_0, u, u, n_j \rangle$ is created. The set of all causal links for GG is denoted $CL(GG)$, or simply CL .

The relationship between the postcondition of actions to the property Q_0 needs to be described. Because causal links describe only relationships between the postcondition and the precondition of tasks (not goals or properties), we introduce a special action to the interaction, and special causal links, as follows (see next figure):

Definition:

For a particular goal $G = \langle P_0, Q_0 \rangle$, a *special action* a_G is added to the goal graph $GG = \langle N, PC \rangle$ to represent the realization of G . The precondition and postcondition of action a_G are both Q_0 . This action is inserted right after the last action a_{iG} achieving Q_0 : $a_{iG} < a_G < a_{iG+1}$.

The causal link $\langle n_i, e, c, a_G \rangle$ is created for each node $n_i \in N$ such that

- (1) $n_i < S_{iG}$
- (2) $e \in \text{postcond}(n_i)$
- (3) $e \rightarrow c$
- (4) $c \rightarrow Q_0$.

For instance, using the example of the previous section, and the semantics

$$\begin{aligned} \text{precond}(a_1) &= \neg p & \text{postcond}(a_1) &= p \wedge q \\ \text{precond}(a_2) &= p \wedge \neg r & \text{postcond}(a_2) &= p \wedge r \\ \text{precond}(n_8) &= \neg p \wedge \neg r & \text{postcond}(n_8) &= p \wedge q \wedge r, \end{aligned}$$

we obtain the structure and the causal links illustrated in Figure 2-6.

Figure 2-6. A goal graph with causal links (extended from Figure 2-4). The literal x remains unchanged through the states because it is not modified by the actions.

To conclude the presentation, let us introduce a characterization of actions in terms of their roles in the achievement of goals. These notions are used in Chapter 3 to describe LEADER.

Definition:

An action is *directly supporting* a condition Q if some effect of it contributes to Q. An action is *indirectly supporting* a condition Q if it is directly supporting Q or if it contributes to the precondition of some action that is directly supporting Q.

CHAPTER 3

Learning from agent interactions

Learning by observing an expert at work is a natural way to acquire expertise. This type of learning is most convenient in situations where the expert does not want to be disturbed and the apprentice cannot validate immediately his or her (or even its) assumptions and findings through actual experience. These situations are the ones that motivate the realization of the present learning system.

General concepts associated with this type of learning have been introduced in Chapter 2. This chapter describes LEADER (LEArning DEsign Rules), a system that learns hierarchical plans by analyzing agents' interactions; the main concepts and contributions are summarized in Section 3.3.

3.1 Description of LEADER

LEADER consists of three components (see next figure): the *Action Executor* producing a sequence of states from an agent interaction (an ordered sequence of actions), the *Interaction Analyzer* that builds structures to explain the interaction, and the *Learner*, the component that generates new knowledge to make the theory more complete. In addition, another module, the *Interaction Observer* (drawn outside LEADER), is required to provide LEADER with the ordered sequence of actions executed by the agent. Each one of these four components are described thereafter in corresponding subsections.

Figure 3-1. Architecture of LEADER.

3.1.1 The Interaction Observer

The purpose of this module is to produce the ordered sequence of actions corresponding to the agent interaction. Although the detailed description of this task depends largely on the world being modelled and on the representation used for its representation, it is possible to generically describe the module.

The Interaction Observer's task is:

Given:

- a world, an agent, a context and a representation

Build:

- I, the agent interaction consisting of an ordered sequence of actions

The effects of each action are not recorded by the Interaction Observer; only the sequence of these actions is (commands instantiated with world objects). In fact, the Interaction Observer may be unable to observe these effects.

Method:

The method depends on the world, the agent and the representation

3.1.2 The Action Executor

This module executes the interaction produced by the Interaction Observer. While the latter only observes the agent, without accounting for the effects of actions on the state, the Action Executor executes in sequence each action of the interaction, and records for each one the state produced after its execution.

To compute the effects of these actions, the Action Executor uses the definition of commands (from the domain theory included in the existing knowledge), and so it contains a mechanism for simulating agent interactions in the world¹⁶.

The Action Executor contains no mechanism to check the executability of a given action, as such a verification is useless. Indeed, because each action is recorded by the Interaction Observer *after* its *real* execution by the agent, the world itself implicitly checks feasibility.

¹⁶ The Action Executor does not "simulate" the agent behaviour in the sense that it can replace the agent in the world and accomplish the same task (i.e., manipulate world objects). Instead, the effects of actions are symbolically computed and stored to form states, in the same manner a planner does not manipulate physical blocks.

In robotics, for instance, if a robot was able to execute the action *put box1 on table* in a given state, it is unnecessary to verify within the Action Executor either if the objects *box1* and *table* exist, or if *box1* can physically be put on *table*. In database design, where the human expert (agent) used a software tool to develop an entity-relationship model, the software implicitly made this type of verifications when it validated the human interaction.

Finally, the Action Executor also records causal links for the precondition of the actions, that is to say, causal links of the form $\langle a_i, e, c, a_j \rangle$. These causal links are used by the next two components (the Interaction Analyzer and the Learner) to trace back which actions were needed to achieve desired conditions in states. Because no goal rules are used, the Action Executor does not identify causal links for hierarchical plans (the Interaction Analyzer does it).

In summary, the Action Executor's task is:

Given:

- a domain theory and a context
- I , the interaction provided by the Interaction Observer
- S_0 , the state at the beginning of the interaction (obtained by executing the special action a_0)

The state is described as a set of properties. This first state may be empty (e.g., at the beginning of the first session in the design of an ER model) or it may contain initial properties (e.g., in planning, the objects and relationships existing in the initial world state, or, in ER model design, the entities and relationships obtained from a previous design session and saved for the next session).

Build:

- the sequence of contiguous states
- the causal links for the actions

Method:

Apply in sequence the actions of the interaction I to create the sequence of states $(S_0, S_1, \dots, S_k)$.

For each action, create a causal link for each literal in its precondition.

3.1.3 The Interaction Analyzer

The purpose of this module is to determine which parts of an interaction are explained by the current set of goal rules. No learning occurs during this phase: goal rules are simply used to explain the interaction. Only fully satisfied hierarchical plans are retained (Section 2.5).

A forward chaining (bottom-up) approach is taken: the first pass identifies goals that are achieved using solely actions, the second pass identifies goals achieved using actions or goals previously identified, and so on, until no more new goals can be identified. This approach is more appropriate than backward chaining (top-down) methods such as EBL [DeJong and Mooney 1986] because

- several goals may have to be decomposed into subgoals
- these goals are not known in advance
- these goals may be at any level of the hierarchy

- a subgoal may be used several times in the goal graph, and in the proposed approach, it is decomposed only once.

The method used to build the goal graph is as follows:

Given:

- I, the interaction provided by the Interaction Observer
- the sequence of contiguous states built by the Action Executor
- the goal rules theory providing all known goal rules

Build:

- the goal graph GG showing all proof trees for realized goals.

Method:

1. {Initialize GG}

Make each action in I a leaf in GG, that is, a trivially achieved goal.

2. {Execute one bottom-up cycle}

For each rule GR in the goal rules theory achieving a goal G, verify that the related hierarchical plan H is realized, that is, that each subtask of H corresponds to a node in GG.

In addition, two constraints are imposed on the goal tree: the tasks must appear in the same order in the interaction as specified in the goal rule, and the distance (defined in Section 2.2) between the first and the last direct action is no more than a specified value¹⁷.

¹⁷ In LEADER, the current value for the distance is 10 actions.

The latter constraint is established to cut down the search time; it is based on the assumption that not too much time elapses between the moment where the agent begins to achieve a goal and the moment where the goal is achieved, that is to say, the agent has a focussed mind. Note that this constraint still allows a given action to be used to achieve several goals. Furthermore, the value of the maximal distance can be set to whatever is appropriate for a given domain, or even for a given agent.

If the hierarchical plan H is realized, determine the first state S' (according to the order specified by the sequence of actions) modified to achieve G , that is, the state occurring before the first action done to achieve G .

Verify that the precondition of the goal rule is satisfied in S' (although the goal rule and agent interaction guarantee that the sequence of actions can be executed, it must be verified that this particular goal rule – along with its precondition – was actually used in the current situation).

If it is so, then G is satisfied, and because the hierarchical plan guarantees the goal achievement when the precondition is satisfied, the postcondition does not have to be verified. So, create a new node n corresponding to G and add it to GG , and add a parent-child link from n to each node in GG contributing to realize the hierarchical plan H . Also, create a causal link $\langle n_i, e, c, n \rangle$ for each node n_i of GG that has in its postcondition a literal e that contributes to a literal c in the precondition of n .

3. {Loop for all cycles}

If executing step 2 did not produce a new node in GG, then stop.

Otherwise, go to 2.

3.2 Learner: the learning component of LEADER

An interaction is either totally or partially explained by the current set of goal rules. In the first case, no learning can occur¹⁸. In the second case, learning can occur to explain the unexplained parts. For instance, new goals and new hierarchical plans can be inferred. We present here how the Learner identifies gaps in the theory and provides proposals to enhance it.

Failing to justify an action of the interaction is the most direct observation of a lack in the theory. To complete the theory, an explanation for this action must be found. According to the way explanations are defined, either a goal rule that uses this action is built, or a causal link is. As we will see, adding a causal link requires building a new goal rule, and vice versa.

Let us see how the Learner learns from the unexplained actions. The following 4-step method is used:

- identify a goal for which the actions contribute to the realization
- identify the hierarchical plan realizing this goal
- identify the semantics of this hierarchical plan
- construct the resulting goal rule.

¹⁸ In fact, some learning is still possible in this case, such as EBL, or learning the utilization frequency of strategies and goals. However, we are not interested here in learning this type of knowledge.

The four following subsections address these problems.

3.2.1 Identifying a candidate goal

A given interaction may achieve a large number of goals. Mechanisms must be used to cut down the number of goals and goal rules proposed by the Learner; these filtering mechanisms must ensure that good candidates are not thrown away.

The easiest way to identify good candidate goals is by having the goal provided by the agent as a component of the training instance. This is the approach taken in typical learning systems such as [DeJong and Mooney 1986], [Segre 1988], [Mitchell and Mabadévan 1990] and [Hammond 1989].

The main limitation of this approach is precisely that the goal must be provided: without this focus of interest, learning (if possible) is restricted. An alternate way to identify goals is by having a predetermined list of *meaningful* goals to look for, a list usually provided by experts when the domain theory is created¹⁹. The major problem with this approach is that the list is static: the searching method never looks for goals that are not in the list, and thus never learns new goals.

A more useful and challenging approach consists in allowing the learning system to create new goals when those present in the domain theory are inadequate to explain the agent's behaviour. For instance, a promising method for identifying new candidate goals might rely on looking at consequences and prerequisites of unexplained actions or partially realized tasks. Indeed, failures to explain an interaction occur either because the hierarchical plan used by the agent is unknown, or because the goal is unknown (or both). So,

¹⁹ This notion of "goal meaningfulness" is discussed in Section 3.2.4 as an approach to refine the conditions defining the semantics of tasks.

focussing on unexplained parts is a good heuristic approach to identify new goals. In fact, this heuristic can also be used to speed up the search for meaningful goals through the interaction, when the list of meaningful goals is provided.

The last two approaches (searching in a list of goals and building new ones) were investigated for implementation in LEADER. In fact, as described next, both are unified through a general classification derived from the notion of achievement vector introduced in Section 2.5.

The approach taken in LEADER is neither the creation of new goals nor a search through a list of provided goals: it is a hybrid of the two. As described in detail below, although a list of goals is provided in the domain theory, LEADER does not directly process it: LEADER first looks for goal rules that are partially realized to get clues about good candidate goals. Then, it uses these candidate goals to propose new goal rules.

So, the general method described next for proposing goals also provides information on what part of the interaction achieves them. It uses the notions of achievement vector and α -satisfaction defined in Section 2.5.

The method (illustrated by Figure 3-2) is the following (additional details are provided thereafter):

Given:

- an interaction I
- the goal graph corresponding to I

2

PM-1 3½"x4" PHOTOGRAPHIC MICROCOPY TARGET
NBS 1010a ANSI/ISO #2 EQUIVALENT

PRECISIONSM RESOLUTION TARGETS

Build:

- a good candidate goal $G' = \langle P', Q' \rangle$
- a sub-interaction I' of I realizing G'

Method:

1. {Find a goal rule that is partially realized}

Process each goal rule of the domain theory to find one (say, $G \Leftarrow \langle P, H, Q \rangle$, corresponding to the goal $G = \langle P_0, Q_0 \rangle$) that is partially satisfied.

2. {Determine a sub-interaction I'' corresponding to this goal rule}

Because at least one of the tasks of the goal rule is realized, some actions of I can be identified as contributing to the realization of the goal rule.

Let I'' be the sub-interaction of I delimited by the first and the last of these contributing actions. Let S_P and S_Q be the states occurring before the first action and after the last action, respectively.

3. {Compute AV}

Compute the logical value of the conditions P_0 , Q_0 , P and Q in the appropriate states S_P and S_Q to get the achievement vector $AV = (P(S_P), P_0(S_P), Q(S_Q), Q_0(S_Q))$.

4. {Determine G' and I' }

According to the meaning of the achievement vector (described below), and using the partial realization of the goal rule, apply appropriate methods to propose the goal G' , along with a sub-interaction I' of I (delimited by the states $S_{P'}$ and $S_{Q'}$) that achieves G' .

Figure 3-2. Illustration of the general algorithm.

Before completing the algorithm by describing the meaning of the nine possible achievement vectors (Section 2.5) and their corresponding effects on the proposal of G' and I' , some comments on this generic method are needed.

First, and most importantly, the above algorithm may fail to provide the *correct* goal (the *correctness* refers to the goal intended by the agent). Such a situation may occur, for instance, when the domain theory contains no goal rules similar to the one used by the agent.

Step 1 requires that at least one task of the goal rule is realized; otherwise, too many candidate goal rules would be proposed uselessly (and inefficiently) as a basis for learning. Similarly, the goal rule must not be totally realized so that it provides learning opportunity.

Finally, due to the way I' is defined, only its first and last actions are guaranteed to contribute to G' , that is to say, $P'(S_P) = \text{true}$ and $Q'(S_Q) = \text{true}$. So, I' may contain actions irrelevant to G' , and a filtering mechanism is required to remove them. This topic is discussed in Section 3.2.2.

Now, the meaning and the use of the achievement vectors are introduced, then two heuristic learning methods are proposed to refine Step 4 of the above generic algorithm.

Achievement vectors are grouped into three disjoint classes:

Class 1: situations where both $P_0(S)$ and $Q_0(S')$ are *true*

Class 2: situations where $P_0(S) = \text{true}$ and $Q_0(S') = \text{false}$

Class 3: situations where $P_0(S) = \text{false}$.

Class 1 has four possibilities for situations where both $P_0(S)$ and $Q_0(S')$ are *true*: $AV = (_, \text{true}, _, \text{true})$. In these cases, G is achieved by I' . In other words, the interaction I' contains an expansion of a hierarchical plan H' realizing G , although the hierarchical plan used by the agent cannot be explained by the current knowledge; the proposed goal G' is then G itself. This case offers significant potential for learning; it is also the easiest because the goal is given for free. Let us have a closer look at these four possibilities:

If $AV = (\text{true}, \text{true}, \text{true}, \text{true})$, both the goal G and the conditions of the hierarchical plan used are satisfied. No new goals are proposed, but the new hierarchical plan corresponding to the interaction will be learned (see Section 3.2.2).

If $AV = (\text{true}, \text{true}, \text{false}, \text{true})$ or $(\text{false}, \text{true}, _, \text{true})$, the conditions of the hierarchical plan used are not satisfied, and the goal G is. Not only a new

hierarchical plan can be learned for G , but this hierarchical plan corresponds probably to the achievement of a specialization of G (similar to the conditions of H) that is worthwhile to learn.

Class 2 pertains to situations where $P_0(S) = \text{true}$ and $Q_0(S') = \text{false}$. It contains two members: $AV = (_, \text{true}, \text{false}, \text{false})$. Here, the goal G is not achieved, although it is feasible, so the interaction I' does not correspond to the achievement of G . In these cases, it should be more productive to consider the possible presence of a new goal G' , similar to G .

Finally, Class 3 contains three possibilities corresponding to situations where $P_0(S)$ is false: $AV = (\text{false}, \text{false}, _, _)$ (the vector $(\text{false}, \text{false}, \text{true}, \text{false})$ is not obtainable). Here, the goal G is not feasible (because $P_0(S) = \text{false}$). Again, it should be worthwhile to explore the feasibility of a new goal.

To complete the refinement of Step 4 of the above generic algorithm, we now present two heuristic methods for proposing a candidate goal G' along with pertinent information on its achievement. This information is used in the next sections to derive the new goal rule.

The heuristic methods can be summarily described as follows (Figures 3-3 and 3-4).

Figure 3-3. Illustration of Learning Heuristic 1.

Learning Heuristic 1 :

learning a new hierarchical plan for the first task of a goal rule.

This method can be applied when the only missing task of a partially realized goal rule is the first one (leftmost)²⁰. The assumption is the following: if the agent executed all but one of the tasks of a known goal rule, the subgoal of the missing task was probably achieved (because required in the known goal rule) in a way different than expected. So, there might be a learning opportunity for acquiring a new hierarchical plan to achieve the subgoal in question.

In more detail, the method is as follows:

- identify the subgoal $G' = \langle P', Q' \rangle$ corresponding to the missing subtask
- starting from the state corresponding to the first (leftmost) direct action appearing in the partially realized goal rule, go backward (to the left) to find a state $S_{Q'}$ where Q' is satisfied
- starting from the state preceding $S_{Q'}$, go backward again to find a state $S_{P'}$ where P' is satisfied.
- the result is a goal $G' = \langle P', Q' \rangle$ with the boundaries ($S_{P'}$ and $S_{Q'}$) of an interaction I' that achieves it.

²⁰ Although other variations can be used (for instance, replacing the rightmost task of a partially explained goal rule), only this one was implemented in LEADER because the purpose was to illustrate the feasibility of the approach.

Again, to cut down the search time while processing the states backward, a maximal search distance is specified²¹.

Figure 3-4. Illustration of Learning Heuristic 2.

Learning Heuristic 2 :

learning a new hierarchical plan for a goal when the corresponding goal rule is partially satisfied

This method can be applied when the hierarchical plan H' of a goal rule $G' \Leftarrow \langle P', H', Q' \rangle$ is only partially realized, although the corresponding goal $G = \langle P_0, Q_0 \rangle$ is realized. This approach corresponds to Class 1 of achievement vectors. The method is the following:

- identify the subgoal $G = \langle P_0, Q_0 \rangle$ corresponding to the partially realized goal rule
- starting from S_P , the state corresponding to the first (leftmost) direct action appearing in the partially realized goal rule, go backward (to the left) to find a state $S_{P'}$ where P_0 is satisfied

²¹ In LEADER, the current value for the maximal distance is 6 actions, for each of the two backward searches.

- starting from S_Q , the state corresponding to the last (rightmost) direct action appearing in the partially realized goal rule, go forward (to the right) to find a state $S_{Q'}$ where Q_0 is satisfied
- the result is a goal $G = \langle P_0, Q_0 \rangle$ with the boundaries ($S_{P'}$ and $S_{Q'}$) of an interaction I' that achieves it.

As for Learning Heuristic 1, a maximal search distance is specified for this heuristic.

Now that a candidate goal G' has been identified along with an interaction I' that achieves it, the next task is to extract from I' the actions relevant to G . This is the topic of the following section.

3.2.2 Identifying the hierarchical plan realizing a goal

This section describes the second step of the process used to learn heuristic rules from unexplained actions, that is to say, the identification of the hierarchical plan involved in the realization of a goal. Here, only the tasks composing the hierarchical plan are of interest; the precondition and postcondition of the hierarchical plan are not determined (this is the topic of the next section).

We assume that a candidate goal, say $G = \langle P_0, Q_0 \rangle$, is provided, along with an interaction $I = (a_{i1}, a_{i1+1}, \dots, a_{i2})$ achieving it. According to the previous section, all is assumed about I is that its first and last actions are needed to achieve the goal, that is to say, $P_0(S_{i1-1}) = \text{true}$ and $Q_0(S_{i2}) = \text{true}$ (remember that the action a_{i1} is executed in the state S_{i1-1} , and that the action a_{i2} produces the state S_{i2}).

The method presented in this section does the following:

- identification of relevant actions
- identification of relevant high level tasks
- creation of a general (un-instantiated) hierarchical plan.

An simple way to identify relevant actions, in cases where the goal rule is not realized only because some of its tasks are not realized, is to search through the unexplained actions by applying a goal-directed version of the method called *loose-ends heuristic* by cognitive scientists [Lewis 1988]:

"if an action cannot be explained, and if an expected command is not realized, then assume that the unexplained action is linked to the unrealized command."

To obtain a better match between unrealized tasks and unexplained commands than the plain version of the heuristic, the goal-directed version would compare the semantics of tasks and commands, on the basis of similarity of purposes (i.e., goals), as in [Genest and Matwin 1990]. Furthermore, because the interaction is fully instantiated, and because the unrealized part of the goal rule is partially instantiated by the instantiation of the explained part, the matching (if found) is likely to be appropriate.

The method used in LEADER is more similar to goal regression: the identification of relevant actions of the interaction makes large use of causal links. It is done in three steps:

- find DSA, the set of *directly supporting actions*, that is to say, the actions that contribute directly to the property Q_0

- find ISA, the set of *indirectly supporting actions*, that is to say, the actions that contribute either to a direct or to an indirect supporting action
- find SA, the set of *supporting actions*: $SA = DSA \cup ISA$.

Using the definitions introduced in Section 2.5, direct supporting actions DSA are given by

$$DSA = \{a_i \in I \mid \langle a_i, e, c, a_G \rangle \in CL, a_{i1} \leq a_i \leq a_{i2}\}.$$

Note that although causal links are defined for both actions and higher level tasks, DSA retains only the actions pertinent to the goal. Also, the set may contain too many actions. For instance, if the actions a_i and a_i' both have the literal c as their unique postcondition, they both will be in DSA, although only one of them would be sufficient.

To avoid as much as possible duplicate actions that contribute to the same literal in Q_0 (i.e., the same node in the proof tree PT of Q_0), the following method can be used:

"to prove a node of PT, consider firstly actions that are already used to prove other nodes of PT, instead of a new action."

Even with the proposed method, DSA may still contain too many actions, because the method does not guarantee that DSA will be the smallest possible set²²: DSA depends on the order in which actions are considered.

So, either an optimization step is desirable – although not necessary for the correctness of the goal rule eventually produced, or a method to find a better DSA in the first place is

²² Although the size of DSA is not necessary a good measure of the appropriateness of the goal rule produced, it helps to have a shorter and more readable goal rule. However, a smaller DSA may lead to a larger unexplained part of the interaction, specially when the agent did not act efficiently.

required (e.g., minimization of the size of DSA by a tabloid method based on dynamic programming).

Similarly, ISA, the set of indirect supporting actions, is determined using causal links by the following (ISA_r is the set of actions that achieves literals of Q_0 in $r+1$ steps):

$$ISA_0 := DSA$$

$$ISA_{r+1} := \{a_j \in I \mid \langle a_i, e, c, a_j \rangle \in CL, a_{i1} \leq a_j, a_j \in ISA_r\}$$

$$ISA := \cup ISA_r, r = 1, 2, \dots, \infty.$$

Note that ISA does not necessarily contain DSA, because the union over r starts at 1.

In similarity with the set DSA, ISA also may contain unnecessary elements. For instance (see Figure 3-5), if the literal u is true in S_{i1-1} , if the precondition of both a_{i1} and a_{i1+2} is u and their effect is $\neg u$, if the precondition of a_{i1+1} is $\neg u$ and its effect is u , and if $Q_0 = \{\neg u\}$, then a_{i1} and a_{i1+2} will be computed as elements of ISA, although they can be removed from the sequence of actions achieving Q_0 (only a_{i1+1} is required). According to the terminology used in [Chapman 1987], this is a case of an *establisher-clobberer* pair of actions.

Figure 3-5. An establisher-clobberer pair of actions. The effect of an action is undone by one that comes after it.

Again, to obtain a smaller ISA, the following method can be used:

"after computing ISA_r , remove from it any action a_k whose effects are all already provided in S_{i1-1} and that are not removed by other actions executed between a_{i1-1} and a_{k-1} ."

Again, even with the proposed method, ISA may contain too many actions, because the method does not guarantee that ISA will be the smallest possible set. Again, an additional process is desirable to optimize the set ISA produced.

At this point, DSA and ISA being identified, the supporting actions SA realizing the goal are obtained by taking the union of DSA and ISA²³. Let $I' = (b_1, b_2, \dots, b_k)$ be the corresponding sub-interaction. Although I' constitutes an interaction achieving G , it might not be the most appropriate one (readable and informative) because it is flat: the high level tasks are expanded in I' and are lost. This is the purpose of the second step.

To recover these high level goals, the goal graph is searched for trees whose leaves are actions of I' . These actions are then replaced with the root of these trees to produce a more structured hierarchical plan H' . This process is repeated on H' until no more actions can be replaced with high level tasks. The resulting H' is an instantiated hierarchical plan realizing the goal G .

The last step is the generalization of H' . This is done simply by applying the inverse of the instantiation used to convert commands into actions. Note that a more powerful

²³ The set SA can also be determined by computing for each literal c of Q_0 the set $SA(c)$ of supporting actions of c (by computing the direct supporting actions of c , and, recursively, the supporting actions for these supporting actions), and by taking the union of all $SA(c)$. However, a filtering mechanism must again be used to remove unnecessary actions.

generalization technique might be used; however, the technique used must ensure that the hierarchical plan obtained still achieves the goal G.

3.2.3 Determining the semantics of a hierarchical plan

The previous sections have described how the goal and the hierarchical plan are identified. This section addresses the problem of determining the semantics of a given hierarchical plan, that is to say, determining its precondition and its postcondition.

Recalling the definition of the precondition and the postcondition of a hierarchical plan H (Section 2.3):

$\text{precond}(H)$

= the weakest condition guaranteeing the executability of H

$\text{postcond}(H)$

= the set of conditions obtained by executing H on $\text{precond}(H)$,

the two problems can be formulated as:

- Problem Pb_1 :

given a hierarchical plan, determine what must have held before its execution to allow its execution

- Problem Pb_2 :

given a condition and a hierarchical plan executed over this condition, determine what holds after the execution.

The first task is known as the *projection problem* [Elkan 1989]; the second one is a version of the *retrojection problem* [ibid] (the determination of what must have held before the execution of a hierarchical plan, given what holds after it).

The next sub-sections address problems Pb_1 and Pb_2 . Sub-section 3.2.3.1 describes an efficient solution for the propositional case in the absence of inferential literals, and Sub-section 3.2.3.2 presents a method for solving the problems in the general case. More details can be found in [Pelletier and Matwin 1992].

3.2.3.1 The non-inferential propositional case

As a first attempt to address problems Pb_1 and Pb_2 , the similarity between goal rules and programming language statements suggested the investigation of the techniques used for program analysis. This section describes the similarities and methods that lead to an efficient solution for the non-inferential propositional case. Additional comparisons between LEADER and program analysis techniques are presented in Section 5.1.

The basic correspondence is established via the concepts of statement and task, as shown in Table 3-1. In particular, both a task and a statement are actions or sequences of actions that can be executed in a given state, and whose execution produces another state.

Statements	Tasks
assignment, procedure call	single task
block H: $T_1 ; T_2 ; \dots ; T_n$	hierarchical plan H: $(T_1, T_2, \dots, T_n)$
conditional branching H: if C then H_1 else H_2	hierarchical plan H: $H \Leftarrow \langle P_1, H_1, Q_1 \rangle$ $H \Leftarrow \langle P_2, H_2, Q_2 \rangle$, where $C \Rightarrow P_1$
while E do T	repeated task: $\text{repeat}(E, T)$ ²⁴

Table 3-1. Programming concepts vs concepts used in LEADER.

The concept of semantics defined in Section 2.3 shares similarities with concepts used in data-flow analysis [Aho, Sethi et al. 1986]. In this domain, the semantics of programs (ordered sequence of statements, where a statement may itself be composed of statements) is specified by using *data-flow information*. This information is usually provided by specifying four sets for each statement H:

$\text{gen}[H] = \{\text{information that is generated by } H\}$

$\text{kill}[H] = \{\text{information that is invalidated (killed) by } H\}$

$\text{in}[H] = \{\text{information that is present (true) when } H \text{ begins its execution}\}$

$\text{out}[H] = \{\text{information that is present (true) when } H \text{ terminates its execution}\}$

These sets correspond to the sets $\text{ADD}(H)$, $\text{DEL}(H)$, $\text{precond}(H)$ and $\text{postcond}(H)$, respectively. When LEADER's preconditions and postconditions are restricted to propositional logic, the similarity between task semantics and data-flow information is even

²⁴ We investigated the idea of having repeated hierarchical tasks; however, this construct was not implemented in the current version of LEADER.

greater. For instance, the relationships between computer programs and hierarchical plans becomes apparent with the search for *available expressions* [Aho, et al. 1986] (Table 3-2):

An expression e containing variables x_i 's is available at line (or block) n if every path from the first line of the program to line n evaluates e , and after the last such evaluation prior to reaching n , there are no assignments to the x_i 's.

Availability of expressions consisting of only one variable corresponds to the truth value of literals, and availability of more complex expressions corresponds to the truth value of properties.

Statements	Available expressions
BEGIN	none
1: $b := 0$	b
2: $c := 0$	b, c
3: $d := 0$	b, c, d
4: $a := b+c$	$a, b, c, d, b+c$
5: $b := a-d$	$a, b, c, d, a-d$
6: $c := b+c$	$a, b, c, d, a-d$
7: $d := a-d$	a, b, c, d
END	a, b, c, d

Table 3-2. Available expressions for a program. The expressions available after the execution of the n first statements are shown after each statement.

The following is a detailed analysis of the relationship between the computation of available expressions and the determination of semantics of hierarchical plans.

By defining the following sets for each block of statements B (these definitions refine the set definitions given above):

$\text{gen}[B] = \{ \text{expressions that are computed by } B \}$

$\text{kill}[B] = \{ \text{expressions where at least one variable is recomputed by } B, \\ \text{and that are not later on recomputed in } B \}$

$\text{in}[B] = \{ \text{expressions that are available at the beginning of } B \}$

$\text{out}[B] = \{ \text{expressions that are available after the execution of } B \},$

corresponding equations can be derived [Aho, et al. 1986]:

$\text{out}[B] = (\text{in}[B] - \text{kill}[B]) \cup \text{gen}[B]$

$\text{in}[B_1] = \{ \},$ where B_1 is the first block of the program

$\text{in}[B] = \bigcap \text{out}[P],$ where $B \neq P$ and P runs over predecessors of B .

The approach proposed in [Aho, et al. 1986] for computing the sets starts with U , a superset of all possible available expressions, and removes from it the superfluous elements. In fact, U is the set of all expressions appearing on the right of any statement of the program. By analogy with hierarchical plans, U corresponds to the set of all conditions used by tasks.

Using these notions, an algorithm for computing $\text{in}[B]$ and $\text{out}[B]$ for each block B can be derived ([ibid] p.625):

```

in[ B1 ] := { }
out[ B1 ] := gen[ B1 ]
for B ≠ B1 do out[ B ] := U - kill[ B ]
Change := true
while Change do begin
    Change := false
    for B ≠ B1 do begin
        in[ B ] :=  $\cap$  out[ P ], where P runs over predecessors of B
        OldOut := out[ B ]
        out[ B ] := (in[ B ] - kill[ B ])  $\cup$  gen[ B ]
        if out[ B ] ≠ OldOut then Change := true
    end
end.

```

Figure 3-6. Algorithm for computing the *in* and *out* sets of available expressions, for each block B of a program.

By analyzing the above algorithm and by establishing a correspondence between available expressions and truth value of properties, following comparison becomes apparent:

- available expressions provide information only about what becomes available (that is to say, *true*); In comparison with postconditions that must also specify what is guaranteed to be *false* (that is to say, not available), the above algorithm must be modified to produce the complementary set. For instance, two separate sets can be used, one for available expressions (true properties) and one for unavailable ones (false properties).
- the algorithm provides more information than what is needed: it produces information about each block of the program. In the case of a hierarchical plan H, only the information about the whole task is required, that is to say,

only $\text{in}[H]$ and $\text{out}[H]$ are required, and these values are provided by the algorithm because they are the right part of the following equations:

$$\text{in}[H] = \text{in}[B_1]$$

$$\text{out}[H] = \text{out}[B_n], \text{ where } B_n \text{ is the last block (task).}$$

- in the computation of available expressions, there are no notion of executability, a notion that is the basis for determining the precondition of hierarchical plans in LEADER. In computer programs, the executability corresponds to the absence of run time errors, such as division by zero, square root of a negative number, use of an uninitialized variable (not always detected) or need for an unavailable resource.

In fact, these errors are *run time* errors precisely because they cannot be detected by an a priori analysis: the task is undecidable. If the language used to describe hierarchical plans is as powerful as a programming language, then finding preconditions is also undecidable. In such a situation, no guaranties can be given on the executability of hierarchical plans, and failures can occur during execution, thus requiring replanning.

- no expressions are available before the execution of the first block. To guarantee the correctness of its execution, the first block must correspond to a variable initialization (using only constants in the right part, and so, using no expressions already computed).

In contrast, the first task of a hierarchical plan requires that its precondition is satisfied. This precondition may refer to the conditions established in the preceding state (for instance, in the initial state produced by the default

action a_0), or it may be *true*, that is to say, the first task is always executable. To take into account the fact that in general hierarchical plans have a non-trivial precondition, the first instruction of the above algorithm can be replaced with

$$\text{in}[B_1] = \text{precond}(H) \text{ }^{25}.$$

Note that this modification raises the problem of computing $\text{precond}(H)$ at the very beginning of the algorithm (probably by doing a similar analysis as done by the above algorithm).

- programming languages make use of loops (handled by the above algorithm); hierarchical plans do not ²⁶.

As loops are not within the scope of our current approach, the algorithm can be simplified by replacing the while loop with an iteration that proceeds from the first block to the last one, and by removing the intersection over the predecessors of blocks (because each block now has only one predecessor). The new algorithm (below) is thus more efficient than the previous one (linear in the worst case):

²⁵ Note that the instruction " $\text{in}[B_1] = \text{precond}(B_1)$ " cannot be used because the execution of the rest of the task (i.e., $(B_2, B_3, \dots, B_n)$) may require a condition that is not present in $\text{out}[B_1]$ (neither in $\text{precond}(B_1)$ nor generated by B_1).

²⁶ As mentioned above, the current version of LEADER does not support strategies containing loops.

```
in[ B1 ] := { }  
out[ B1 ] := gen[ B1 ]  
for i := 2 to n do begin  
    in[ Bi ] := out[ Bi-1 ]  
    out[ Bi ] := (in[ Bi ] - kill[ Bi ]) ∪ gen[ Bi ]  
end
```

Because of this similarity between the semantics of conditions and the definition of the *in*, *out*, *kill*, and *gen* sets, the methods used to determine the latter sets can be adapted to compute the former semantics.

To do so, we designed TWEAK⁺, a variation of the TWEAK representation introduced by [Chapman 1987]. TWEAK⁺ is a STRIPS-like representation that uses the traditional DEL and ADD lists [Fikes and Nilsson 1971] to define actions, as well as lists (conjunction of literals) for the precondition and for the postcondition, and that allows these four lists to contain negative literals.

Using this representation, algorithms in the form of boolean equations (equations involving only sets and elementary operations over sets) were derived to produce an efficient solution to problems Pb₁ and Pb₂ using a bit vector representation for the conditions [Aho, Hopcroft et al. 1974], that is to say, using vectors whose elements are bits indicating whether the corresponding elements are present or absent from the current state. The resulting algorithm is summarized by the following:

Given: two tasks T_1 and T_2 along with their respective descriptions
 (P_1, D_1, A_1, Q_1) and (P_2, D_2, A_2, Q_2) (for the precondition, DEL list,
 ADD list and postcondition)

the hierarchical plan $H = (T_1, T_2)$

Find: the description of H in the form (P_H, D_H, A_H, Q_H)

Method: $P_H = P_1 \cup (P_2 - Q_1)$

$A_H = [(A_1 - D_2) \cup A_2] - P_{1,2}$

$D_H = [[(D_1 - A_1) \cup D_2] - A_2] - \neg P_H$

$Q_H = (P_H - D_H) \cup A_H$

where the negation of a set is given by

$\neg \{l_1, l_2, \dots, l_k\} = \{\neg l_1, \neg l_2, \dots, \neg l_k\}.$

We discuss here the properties of the TWEAK⁺ representation and we justify the general approach presented in the next sub-section:

- an efficient solution in the form of bit vector equations is obtained for computing the description (precondition, postcondition, ADD and DEL lists) of hierarchical plans described in the TWEAK⁺ representation (restricted to propositional logic, without inferential literals). The bit vector equations lead to an algorithm that runs in linear time (or even sub-linear time [Aho, et al. 1974]) and linear space, with respect to the number of tasks contained in the hierarchical plans.
- augmenting TWEAK⁺ with inferential literals where axioms are implication formulas in the disjunctive normal form $c \leq p_1 \vee p_2 \vee \dots \vee p_n$ (where p_i 's

are products (conjunctions) of literals) can still be accommodated by the bit vector equations in certain situations. These situations are characterized by the absence of disjunctions in the definition of inferential literals as well as negations of inferential literals in the description of the semantics of actions.

- in other situations (i.e., in the presence of disjunctions) the bit vector equations cannot be applied. Heuristic methods for computing precondition and postcondition of hierarchical plans are proposed. They are described in detail in the next sub-section.

More details about the computation of tasks' semantics can be found in [Pelletier and Matwin 1992].

3.2.3.2 The inferential predicate calculus case

This section presents a method for determining the semantics of a hierarchical plan, giving the semantics of its subtasks, in the general case: literals are predicates, the DEL and ADD lists are not used (we shown in Section 2.4 that the computation of the semantics of a hierarchical plan cannot always be obtained only via DEL/ADD lists which are restricted to the conjunction of primitive literals), and preconditions and postconditions may contain inferential and negated literals. Because DEL and ADD lists are not used, only algorithms for computing preconditions and postconditions are provided.

More precisely, this section includes following results:

- an algorithm for computing the postcondition of a composition of tasks, in the general case

- an algorithm for applying a task under some initial condition, in the general case
- an algorithm for computing the precondition of a composition of tasks, in the general case
- a discussion on the complexity of these algorithms and on their application to simpler cases.

As mentioned earlier, by definition, the computation of the postcondition of a hierarchical plan depends on the knowledge of its precondition. As described next, the determination of the precondition of a task also depends on the computation of the postcondition of other tasks. However, these two mutually calling algorithms never fall into an infinite loop because these other tasks whose postcondition is needed are the subtasks of the initial task, and so the recursive calls apply always on shorter hierarchical plans.

In what follows, we assume that the precondition of a hierarchical plan can be computed, and we concentrate first on deriving the procedure that computes postconditions. The procedure for computing the precondition is described thereafter.

So, let $H = (t_1, t_2, \dots, t_n)$ be a hierarchical plan, with the tasks' semantics $\text{precond}(t_i)$ and $\text{postcond}(t_i)$.

To compute $\text{postcond}(H)$, we use the function $\text{postcond_from}(P, H, Ok)$ (described below) returning the condition obtained by executing the hierarchical plan H under the

initial condition P. Using that function, we have, following the definition of `postcond(H,Ok)`²⁷:

```

function postcond(H, Ok)
    (Returns the resulting condition when applying
     the hierarchical plan H on its precondition)
begin
    Prec := precond(H, Ok)
    if Ok then postcond := postcond_from(Prec, H, Ok)
end

```

Because applying a hierarchical plan H consists in applying in sequence each element of H, and because the task H(i) is applied on the condition obtained after executing the sub-hierarchical plan H(1..i-1), the function `postcond_from(P,H,Ok)` is:

```

function postcond_from(P, H, Ok)
    (Returns the resulting condition when applying
     the hierarchical plan H on P)
begin
    Q := P
    Ok := true
    while (H ≠ ( )) and Ok do
        {postcond_from_task(Q,T,Ok) is like
         postcond_from(Q,H,Ok), except that T is a single task}
        Q := postcond_from_task(Q, H(1), Ok)
        H := H(2..length(H)) {apply the remaining part}
    endwhile
    postcond_from := Q
end

```

²⁷ Some of the algorithms developed below may detect errors during computations. For instance, an important case of error is trying to compute the precondition of a non-executable hierarchical plan. In any case where an error is detected, the boolean parameter Ok returns false to indicate the failure of the algorithm.

Now, we will concentrate on designing an algorithm for the function `postcond_from_task(P, T, Ok)`.

Two similar approaches will be presented and illustrated with the same example. An algorithm will be presented for the second one. The axiomatic theory described in Figure 3-7 will be used to illustrate the computation of the function `postcond_from_task(P, T, Ok)` for both approaches.

Figure 3-7. Axiomatic theory used to illustrate the two approaches. The primitive literals used in the theory are: $p1, p2, p3, p4, p5, p6$. The inferential literals are: $q1, q2, r1, r2$. Lines show how the inferential literals in the top level are defined in terms of primitive ones in the bottom level.

Assume as shown in Figure 3-8 that a task T is executed in an initial state P .

Figure 3-8. An initial condition P and a task T . For more clarity, the conditions are divided into primitive literals and inferential literals.

Note that although $p3$ appears in $\text{precond}(T)$, neither $p3$ nor $\neg p3$ are in $\text{postcond}(T)$; because T is defined this way, it is not known if $p3$ is true or false after its execution. So, T is non-deterministic. Similarly, $q1$ is mentioned in $\text{precond}(T)$, but not in $\text{postcond}(T)$.

Approach 1

This approach is based on the following observation:

"the precondition/postcondition pair describes the global effects of executing a task; so, these effects can be summarized by replacing the precondition with the postcondition in the condition describing the world prior to the execution of the task."

This is illustrated by the next figures.

Figure 3-9. Global effect of a task T : T is seen as a process that transforms its precondition into its postcondition.

Figure 3-10. Graphical representation of Approach 1.

When adapted in the general case, this approach becomes: to find $\text{postcond_from_task}(P, T)$:

- (a) start with P
- (b) identify the precondition of T
- (c) delete this precondition from P
- (d) remove from the result any literal (originating from P) that can no more be proven (the circle on the figure)
- (e) add $\text{postcond}(T)$ to the result²⁸
- (f) remove (the square) from the new result as few literals (originating from P) as possible to remove contradictions (if any)
- (g) delete from it literals (originating from P) that can no more be proven (the rectangle).

²⁸ Because $\text{precond}(T)$ and $\text{postcond}(T)$ have usually literals in common, it is inefficient to remove (in (c)) and to add back (in (e)) these common literals; instead, only different literals should be handled.

This is informally represented by:

$$\begin{aligned} & \text{postcond_from_task}(P, T) \\ &= [(P - \text{precond}(T) - (c)) \cup \text{postcond}(T)] - (f) - (g) \end{aligned}$$

When applied to the previous example, we have:

$$\begin{aligned} & P - \text{precond}(T) - (c) \\ &= \{p1, p2, p3, \neg p4, p6, q1, q2, r1, r2\} - \{p1, p2, p3, q1, q2, r2\} - (c) \\ &= \{\neg p4, p6, r1\} - \{ \} \\ & \quad (\text{because all remaining literals have at least one proof, none are deleted}) \end{aligned}$$

Next, $\text{postcond}(T)$ is added:

$$\begin{aligned} & [(P - \text{precond}(T) - (c)) \cup \text{postcond}(T)] - (f) - (g) \\ &= [\{\neg p4, p6, r1\} \cup \{\neg p1, p2, p4, q2, r2\}] - (f) - (g) \\ &= \{\neg p4, p6, r1, \neg p1, p2, p4, q2, r2\} - (f) - (g) \\ &= \{\neg p4, p6, r1, \neg p1, p2, p4, q2, r2\} - \{\neg p4\} - (g) \\ & \quad \text{because } \neg p4 \text{ is inconsistent with } p4 \\ &= \{p6, r1, \neg p1, p2, p4, q2, r2\} - (g) \\ &= \{p6, r1, \neg p1, p2, p4, q2, r2\} - \{r1\} \\ & \quad \text{because } r1 \text{ was proven only by } \neg p4 \\ &= \{p6, \neg p1, p2, p4, q2, r2\} \\ & \quad r2 \text{ remains because it is still proven by } p2. \end{aligned}$$

Approach 2

This approach is based on the following observation:

"the resulting condition can be obtained by taking $\text{postcond}(T)$ and adding to it what was true before the execution of T , as far as there are no contradictions."

Figure 3-11. Graphical representation of Approach 2.

When adapted in the general case, it becomes: to find $\text{postcond_from_task}(P,T)$:

- (a) start with $\text{postcond}(T)$,
- (b) identify the literals in P that are not in $\text{precond}(T)$ (the other literals of $\text{precond}(T)$ are already handled by T , that is to say, by $\text{postcond}(T)$)
- (c) add these literals to $\text{postcond}(T)$,
- (d) remove from the result any literal (originating from P) that can no more be proven (the circle on the figure)

- (e) delete (the square) as few literals (originating from P) as possible to remove contradictions (if any),
- (f) delete from it literals (originating from P) that can no more be proven (the rectangle).

This is informally represented by:

$$\begin{aligned} \text{postcond_from_task}(P,T) \\ = [\text{postcond}(T) \cup (P - \text{precond}(T))] - (d) - (e) - (f). \end{aligned}$$

The formula is similar to the one used in the approach 1; however, here, there is no verification in the computation of $P - \text{precond}(T)$.

When applied to the previous example, we have:

$$\begin{aligned} P - \text{precond}(T) \\ = \{p1, p2, p3, \neg p4, p6, q1, q2, r1, r2\} - \{p1, p2, p3, q1, q2, r2\} \\ = \{\neg p4, p6, r1\}. \end{aligned}$$

Next, the result is added to $\text{postcond}(T)$:

$$\begin{aligned} & [\text{postcond}(T) \cup (P - \text{precond}(T))] - (d) - (e) - (f) \\ & = (\{\neg p1, p2, p4, q2, r2\} \cup \{\neg p4, p6, r1\}) - (d) - (e) - (f) \\ & = (\{\neg p1, p2, p4, q2, r2, \neg p4, p6, r1\}) - (d) - (e) - (f) \\ & = (\{\neg p1, p2, p4, q2, r2, \neg p4, p6, r1\}) - \{ \} - (e) - (f) \\ & = (\{\neg p1, p2, p4, q2, r2, \neg p4, p6, r1\}) - (e) - (f) \\ & = (\{\neg p1, p2, p4, q2, r2, \neg p4, p6, r1\}) - \{\neg p4\} - (f) \\ & \quad \text{(because } \neg p4 \text{ is inconsistent with } p4) \\ & = \{\neg p1, p2, p4, q2, r2, p6, r1\} - (f) \\ & = \{\neg p1, p2, p4, q2, r2, p6, r1\} - \{r1\} \\ & \quad \text{(because } r1 \text{ was proven only by } \neg p4) \end{aligned}$$

$= \{\neg p1, p2, p4, q2, r2, p6\}$

(r2 remains because it is still proven by p2).

For the purpose of algorithm design, the approach 2 is easier because fewer verifications are required. The following algorithm computes the postcondition of a hierarchical plan, according to a slight variation of the approach 2, where literals from $(P - \text{postcond}(T))$ are added and tested for inconsistency one by one):

```

function postcond_from_task(P,T,Ok)
    {Input: a task T, with precondition(T) and postcond(T)
     Output: the result of applying T under P (if Ok = true)
     Variable: Q = literals of (P - precondition(T))}
begin
    if P | -/- precondition(T,Ok1) {Ok1 is not tested because T is
                                   executable under some precondition}
    then
        Ok := false {T is not executable under P}
    else
        begin
            Ok := true
            Post := postcond(T,Ok2) {Again, Ok2 is not tested}
            Q := P
            while Q ≠ { } do
                choose L ∈ Q {non-deterministic choice}
                Q := Q - {L} {remove L from Q}
                if L ∈ precondition(T) then {L is not handled by T}
                    if Post ∪ {L} | -/- false(Post ∪ {L} is consistent)
                    then {if Post does not derive L, then add L to it,
                        otherwise, adding L would be redundant}
                        if Post | -/- L then Post := Post ∪ {L}
                    else {L generates an inconsistency with Post}
                        remove_contributions_of(L,Q)
                postcond_from_task := Post
            end
        end
    end

```

The procedure `remove_contributions_of(L, Q)` removes all literals of `Q` whose all proofs depend, directly or indirectly, on the literal `L`. More precisely, for each literal of `Q` it removes every proof that uses the literal `L`, and, for each literal `L'` whose last proof is removed, the procedure recursively removes all *contributions* of `L'`:

```
procedure remove_contributions_of(L, Q)
  {Input:  a literal L and a set Q
   Output: a new value for Q, where literals inconsistent
           with L are removed}
begin
for each literal L' ∈ Q do
  Proofs := all_proofs_of(L') do
    for each proof Pf in Proofs do
      if Pf requires L then remove Pf from Proofs
if there exists a literal L' ∈ Q that has no proof tree
  then Q := Q - {L'}
      remove_contributions_of(L', Cond, Proofs)
end
```

Now that the problem of finding the postcondition of a hierarchical plan `H` is discussed, let us address the problem of computing the precondition.

Because the computation of `precond(H)` does not depend on `postcond(H)`, in contrast with the computation of `postcond(H)`, it would be surprising if both computations were symmetrical; as we will see, they are not. The basic idea consists of applying repeatedly over a precondition `P` (initially empty) the sub-hierarchical plans `H(1..1)`, `H(1..2)`, `H(1..3)`, ..., `H`, and by making an update of the initial precondition for each application where a literal required is missing. The idea can be represented by the following pseudo-code:

```

P := { }
for i := 1 to length(H) do
  if postcond_from(P,H(1..i-1)) | -/- precondition(H(i))
  then "update P accordingly".

```

Before presenting the algorithm, we will illustrate the idea with the following example.

Let $H = (t_1, t_2)$ be a hierarchical plan having the following preconditions and postconditions:

```

precond( $t_1$ ) = {p}      postcond( $t_1$ ) = {p,q}
precond( $t_2$ ) = {q,¬r}   postcond( $t_2$ ) = {q,¬r,t}.

```

The trace of the above pseudo-code is as follows:

```

P := { }
i := 1
P      postcond_from(P,H(1..i-1))      precondition(H(i))
{ }    postcond_from(P,H(1..0)) = { }    precondition( $t_1$ ) = {p}

Because postcond_from(P,H(1..0)) | -/- precondition( $t_1$ )
      (i.e., { } | -/- {p})
P is updated to {p} (i.e., to precondition( $t_1$ ))
(so, p must be provided by the initial condition).
i := 2
P      postcond_from(P,H(1..i-1))      precondition(H(i))
{p}    postcond_from(P,H(1..1)) = {p,q}  precondition( $t_2$ ) = {q,¬r}

Because postcond_from(P,H(1..1)) | -/- precondition( $t_2$ )
      (i.e., {p,q} | -/- {q,¬r})
P is updated to  $P \cup \{\neg r\} = \{p, \neg r\}$ , i.e.,  $P := \{p, \neg r\}$ 

(the missing condition  $\neg r$  must be provided by the initial
condition).

```

The precondition is thus $P = \{p, \neg r\}$, and the postcondition (obtained by executing (t_1, t_2) under P) is $\{p, q, \neg r, t\}$.

In summary:

because $\text{precond}(t_1) = \{p\}$, the initial condition P must contain $\{p\}$ (in order for t_1 to be executable). If $P = \{p\}$, then $\text{postcond_from}(P, t_1)$ returns $\{p, q\}$, and $\text{precond}(t_2)$ is only partially satisfied. Because the missing precondition $\neg r$ is not provided as a postcondition of the previous task t_1 , it must be provided by the initial condition, probably under the closed-world assumption. So, P must be set to $\{p, \neg r\}$, and H can then be executed, resulting in the postcondition $\text{postcond_from}(P, H) = \{p, q, \neg r, t\}$.

Example 3-1. Computing the precondition of a task.

Now, let us see an example of a hierarchical plan that is not executable, no matter the initial condition.

A variant of the previous example is taken: $H' = (t_1, t_3)$, where

$\text{precond}(t_1) = \{p\}$ $\text{postcond}(t_1) = \{p, q\}$
 $\text{precond}(t_3) = \{\neg q\}$ $\text{postcond}(t_3) = \{\neg q, t\}$.

The trace is as follows:

$P := \{ \}$

$i := 1$

$\text{P} \quad \text{postcond_from}(P, H'(1..i-1)) \quad \text{precond}(H'(i))$
 $\{ \} \quad \text{postcond_from}(P, H'(1..0)) = \{ \} \quad \text{precond}(t_1) = \{p\}$

Because $\text{postcond_from}(P, H'(1..0)) \not\models \text{precond}(t_1)$

(i.e., $\{ \} \not\models \{p\}$)

P is updated to $\{p\}$, (i.e., to $\text{precond}(t_1)$)

(p must be provided by the initial condition).

i := 2

$P \xrightarrow{\text{postcond_from}(P, H'(1..i-1))} \text{precond}(H'(i))$

{p} $\text{postcond_from}(P, H'(1..1)) = \{p, q\}$ $\text{precond}(t_3) = \{\neg q\}$

Because $\text{postcond_from}(P, H'(1..1)) \not\models \text{precond}(t_3)$

(i.e., $\{p, q\} \not\models \{\neg q\}$)

P is updated to $P \cup \{\neg q\} = \{p, \neg q\}$

($\neg q$ must be provided by the initial condition).

However, the postcondition resulting from the application of t_1 under P is $\text{postcond_from}(\{p, \neg q\}, t_1) = \{p, q\}$, which does not satisfy the precondition of t_3 .

So, the hierarchical plan H' is not executable.

In summary:

again, the initial condition P must contain {p}, and so $\text{postcond_from}(P, t_1)$ returns {p, q}. The precondition of t_3 is not satisfied, and the missing condition $\neg q$ is added to P. However, if H' is executed on this new $P = \{p, \neg q\}$, $\text{postcond_from}(P, t_1)$ will be again {p, q}, because t_1 overwrites the literal $\neg q$, leading us in the same problematic previous situation. Here, the problem is due to the impossibility of executing H' , no matter the initial condition.

Example 3-2. Computing the precondition of a non-executable task.

The code for computing $\text{precond}(H, Ok)$ is given below. It uses the function $\text{part_of_prec_causing_failure}(P, H, i)$ that identifies the components (literals) of the precondition of $H(i)$ that are missing (not provided by the state obtained by executing $H(1..i-1)$ under P).

```
function precondition(H, Ok)
  (Input: a hierarchical plan H
   Output: the precondition of the hierarchical plan H
   To avoid infinite loops, the variable "Failure" detects if
    a problem occurs twice when trying to compute the
    precondition of H(1..i))
begin
  Ok := true
case H.type of
    command: precondition := precondition(H) (directly provided)
    goal:     precondition := H.P0 (directly provided)
    otherwise: {H is a sequence of tasks}
      P := { }
      Failure := 0
      i := 1
      while (i <= length(H)) and Ok do
        Q := postcond_from(P, H(1..i), Ok)
        if Ok
          then i := i + 1
          else if Failure = i
            then Ok := false (an infinite loop is detected)
            else
              Failure := i (store the failure point)
              FailedPrec :=
                part_of_prec_causing_failure(P,H,i)
              Ok := ( P ∪ FailedPrec | -/- false )
              if Ok then P := P ∪ FailedPrec
            end
          endwhile
        precondition := P
      endcase
end
```

Let us now describe how `part_of_prec_causing_failure(P,H,i)` can be identified.

If there is a failure to satisfy $\text{precond}(H(i))$, then some element u of $\text{precond}(H(i))$ has no proof trees. However, u may have a partial proof tree, that is to say, a proof tree where some nodes fail to be proven. Any one of these nodes, say v (called a failing node), is a part of $\text{precond}(H(i))$ that causes the failure, although $v \notin \text{precond}(H(i))$.

Because several proofs may exist for u , and because making to succeed one failing node v (by adding it to P) may make another failing node to succeed (e.g., ancestors of v in the proof), the choice for which v to select is not deterministic.

One way to choose among the possible failing nodes is to consider the possible proofs and the failing nodes within them in some fixed order, to select failing nodes according to this order, and to backtrack when the choice later on appears to have been wrong (e.g., when a contradiction is detected).

The current implementation of `part_of_prec_causing_failure(P,H,i)` does backtrack: it selects all the failing nodes encountered at the lowest levels possible of the proof trees, and if a dead end appears later on, the algorithm climbs the proof trees to return ancestors nodes as failing nodes.

To complete this section, we will discuss the complexity of computing $\text{precond}(H)$ and $\text{postcond}(H)$ in the general case, and present simplifications that can be done according to the representation used to describe hierarchical plans.

The difficulty (complexity) of computing $\text{precond}(H)$ and $\text{postcond}(H)$ depends mainly on the complexity of the following sub-problems, where P and Q are sets (properties), and H is a hierarchical plan:

- determine if $P \models Q$
- determine if $P \cup Q \models \text{false}$, knowing that $P \not\models \text{false}$
- compute `part_of_prec_causing_failure(P, H, i)`.

The complexity of answering these questions (and even the necessity for answering them) depends on various parameters, including:

- hierarchical plan executability: the hierarchical plan is known to be executable, or it is unknown
- hierarchical plan instantiation: the hierarchical plan is fully instantiated, or partially instantiated
- use of negative literals in precondition/postcondition: all literals are positive, or some may be negative
- precondition/postcondition instantiation: the conditions contain only fully instantiated literals, they may contain partially instantiated literals, or even some formula in first order predicate logic
- axiom type: the theory contains no axioms, axioms are Horn clauses, axioms may have a negative literal as their consequent, or axioms can be any clauses (note that any first order predicate formula can be transformed into clauses [Nilsson 1980]).

We will discuss the complexity of answering the above questions for the two opposite situations: the simple case and the general case, respectively characterized as:

simple case:

- hierarchical plans are fully instantiated and known to be executable
- the precondition/postcondition is fully instantiated for each task of the hierarchical plan
- the precondition/postcondition of each task T contains only positive literals; the precondition being defined by the list $\text{precond}(T)$, the postcondition being specified by the pair $\text{ADD}(T) / \text{DEL}(T)$
- the theory contains no axioms;

general case:

- the hierarchical plan is not known to be executable, although generally this will be the case (in the case of an interaction analysis, the hierarchical plan corresponds to an actual executed interaction; in the other case, it results of combinations of executable hierarchical plans)
- the hierarchical plan is not fully instantiated (in the first case, the de-instantiation is done on the hierarchical plan prior to the computation of its conditions; in the second case, the hierarchical plan results of combinations of non-instantiated hierarchical plans)

- conditions may contain negative literals (predicates), but do not contain explicit disjunctions (recall that disjunctions are expressed via axioms)
- axioms are Horn clauses, and constraints are similar to axioms except that their left part may consist of a negative literal.

For the simpler case, above questions are answered as follows:

- determine if $P \models Q$
determine if $Q \subseteq P$
- determine if $P \cup Q \models \text{false}$, knowing that $P \not\models \text{false}$
always *false* (because it is impossible to derive *false*)

Because there are no axioms and no constraints in this simple case, a contradiction can be derived only from the presence of literals p and $\neg p$ in the same set (condition); because negative literals are not allowed, no contradictions can occur.

- compute `part_of_prec_causing_failure(P, H, i)`
compute $\text{precond}(H(i)) - P$.

In fact, for this simple case, it is possible to determine the precondition P and the postcondition Q of a hierarchical plan H in a very efficient manner (in linear time) if sets are implemented by bit vectors, as shown by the following algorithm that computes both of them in one pass over the hierarchical plan H :

```

P := { }      (precondition of H(1..i))
Q := { }      (postcondition of H(1..i))
for i := 1 to length(H) do
  ( add to P preconditions of H(i) that are
    not provided by Q, i.e., by postcond(H(1..i-1)))
  P := P  $\cup$  [precond(H(i)) - Q]
  (add to Q preconditions of H(i) (those already in Q
    and the new ones added to P),
    and delete then add conditions of H(i))
  Q := [(Q  $\cup$  precond(H(i))) - DEL(H(i))]  $\cup$  ADD(H(i)).

```

For the general case, the same questions are answered as follows:

- determine if $P \models Q$
 - determine if Q can be deduced from P using Horn clauses
 - This might require the computational power of a Turing machine.
- determine if $P \cup Q \models \text{false}$, knowing that $P \not\models \text{false}$
 - determine if $\neg Q$ can be deduced from P using Horn clauses
 - and constraints whose left part is possibly a negative literal
 - This might require the computational power of a Turing machine.
- compute `part_of_prec_causing_failure(P,H,i)`
 - This might require the computational power of a Turing machine.
 - (see the discussion following the algorithm for *precond(H,Ok)*)

In summary, the computation of *precond(H)* and *postcond(H)* can be done in linear time in the case of simplest representations, and the power of a full theorem prover for the more general representations required by the type of applications the proposed learning system is intended to deal with.

3.2.4 Constructing the resulting goal rule

This section addresses the problem of building a goal rule GR, given a goal $G = \langle P_0, Q_0 \rangle$ and a hierarchical plan H achieving it, along with its precondition and its postcondition.

Although this last computation may seem easy, it hides some particularities. At first glance, the goal rule being built should be

$$G \Leftarrow \langle \text{precond}(H), H, \text{postcond}(H) \rangle.$$

To see why it is not always the case, recall (Section 2.2) the difference between hierarchical plan triples and goal rules. If $\langle \text{precond}(H), H, \text{postcond}(H) \rangle$ is a hierarchical plan triple and $G \Leftarrow \langle P, H, Q \rangle$ is a goal rule, then $P \Rightarrow \text{precond}(H)$ (because H is executable under P) and $P \Rightarrow P_0$ (because the goal rule achieves G). However, P is not necessarily $\text{precond}(H)$, and nothing guarantees that $\text{precond}(H) \Rightarrow P_0$ (e.g., consider $\text{precond}(H) = \{p\}$ and $P = P_0 = \{p, q\}$).

Thus, it may be required to add some literals S to $\text{precond}(H)$ such that $\text{precond}(H) \cup S \Rightarrow P_0$. The same is true for the postcondition. Consequently, in order to create a goal rule $G \Leftarrow \langle P, H, Q \rangle$ such that $P \Rightarrow P_0$ and $Q \Rightarrow Q_0$, using the goal triple $\langle \text{precond}(H), H, \text{postcond}(H) \rangle$, the following can be used:

- add to $\text{precond}(H)$ as few literals as possible from P_0 to get a precondition P such that $P \Rightarrow P_0$
- compute $Q := \text{postcond_from}(P, H)$.

Another consideration to take into account when building the goal rule is its readability. Although the goal rule $G \Leftarrow \langle P, H, Q \rangle$ built as above is correct, it may contain superfluous knowledge. For instance, consider a theory having the two following axioms:

$$r \Leftarrow q \text{ and } q \Leftarrow p.$$

In this theory, the following four conditions are logically equivalent (because $p \Rightarrow q \Rightarrow r$):

$$p, p \wedge q, p \wedge r, p \wedge q \wedge r.$$

In this case, which one of these conditions should be presented to a human, for instance, when a goal rule is displayed?

Although in theory a condition P can be replaced with the set of conditions that logically follow from P , for the purpose of readability, a condition should be as short as possible (for instance, in terms of the number of conjunctions). A notion of *minimal conditions* is therefore desirable. Intuitively, a minimal condition should be such that the removal of any literal changes the truth value of the condition. In some cases, as in [Levi, Perschbacher et al. 1989], conditions that are normally present in the situation or features that have a value within normal range are removed from the resulting condition. The difficulty of this approach is to distinguish between ordinary and unusual conditions and values.

For instance, let us consider the following example taken from the domain of ER model design:

$$P = \text{entity}(A) \wedge \text{entity}(B) \wedge \text{related}(A, B)$$

$$\text{Axioms: } \text{related}(A, B) \Leftarrow \text{relation}(A, B, R).$$

$$\text{related}(A, B) \Leftarrow \text{card}(\text{min}, A, B, C) \wedge C \neq 0.$$

According to these axioms, the condition P is minimal. However, it is known that in this domain, a relationship cannot exist if the entities do not exist. Thus, we have:

$$\text{entity}(A) \leq \text{related}(A, _)$$

$$\text{entity}(A) \leq \text{related}(_, A).$$

These are not axioms, because making them so would transform $\text{entity}(A)$ into an inferential literal, thus prohibiting the use of a so basic literal inside states. Rather, they are constraints, as are their contrapositives

$$\neg \text{related}(A, _) \leq \neg \text{entity}(A)$$

$$\neg \text{related}(_, A) \leq \neg \text{entity}(A).$$

So, using axioms as well as constraints, the redundancy of literals inside preconditions is pointed out. For the previous example, the condition P can be rewritten as

$$P = \text{related}(A, B).$$

A given condition may be equivalent to several different minimal conditions. For instance, if the axiomatic theory allows circular definitions, the uniqueness of minimal conditions is not guaranteed. For instance, for the axioms

$$p \leq q \text{ and } q \leq p,$$

the condition $\{p, q\}$ is equivalent to two minimal conditions, $\{p\}$ and $\{q\}$.

Moreover, it might be useful for the human if the condition contains a particular interesting or meaningful literal, even if this literal is logically redundant. This is specially true for high level literals: they are meaningful and inferential.

To achieve a compromise between (1) readability by being short²⁹, (2) readability by being meaningful, and (3) conservation of the logical equivalence, the following approach can be used:

before displaying a condition P , all logically redundant literals are removed from P , except literals that are *meaningful*.

As mentioned in Section 3.2.1, the meaningfulness of literals is usually provided by experts. It might be specified by the technique generally used to define the operability criterion in explanation-based learning: a priori, and in a static manner, specify for each literal if it is or not meaningful.

This is not always sufficient. For instance, in entity-relationship model design, if *1_to_many* and *many_to_many* are both declared as meaningful to ensure that *many_to_many* will be displayed when such a relationship occurs, and that *1_to_many* will be displayed for the description of a 1-to-many relationship that is not a many-to-many relationship, then both will be displayed for the description of a many-to-many relationship (because in this case, both are true and meaningful):

$$Q = \text{many_to_many}(A,B) \wedge \text{1_to_many}(A,B) \wedge \text{1_to_many}(B,A).$$

However, *1_to_many* should not be displayed when *many_to_many* is. This illustrates that the display of conditions, and thus the definition of meaningfulness, should be relative to the proof tree(s) of the literals inside the condition, and so, should be dynamic³⁰.

²⁹ This is called *simplicity* (the number of literals in the condition is less than a specified value) in [Wilkins 1990], p.504.

³⁰ This could also be avoided by using a *subsumption table* specifying the *most-specific-than* relationship among literals.

This chapter introduced general methods used to learn from agent interactions; the following chapter proposes a protocol to evaluate these methods and illustrates their applications for two domains, the entity-relationship model design and the office tasks domain.

3.3 Summary of the approach

This section summarizes the approach proposed in this chapter to the problem of non-obtrusive knowledge acquisition in the context of an incomplete domain theory.

The approach is described by presenting the sequenced list of its main elements, as follows:

- the assumption that the agent achieves goals by applying a sequence of actions that are derived from the decomposition of goals into subgoals
- the assumption that this sequence of actions (i.e., the interaction) is observable in a non-obtrusive way and executable (simulable) by the learning system for producing the sequences of corresponding states (using the domain theory)
- the use of EBL (Explanation-Based Learning) techniques to explain (through the domain theory) the interaction
- the assumption that the explanation of the interaction does not cover the entire interaction
- the use of heuristics to identify learning opportunities within the interaction (goals, goal rules)

- the computation of precondition and postcondition the proposed goal rules (efficient algorithms for the non-inferential cases, a conservative approach for the more complex cases)
- proposals for enhancements of the domain theory that result (in a first step) in additional explanations of the agent's interaction used for training, and (later on) in explanations (or even guidance) in interactions of future agents.

CHAPTER 4

Experimentation with LEADER

In previous chapters, we have developed general methods to acquire knowledge, in an unsupervised manner, to enhance a knowledge-based system.

This chapter illustrates the applicability of ideas described in previous chapters. Two hypotheses were verified experimentally:

- the learning system can be used to increase significantly the *coverage of a domain theory* (the number of situations taken into account by the domain theory), that is to say, a significant gain in the number of actions explained can be obtained by the enhanced domain theory
- the learning system can propose useful enhancements for domain theories designed for practical domains.

We have implemented a version of LEADER in Prolog [Quintus Computer Systems 1987] and have used it to enhance the domain theory of two different domains: the execution of office tasks, and the design of entity-relationship models. The first domain was used to show that LEADER can be used to enhance significantly a domain theory. The second domain was more pertinent because existent expert systems rely on such a domain theory; it was used to illustrate that LEADER can produce useful enhancements.

Section 4.1 describes the experimental protocol used for generating new knowledge with LEADER. Section 4.2 presents the application of this protocol to the Office Tasks domain. Section 4.3 presents the application of the learning methods to the Modeller, an expert system for database design.

Note that the protocol was used only on the first domain to study the overall behaviour of LEADER. The protocol (and the large scale tests that go with it) was not applied on the modelling domain because the purpose of this domain was to illustrate the applicability (capability of producing some useful knowledge) of LEADER on a real domain.

4.1 Experimental protocol

In order to tune LEADER in a way that it can produce appropriate knowledge and to evaluate its learning performances, the plan shown below was used. The main components of this plan are elaborated in subsequent sections.

1. Identify an appropriate domain and its domain knowledge (basic goals, literals, commands and hierarchical plans).
2. Define a training interaction suitable for learning the knowledge identified in Step 1.
3. Run LEADER over the interaction defined in Step 2.
4. Run LEADER over new (testing) sets of interactions, preferably independent ones.
5. Using results of Steps 3 and 4, verify if LEADER succeeds to produce the pertinent knowledge.

4.1.1 Choice of a testing domain

The first step to evaluate LEADER consists in selecting an appropriate domain. The domain should be chosen according to given characteristics. Among these suitable characteristics, the following can be identified:

- Experts for the domain must be available. This is mandatory for two reasons: experts are required to design the initial domain theory and to evaluate the knowledge produced the learning algorithm: poor initial domain theory may lead to poor learning (the latter could incorrectly be attributed to the learning method), and wrong evaluation of the results may cause the rejection of an appropriate learning method. Also, domain experts can be helpful for designing the training and testing interactions (Steps 2 and 4).
- The domain must be flexible and rich. In order for the experts to have good control over the input and output of the learning system, the domain should allow a wide range of values for parameters that define it. Otherwise, the domain would be too constraining to exploit the learning capabilities of the system.

In particular, the domain should have hierarchical goal rules to exploit the learning capability of LEADER.

- The domain should be meaningful. Although it is not a requirement, a domain whose actions and goals are meaningful is more appropriate for explaining the learning method, the protocol and the results.

Because experts are often difficult (and costly) to find, a tempting approach could be to choose an artificial domain. This would allow the designer of the domain to have a good control over it (design of the domain knowledge, construction of the training and testing sets...), and, especially, to be the expert of that domain. To explore this avenue, we considered using the artificial domains described in [Kambhampati and Jengchin 1993]. These domains (defined in Figure 4-1) can be described as follows:

ART-IND (artificial domain with independent goals)

This domain contains only independent goals (the DEL list of every action is empty). Consequently, in order to achieve the goal $G_i \wedge G_j \wedge G_k$, with $i < j < k$, the actions A_i , A_j and A_k must be executed, in any order:

Goal = $G_i \wedge G_j \wedge G_k$, with $i < j < k$

Precond = $I_i \wedge I_j \wedge I_k$

DO (in any order): (A_i , A_j , A_k)

Postcond = $I_i, I_j, I_k, G_i, G_j, G_k$

ART-MD (artificial domain with interacting *sequenceable*³¹ goals)

Although the domain contains dependent goals, some plans can be sequenced with actions to achieve new goals. For instance, to achieve the goal $G_i \wedge G_j \wedge G_k$, with $i < j < k$, given the plan (A_i , A_j) for achieving the partial goal $G_i \wedge G_j$, the action A_k must be appended at the end of the plan:

Goal = $G_i \wedge G_j \wedge G_k$, with $i < j < k$

Precond = $I_i \wedge I_j \wedge I_k$

DO (in that precise order): (A_i , A_j , A_k)

Postcond = $\neg I_1 \wedge \neg I_2 \wedge \dots \wedge \neg I_{k-1} \wedge I_k \wedge G_i \wedge G_j \wedge G_k$

ART-MD-NS (artificial domain with interacting non-sequenceable goals)

In that domain, goals are not sequenceable. For instance, to achieve the goal $G_i \wedge G_j \wedge G_k$, with $i < j < k$, given the plan (A^1_i , A^1_j , A^2_i , A^2_j) for

³¹ A goal g is sequenceable with a plan H achieving a set G of goals if and only if there exists a subplan P for achieving g such that (H,P) or (P,H) is a correct plan for achieving the set of goals $G \cup \{g\}$.

achieving the partial goal $G_i \wedge G_j$, the action A^1_k must be inserted inside the plan:

Goal = $G_i \wedge G_j \wedge G_k$, with $i < j < k$

Precond = $I_i \wedge I_j \wedge I_k$

DO (in that precise order): $(A^1_i, A^1_j, A^1_k, A^2_i, A^2_j, A^2_k)$

Postcond = $\neg I_1 \wedge \neg I_2 \dots \wedge \neg P_1 \wedge \neg P_2 \wedge \dots \wedge \neg P_{k-1} \wedge P_k \wedge G_i \wedge G_j \wedge G_k$.

ART-IND

Action: A_i

Precond = I_i

ADD = $\{ G_i \}$, DEL = $\{ \}$

ART-MD

Action: A_i

Precond = I_i

ADD = $\{ G_i \}$, DEL = $\{ I_j \mid j < i \}$

ART-MD-NS

Action: A^1_i

Precond = I_i

ADD = $\{ P_i \}$, DEL = $\{ I_j \mid j < i \}$

Action: A^2_i

Precond = P_i

ADD = $\{ G_i \}$, DEL = $\{ I_j \mid \forall j \} \cup \{ P_j \mid j < i \}$

Figure 4-1. Candidate artificial domains.

We did not retain any of these domains for evaluating LEADER for the following reasons:

- The domains are not meaningful. Although they are simple and relatively easy to understand, their interpretation lacks a desirable semantics.

- The domains do not involve hierarchical goal rules: all explanations for goal achievements are flat. This is not rich enough to exploit the learning capabilities of LEADER that supports hierarchical goal rules. An alternative would have been to add hierarchical goals and rules to these domains. However, the artificial nature of this domain would have given rise to the problem of building this hierarchical knowledge.
- In the first domain (ART-IND), there are many ways to achieve the goal $G_i \wedge G_j \wedge G_k$, however, all these ways use the same actions. This is because each different action achieves a different goal, that is to say, there exist no two different actions that achieve the same goal. So, only the order of the actions can be learned. In general, learning the order might be interesting, however, because in this particular domain the order is irrelevant to achieve a given goal, the learning becomes uninteresting. In addition, the fact that there is only one action for achieving each elementary goal G_i means that the learning capability of LEADER is not fully exploited.
- In the third domain (ART-MD-NS), there is only one way to achieve the goal $G_i \wedge G_j \wedge G_k$, (any other ordering of the required actions fails to achieve the goal). So, it is not possible to learn a new way to achieve a goal, and again, these constraints are not suitable for LEADER.

4.1.2 Choice of the training interactions

The design of a training interaction is divided into two tasks: first, the determination of an initial state in which the agent interacts, and then determination of the sequence of actions that constitute the interaction.

The choice of the initial state is critical because the state must allow a good number of actions at each step of the interaction: not so few that the interaction is too strongly biased, and so too many that there is a combinatorial explosion or a lack of regularity and similarity required for learning.

For example, in a blocks world domain consisting of only one block and two actions (pick_up(B) and put_down(B)), only two different actions are possible, thus learning is limited. As another example, in a domain where objects can be created by actions (as in modelling domains), the number of different actions can be infinite, and structural regularity required for learning might not manifest itself through the interaction.

To analyze the effects of the initial state on learning, the experiments should be done using different initial states of the same domain.

For the same reasons as for the choice of the initial state, the generation of interactions is also a critical process: LEADER learns from the part of the interaction that is unexplained, and that part must have a strong link with the current domain theory (so the interaction must have a good number of actions).

The generation of interactions can be done in several ways: by observing experts, by observing novices, by biased random generation, and by random generation. The first method is the best one for producing new pertinent knowledge, the last one is the worst (one should expect to learn more from an expert than from a random generator). Because the overall experiments required a large number of independent interactions (about 1000), we took the third approach. The random nature guarantees the independency between the interactions and should provide a lower bound on learning performances. The particular bias used will be described in Section 4.2.2.

The length of the interaction is also an important factor because the quality and quantity of knowledge produced may be directly related to it. To study this possible bias, we repeated the experiments with interactions of different lengths.

The interactions generated must be executable in the given initial state. To guarantee the executability, the following method was used to produce each interaction of a given length L:

0. State := the initial state
1. Interaction := the empty interaction
2. For N := 1 to L do
 - 2.1 Actions := all actions executable in the state State
 - 2.2 Action := randomly selected action within Actions
Interaction := concatenation of Interaction and Action
 - 2.3 State := application of Action to the state State

A bias has been added for the generation of interactions. Accordingly, Step 2.1 above has been replaced with:

- 2.1 Actions₁ := all actions executable in the state State
Actions₂ := Actions₁ – *absurd* actions
Actions₃ := Actions₂ – actions previously done
IF Actions₃ is not empty
THEN Actions := Actions₃
ELSE IF Actions₂ is not empty
THEN Actions := Actions₂
ELSE Actions := Actions₁.

Absurd actions are the ones that nobody executes in real life (such as *P says something to P*, *P phones P*). According to the new Step 2.1, the effect of the bias would produce realistic (although random) interactions.

4.1.3 Choice of the testing interactions

As for the training interactions, the design of testing interactions is divided into the two subtasks: the determination of the initial state, and the determination of the sequence of actions.

The initial state can be chosen as the one in the training interaction, or can be entirely different. Following the classical approach consisting in dividing the available instances into a training set and a testing set, we took a given initial state for generating $N+1$ interactions of the same length, and to divide these instances into 1 interaction for training and N interactions for testing. To cancel any marginal point of the learning curve, experiments were executed several times with the same length, and all of it was repeated for various lengths.

4.1.4 Choice of the evaluation criteria

The last step of the protocol is the evaluation of the knowledge produced by LEADER. Several measures of the performance were computed, including the coverage of the domain theory (the number of actions that are explained) and the number of rules produced.

The most important measures evaluated the explanatory power of the learned part of the theory, as follows: LEADER was applied with the initial domain theory on the training set to produce new rules, and these new rules were added to the initial domain theory to produce the enhanced theory. Then, LEADER was applied on independent testing sets,

using separately for each training set the initial domain theory and the enhanced one. This operation gave the number of actions explained by the initial domain theory (before learning on the training interaction), and the number of actions explained by the enhanced domain theory (after learning). These numbers were divided by the length of the interaction to provide the relative coverage (proportion of the interaction that's explained).

Among the different ways that may be used to specify that a particular action is explained by a domain theory, two relevant ones can be mentioned (Section 2.5.2):

- The action is direct (the action is directly specified in a goal rule used to achieve a goal)
- The action is indirect (it is required to achieve the precondition of a goal rule used in a goal tree).

Using these, the two following statistics were retained to evaluate the coverage:

- The number of actions that are a leaf of a goal tree (direct actions)
- The number of actions that are required by a goal tree (direct or indirect actions).

In addition, the relative gain of coverage (the difference of the relative coverage after learning and before learning) was used as a measure of the learning performance.

In addition, the variability of situations appearing in the same interaction was also of interest to us. Indeed, there is a relationship between the richness of a domain (in terms of the variety of situations that can appear) and the complexity of the domain theory required to describe it.

We proposed two parameters to measure this variability, based on the number of different actions appearing in a given interaction: the proportion of the interaction that contains different actions (the number of different actions appearing in a given interaction divided by its length), and the proportion of the total space of actions that appears in the interaction (the number of different actions appearing in a given interaction divided by the total number of possible different actions).

Finally, the number of rules produced by LEADER was used as a measure of the learning performance, in relationship with the relative gain.

4.2 Application to the Office Tasks domain

The previous section presented the protocol used to evaluate the learning performance of LEADER. This section describes the domain on which the protocol was applied, the Office Tasks domain (Section 4.2.1), the experimentation performed on it (Section 4.2.2) and the results obtained (Section 4.2.3).

4.2.1 The Office Tasks domain

The Office Tasks domain describes the tasks, goals and actions arising in a typical office involving persons, information, letters, phones and fax machines. In this domain, a given number of persons have information that they want to exchange. This information takes the form of letters transmittable by fax, or of any verbal information transmittable by phone (such as phone and fax numbers). To transmit information, a person must establish the communication (using a fax machine or a phone), transmit the information, and break the connection.

Table 4-1 shows the actions of the domain (without the condition components), Figure 4-2 presents an action with its four condition components: the precondition, the DEL and ADD lists and the postcondition, Figure 4-3 presents a goal and a hierarchical goal rule achieving it (the predicate *a_goal* provides the conditions P_0 and Q_0 for the given goal). The domain theory defining the actions is given in Appendix B, and the domain theory for goals is provided in Appendix C.

Actions	Meanings
say(T,M,R)	T says the info M to R
ask(T,M,R)	R asks the info M to T
fax(T,M,R)	T fax the letter M to R
look_up_phone_book(T,R,No)	T looks in his phone book to get the phone no No of R
look_up_fax_book(T,R,No)	T looks in his fax book to get the fax no No of R
call_phone(T,No,R)	T dials the no No to call R by phone
call_fax(T,No,R)	T dials the no No to call R by fax
redial_phone(T,No,R)	T presses the redial button to call R by the phone no No
redial_fax(T,No,R)	T presses the redial button to call R by the fax no No
hang_off_phone(T,R)	T closes the phone connection with R
hang_off_fax(T,R)	T closes the fax connection with R

Table 4-1. Actions of the Office Tasks domain.

```

action(redial_phone(T,No,R),
  [phone_free, not in_contact(T,R,verbal), phone_mem(T,No,R)],
  [phone_free],
  [in_contact(T,R,verbal)],
  [not phone_free, in_contact(T,R,verbal), phone_mem(T,No,R)]).

```

Figure 4-2. The *redial_phone* action.

```

a_goal( transmit_info(T,M,R),
  [has(T,info,M)],
  [has(T,info,M),has(R,info,M)]).

gr( trans_info1, (transmit_info(T,M,R) :- [
  [has(T,info,M),tellable(M)],
  [has(T,info,M),tellable(M),has(R,info,M)],
  get_phone_no(T,R,No),
  transmit_mes_phone(T,M,R,No)]))).

gr( get_ph_no1, (get_phone_no(T,P,No) :- [
  [phone_book(T,P/No)],
  [phone_book(T,P/No), has(T,info,P/phone_no/No)],
  com(look_up_phone_book(T,P,No)) ] ) ).

gr( get_ph_con1, (get_ph_connection(T,No,R) :- [
  [phone_free, not in_contact(T,R,verbal)],
  [not phone_free, has(T,info,R/phone_no/No),
    in_contact(T,R,verbal)],
  com(call_phone(T,No,R)) ] ) ).

gr( trans_mes_ph, (transmit_mes_phone(T,M,R,No) :- [
  [phone_free, has(T,info,M), tellable(M)],
  [phone_free, has(T,info,M), tellable(M), has(R,info,M)],
  get_ph_connection(T,No,R),
  com(say(T,M,R)),
  com(hang_off_phone(T,R)) ]))).

```

Figure 4-3. A goal and related goal rules.

The following figure describes an initial state involving three persons (*a*, *b* and *c*) in a situation where the phone and fax lines are free, the memory of all phones and fax machines is empty, the person *a* possesses in his phone book all the phone numbers and nothing else, *b* possesses in his fax book all the fax numbers and nothing else, and *c* possesses two letters (*l1* and *l2*) and nothing else.

```
[  phone_free,  
    fax_free,  
    phone_mem(no_number,no_one),  
    fax_mem(no_number,no_one),  
    phone_book(a,a/ap),  
    phone_book(a,b/bp),  
    phone_book(a,c/cp),  
    fax_book(b,a/af),  
    fax_book(b,b/bf),  
    fax_book(b,c/cf),  
    has(c,info,letter/l1),  
    has(c,info,letter/l2) ]
```

Figure 4-4. An initial state for the Office Tasks domain.

In this domain, if the initial state contains P persons and L letters, the number of different possible actions allowed by the various instantiations of the actions shown in Table 4-1 is:

$$\text{NbActions}(P,L) = 4 P^3 + (L + 4) P^3 - (L + 6) P.$$

We assume that the Office Tasks domain has the following properties:

- All the phones are on the same phone line; so either the phone line is free, or two persons are in verbal contact by phone. The same is true for fax machines that share their own fax line.
- Each phone (and fax machine) has a memory that stores the last number dialed on it.
- Each person has a phone book and a fax book. The content of these books is described by the initial state.
- In order to be verbally transmitted (by phone, by saying or by asking), a piece of information must be tellable.

- In order to be transmittable by fax machines, an information must be faxable. Only letters are faxable.

This domain was chosen to satisfy the requirements stated in Section 4.1.1, as:

- it is meaningful (it corresponds to real situations where humans have to perform tasks and learn)
- it involves hierarchical goals
- there are many different ways to achieve some goals
- it is relatively easy to create domain theory, states and interactions
- it is easy to judge the appropriateness of the knowledge produced by the learning methods
- no external experts are required for this domain.

4.2.2 Experimentation with the Office Tasks domain

Experiments were done using two initial states. The first one is described in Figure 4-4. In this state, the total number of different actions is $NbActions(3,2) = 138$.

The second initial state (Figure 4-5) has more potential because it involves 6 persons and 6 letters. Again in this state, the person *a* knows the phone numbers of the six persons and nothing else, *b* knows all the fax numbers and nothing else, and *c* possesses all the letters and nothing else. In this configuration, there are a total number of $NbActions(6,6) = 1152$ different actions.

```
[ phone_free),  
  fax_free,  
  phone_mem(no_number,no_one),  
  fax_mem(no_number,no_one),  
  phone_book(a,a/ap),  
  phone_book(a,b/bp),  
  phone_book(a,c/cp),  
  phone_book(a,d/dp),  
  phone_book(a,e/ep),  
  phone_book(a,f/fp),  
  fax_book(b,a/af),  
  fax_book(b,b/bf),  
  fax_book(b,c/cf),  
  fax_book(b,d/df),  
  fax_book(b,e/ef),  
  fax_book(b,f/ff),  
  has(c,info,letter/l1),  
  has(c,info,letter/l2),  
  has(c,info,letter/l3),  
  has(c,info,letter/l4),  
  has(c,info,letter/l5),  
  has(c,info,letter/l6) ]
```

Figure 4-5. The second initial state for the Office Tasks domain.

As explained before (Section 4.1.2), a bias was used to generate interactions. For both initial states, the following actions were removed as absurd actions:

```
ask(P,_,P),          say(P,_,P),  
call_phone(P,_,P),   call_fax(P,_,P),  
hang_off_phone(P,P), hang_off_fax(P,P).
```

For each initial interaction, experiments with interactions of length 50, 100, 150, 200, 250 and 300 were performed. For each different length and initial state, 1 training interaction and 10 testing interactions were generated (by the biased random method described above), the results produced by the training interaction were used on the testing sets, the statistics were collected, and the entire operation was repeated 20 times (each time with a new training interaction).

4.2.3 Results for the Office Tasks domain

This section describes in detail the results of experimentations with LEADER. These results can be summarized as follows:

- Learning Heuristic 1 (learning a new hierarchical plan for the first task of a goal rule, Section 3.2) is appropriate for this domain: all proposed rules were meaningful, and some of them were useful.
- Learning Heuristic 2 (learning a new hierarchical plan for a goal when the corresponding goal rule is partially satisfied, Section 3.2) is not appropriate for the Office Tasks domain: almost all proposed rules contained redundant action sequences such as *call_fax(a,_b)*, *hang_off_fax(a,b)*, *call_fax(a,_b)*. No results derived by this heuristic are presented here.
- The number of rules proposed increases almost linearly with the length of the interactions (as explained next, this results from the fact that no mechanisms for fusioning similar rules are used).
- The relative coverage decreases with the length of the interactions (this phenomenon is explained later).
- The relative gain of coverage decreases with the number of rules proposed (this phenomenon is also explained later). The gain goes from 5.6% to 3.5% when both direct or indirect actions are considered, and goes from 13.1% to 5.6% when only direct actions are considered.

Figure 4-6 shows examples of rules produced by LEADER for the Office Tasks domain.

"in order for a person T to get the fax number N of the person P,
P can tell it to T"

```
gr( new_gr, (get_fax_no(T,P,No) :- [
  [has(P,info,P/fax_no/No), in_contact(P,T,verbal)],
  [has(P,info,P/fax_no/No), in_contact(F,T,verbal)
    tellable(P/fax_no/No), has(T,info,P/fax_no/No)]
  com(say(P,P/fax_no/No,T))])).
```

"in order for a person T to get the fax number N of the person P,
T can ask it to P"

```
gr( new_gr, (get_fax_no(T,P,No) :- [
  [has(P,info,P/fax_no/No), in_contact(T,P,verbal)],
  [has(P,info,P/fax_no/No), in_contact(T,P,verbal)
    tellable(P/fax_no/No), has(T,info,P/fax_no/No)]
  com(ask(T,P/fax_no/No,P))])).
```

"in order for a person T to get a fax connection with a person R
whose no is No, T can press the redial button"

```
gr( new_gr, (get_fax_connection(T,No,R) :- [
  [fax_free, ¬ in_contact(T,R,fax), fax_mem(T,No,R)],
  [¬ fax_free, in_contact(T,R,fax), fax_mem(T,No,R)],
  com(redial_fax(T,No,R))])).
```

Figure 4-6. Goal rules produced by LEADER.

In some cases, no rules were produced by analyzing the training interaction. This is different than producing goal rules that provide no coverage gain in the testing interactions. In the latter case, the new knowledge provides none of the expected gain, and this failure must be considered in statistics about gain. In the former case, because nothing is proposed by LEADER, no tests will occur on training interactions. Thus, these situations must not be considered for computing relative gain, and for this reason we rejected them for the computation of these statistics.

However, they were included in Figure 4-7 to show the average number of rules produced for the different interaction lengths, for the two initial states, and in Figure 4-8 presenting the relative coverage gain as a function of the number of rules produced (for the second initial state).

Figure 4-7. Average number of rules produced by LEADER for the two initial states.

The above graph shows that the number of rules proposed by LEADER increases almost linearly with the length of the interactions. However, the graph in Figure 4.8 shows that the *relative* gain of coverage decreases with the number of rules proposed. This indicates that although the number of rules learned increases with the length of interactions, the variety of situations covered (the quality and diversity of rules) does not increase as fast: one rule is good, but two are not twice as good. This increase can be explained by the fact that no mechanisms are provided for generalizing and fusing similar rules among those produced. Thus, similar situations lead to similar but distinct (counted several times) rules. Note that the production of such similar goal rules contributes to aggravate the inefficiency

of the domain domain (a problem that is not addressed in this work). It is also possible that the relative coverage is not a good measure of learning performance: dividing by the interaction length may cause an artificial drop in the performance measurements.

Figure 4-8. Relative gain in coverage of interactions in function of the number of rules produced (initial state 2). For this statistic, the average of the relative gain in coverage was computed over the learning sessions having produced the same number of rules.

The following figures present the coverage measure after the removal of cases where no rules were produced by LEADER.

Figure 4-9 presents the relative coverage for the first initial state, before and after learning, according to the two measures described in Section 4.1.4: the proportion (with respect to the interaction length) of direct actions only, and the proportion of direct and indirect actions. As expected from the definitions of the two measures, the latter proportion is higher than the former. In fact, the large proportion of actions explained after learning on a biased random interaction is surprising: almost 68%!

Here again, the figure illustrates that the variety (in proportion) of situations covered by the domain theory decreases with the length of interactions. Also, a gain is obtained after learning (as shown by the difference of the two curves at the top, and the difference of the two curves at the bottom); results related to the gain will be analyzed in detail thereafter.

Figure 4-9. Relative coverage of interactions for the first initial state. The two curves at the bottom illustrate the coverage with direct actions, before and after learning (the curves also appear in Figure 4-11 for a comparison with the second initial state). The two curves at the top illustrate the coverage using both direct and indirect actions.

Figure 4-10 illustrates the variability of situations appearing in the interactions. The curves represent, for each initial state, the proportion of different actions inside interactions (different/length), and the proportion of possible actions present in the interactions (different/possible).

The figure shows that the proportion of possible actions increases (to converge toward 100%) with the interaction length, whereas the proportion of different actions decreases,

eventually reaching 0%. Although the number of different actions used in the testing interactions increases with the length of interactions, the combinatorics creates new behaviours not taken into account by the theory.

Figure 4-10. Variety (%) of actions used. The two curves at the top (decreasing towards 0%) represent the proportion of interactions containing different actions. The two curves at the bottom (increasing towards 100%) are the proportion of possible actions that were present in the interactions.

The next figure presents the relative coverage after and before learning. The graph shows the results for the two initial states and considers only direct actions. All the curves tell the same story: the relative coverage decreases rapidly as the interaction length increases. So, both initial states provide interactions of similar nature (in terms of variety of situations). However, the high coverage of the interactions generated from the second initial state indicates that this state provides more regularity in the interactions.

Figure 4-11. Relative coverage of interactions for the two initial states, considering only direct actions. The two lower curves represent the coverage before learning, the two higher ones illustrate the coverage after learning.

The relative gain of coverage, defined by the difference of relative coverage after and before learning, is presented in Figure 4-12. The graph shows the results for the two initial states. The observed gain varies from 4% to 13%. In contrast with the relative coverage, the relative gain for both initial states is higher when only direct actions are considered. This can be explained by the fact that the higher the coverage, the more difficult it is to increase it. Also, for this statistic (considering only direct actions), the second initial state provides a higher relative gain, indicating a better opportunity for learning.

Figure 4-12. Relative gain of coverage for the two initial states. The two higher curves represent the relative gain of coverage when only direct actions are considered. The two lower curves consider both direct and indirect actions.

4.3 Application to an expert system for modelling

This section presents the application of LEADER to enhance the domain theory of Modeller, an expert system for database design. The purpose of this second domain is to illustrate the potential of LEADER when applied to a real domain. Because of this particular purpose, initial states and interaction were intentionally designed to give LEADER the opportunity of producing pertinent knowledge.

Section 4.3.1 presents the components of the Modeller, with emphasis on the component used for illustrating the learning; Section 4.3.2 describes experiments and the results obtained.

4.3.1 Description of the Modeller

This section gives an overview of the Modeller. A more detailed description is provided in [Tauzovich 1990].

The Modeller is an assistant to a database designer. It consists of three levels, each one supported by an appropriate expert system implementing one of the three main steps in database design methodology [Navathe 1988]: conceptual design, logical design and physical design (Figure 4-13).

Through interactions with a graphical interface, the designer develops, at the first level, a description of the enterprise being modelled. In addition to the graphical display, the model is represented internally by an assertional language. The model, as well as its verification, is translated by the conceptual design expert system. The output of this first phase is a DBMS-independent (Data Base Management System) model, called the conceptual model.

For each action of the designer, the conceptual design expert system uses a fixed set of axioms (such as the semantics of commands, or the taxonomy of properties) to derive the new model, checking for redundancy, subsumption (a new fact specializing an old one), inconsistency, potential topological problems, the application of default rules, etc.

Figure 4-13. Architecture of the Modeller.

The input of the logical design expert (the second level of design) is the conceptual model. Its output is the logical description of the model, called the logical model. Finally, in the third step, the logical model is converted into a particular type of physical database, using the physical design expert corresponding to the selected database.

Of the three experts forming the Modeller, the conceptual design expert is the most interesting choice for an application with LEADER because the result of the interaction with the user depends largely on the expertise of the user (in contrast, most of the tasks of the two other expert components can be automated). So, the generic LEADER described in Chapter 3 was instantiated to learn knowledge for enhancing the Modeller's conceptual design expert.

Before describing the result of the experimental protocol for this particular domain, the parallel between general concepts introduced in previous chapters and concepts used in the

context of assisted design is presented (Table 4-2), as well as the nature of actions and commands intercepted by the Interaction Observer (Table 4-3). The semantics (precondition, DEL and ADD lists, postcondition) of actions used in the next section is shown in Figure 4-14. The semantics of Modeller actions implemented³² in LEADER is provided in Appendix D, the inferential literals are defined in Figure 4-16, and the goal structure is given in Appendix E.

General Concepts	Design Concepts
agent	expert designer (user)
world	space of possible designs
context	design representation, software tools
state	design
command	software command
action	use of a software command
property	particularity of a model
interaction	user session
axiom	semantics of software commands
goal	design modification
hierarchical plan	design methodology
plan	expansion of a design methodology

Table 4-2. General concepts vs concepts used in design.

³² We have implemented in LEADER a subset of the actions and the inferential literals available to designers using Modeller.

Commands	Meanings
addE(Ent)	create the entity Ent
delE(Ent)	delete the entity Ent
addR(Ent1,Ent2, Type)	create a relationship of type Type going from entity Ent1 to entity Ent2
delR(Ent1,Ent2)	delete the relationship going from entity Ent1 to entity Ent2
chgRT(Ent1,Ent2, Type)	change the type of the relationship going from entity Ent1 to entity Ent2 to the value Type

Table 4-3. List of commands available to describe the designer interaction.

```

action(addE(Ent),
  [¬entity(Ent)],
  [ ],
  [entity(Ent)],
  [entity(Ent)]).

action(addR(Ent1,Ent2,R),
  [entity(Ent1) ∧ entity(Ent2) ∧ ¬related(Ent1,Ent2)],
  [ ],
  [relation(Ent1,Ent2,R)],
  [entity(Ent1) ∧ entity(Ent2) ∧ relation(Ent1,Ent2,R)]).

action(delR(Ent1,Ent2),
  [entity(Ent1) ∧ entity(Ent2) ∧ relation(Ent1,Ent2,R)],
  [relation(Ent1,Ent2,R)],
  [ ],
  [entity(Ent1) ∧ entity(Ent2) ∧ ¬related(Ent1,Ent2)]).

```

Figure 4-14. Semantics of some Modeller's actions.

4.3.2 Experimentation with Modeller

This section illustrates the application of LEADER to enhance the knowledge used by the conceptual design expert of the Modeller, and presents the results obtained.

First, the initial domain theory was extracted from the Modeller's conceptual design expert. Then, appropriate and realistic initial states and interactions were designed to illustrate learning using Learning Heuristic 2 (*infer a new goal rule that realizes a known goal, given a partially realized goal rule for that goal*, Chapter 3). A complete trace of the session with LEADER is provided in Appendix F. Main components of this session are described below.

The following goal was selected to exhibit learning:

gen_entity(A,B,G) =
"create a more general entity by creating a new entity and
a dependency relationship."

The definition of the goal along with the appropriate goal rules are provided in Figure 4-15. Figure 4-16 presents the inferential literals involved (in particular, a dependency relationship is either a *role* or a *characteristic* relationship). Notice that in the current domain theory, it is assumed that the only way known to create a dependency relationship is to create a *role* link, although a characteristic link is also a dependency relationship.

An example of an instantiation of the goal is pictured in Figure 4-17: to generalize the entity *teacher* by creating the more general entity *dept_employee* and by transferring the old link *related* between *teacher* and *department* to the new entity.

```

gen_entity(A,B,G) <= < P0, Q0 >
    P0 = entity(A) ∧ entity(B) ∧ ¬entity(G) ∧ relation(A, B, R)
    Q0 = entity(A) ∧ entity(B) ∧ entity(G) ∧ ¬related(A, B)
        ∧ relation(G, B, R) ∧ depend_on(A, B)

/* Generalization of a relationship by creating a role relationship */
gen_entity(A,B,G) <=
    < entity(A) ∧ entity(B) ∧ ¬entity(G) ∧ relation(A, B, R),
      ( com(addE(G)),
        make_dependent_on(A,G),
        com(addR(G, B, R)),
        com(delR(A, B)) ),
      entity(A) ∧ entity(B) ∧ entity(G) ∧ ¬related(A, B)
        ∧ relation(G, B, R) ∧ depend_on(A, G) >

make_dependent_on(A,G) <=
    < entity(A) ∧ entity(G) ∧ ¬depend_on(A,G),
      entity(A) ∧ entity(G) ∧ depend_on(A,G) >

/* Creation of a dependency relationship by creating a role relationship */
make_dependent_on(A,G) <=
    < entity(A) ∧ entity(G) ∧ ¬related(A,G),
      ( com(addR(A, G, role)) ),
      entity(A) ∧ entity(G) ∧ relation(A,G,role) >

```

Figure 4-15. Part of the initial domain theory for Modeller.


```

related(Ent1, Ent2) <= relation(Ent1, Ent2, _).
depend_on(Ent1, Ent2) <= relation(Ent1, Ent2, role).
depend_on(Ent1, Ent2) <= relation(Ent1, Ent2, char).

```

Figure 4-16. Definition of some Modeller's inferential literals.

Figure 4-17. Graphical representation of a goal for Modeller.

Then, the interaction given in Figure 4-18 was designed and applied on the empty (initial) state (actions are numbered for future reference). The purpose of the interaction was to provide an opportunity to learn a new way for realizing the goal *gen_entity(A,B,G)*.

The intermediary and final states are given in Figure 4-19 in the internal format. Each state is a list of pairs *literal / [no of the action that asserted the literal]*. The second element of these pairs, the action that contained the literal in its ADD list, is used to compute the causal links (Sections 2.5.2 and 3.2.2).

```

a1:  addE( teacher ),
a2:  addE( department ),
a3:  addR( teacher, department, related ),
a4:  addE( university ),
a5:  addE( student ),
a6:  addE( dept_employee ),
a7:  addR( teacher, dept_employee, char ),
a8:  addR( dept_employee, department, related ),
a9:  delR( teacher, department ),
a10: addR( department, university, role ) )

```

Figure 4-18. An interaction and its resulting state.

State 0

```
State 1    addE(teacher)
           entity(teacher)/[1]
```

```
State 2    addE(department)
           entity(department)/[2]
```

entity(teacher)/[1]

```
State 3    addR(teacher,department,related)
           relation(teacher,department,related)/[3]
           entity(teacher)/[1]
```

entity(department)/[2]

```
State 4      addE(university)
             entity(university)/[4]
             relation(teacher,department,related)/[3]
             entity(department)/[2]
```

entity(teacher)/[1]

State 5	addE(student)	
	entity(student)/[5]	entity(university)/[4]
	relation(teacher,department,related)/[3]	entity(department)/[2]
	entity(teacher)/[1]	
State 6	addE(dept_employee)	
	entity(dept_employee)/[6]	entity(student)/[5]
	entity(university)/[4]	
	relation(teacher,department,related)/[3]	
	entity(department)/[2]	entity(teacher)/[1]
State 7	addR(teacher,dept_employee,char)	
	relation(teacher,dept_employee,char)/[7]	entity(dept_employee)/[6]
	entity(student)/[5]	entity(university)/[4]
	relation(teacher,department,related)/[3]	entity(department)/[2]
	entity(teacher)/[1]	
State 8	addR(dept_employee,department,related)	
	relation(dept_employee,department,related)/[8]	
	relation(teacher,dept_employee,char)/[7]	
	entity(dept_employee)/[6]	entity(student)/[5]
	entity(university)/[4]	
	relation(teacher,department,related)/[3]	
	entity(department)/[2]	entity(teacher)/[1]
State 9	delR(teacher,department)	
	relation(dept_employee,department,related)/[8]	
	relation(teacher,dept_employee,char)/[7]	
	entity(dept_employee)/[6]	entity(student)/[5]
	entity(university)/[4]	entity(department)/[2]
	entity(teacher)/[1]	
State 10	addR(department,university,role)	
	relation(department,university,role)/[10]	
	relation(dept_employee,department,related)/[8]	
	relation(teacher,dept_employee,char)/[7]	entity(dept_employee)/[6]
	entity(student)/[5]	entity(university)/[4]
	entity(department)/[2]	entity(teacher)/[1]

Figure 4-19. Resulting states in internal format. The sequence of states represents the creation of the entity-relationship model for the interaction shown in Figure 4-18.

In the following step, the Modeller domain theory was applied on the interaction defined above, and the goal graph that resulted is shown (slightly edited) in Figure 4-20. In

particular, the goal graph shows that only the actions a10 is explained as a leaf of the goal rule, and that actions a2 and a4 were used to achieve the precondition of the goal rule.

```
make_dependent_on(department,university) (g11, role_dep2)  
precondition actions: a2, a4  
subgoals :  
    a10: addR(department,university,role)
```

Figure 4-20. Goal graph for the interaction of Figure 4-18. The first line indicates the goal, the name of the goal tree (g_{11}) and the name of the goal rule used ($role_dep_2$). The second line indicates the indirect actions (actions used to satisfy the precondition of the goal rule), and the line(s) below the keyword *subgoals* represent the subtasks of the goal rule when ordered from left to right.

Finally, the learning component of LEADER was applied on the goal graph and on the interaction, using Learning Heuristic 2 (learning a new hierarchical plan for a goal when the corresponding goal rule is partially satisfied). The goal $gen_entity(A,B,G)$ defined in Figure 4-15 was achieved by the partially realized corresponding goal rules. In particular, the condition P_0 was true in state 5 (before the creation of the entity *dept_employee* by action a6), and the condition Q_0 was true in state 9 (after the deletion of the relationship), as shown by the proof trees below.

Figure 4-21. Proof trees for P_0 and Q_0 .

Using causal links to retrace the actions responsible for the achievement of Q_0 and to remove the unnecessary actions (such as action a5), one goal rule was proposed, shown in Figure 4-22. The goal rule provides a new way to realize the goal $gen_entity(A,B,G)$, this time by creating a characteristic relationship.

```

gen_entity(A,B,G) <=
  < entity(A) ∧ entity(B) ∧ ¬entity(G)
    ∧ ¬related(G,B) ∧ ¬related(A,G) ∧ relation(A, B, R),
  ( com(addE(G)),
    com(addR(A,G,char)),
    com(addR(G, B, R)),
    com(delR(A, B)) ),
  entity(A) ∧ entity(B) ∧ entity(G) ∧ ¬related(A, B)
    ∧ relation(G, B, R) ∧ relation(A, G, char) >

```

Figure 4-22. Goal rule produced from the interaction of Figure 4-18.

It can be noticed that the precondition of the goal rule is not minimal (as defined in Section 3.2.4). Indeed, because $\neg \text{entity}(G) \Rightarrow \neg \text{related}(G,B)$ and $\neg \text{entity}(G) \Rightarrow \neg \text{related}(A,G)$, these two consequents could be removed from the precondition to produce:

$$\text{entity}(A) \wedge \text{entity}(B) \wedge \neg \text{entity}(G) \wedge \text{relation}(A,B,R).$$

However, this knowledge is domain knowledge, and is not currently incorporated to the domain theory.

As a verification, the new goal rule was added to the initial domain theory and the interaction was again analyzed using this enhanced theory, producing the goal graph shown in the next figure.

```
make_dependent_on(department,university) (g11, role_dep2)
  precondition actions: a2, a4
  subgoals :
    a10: addR(department,university,role)

gen_entity(teacher,department,dept_employee) (g12, new_gr1)
  precondition actions: a1, a2, a3
  subgoals :
    a6: addE(dept_employee)
    a7: addR(teacher,dept_employee,char)
    a8: addR(dept_employee,department,related)
    a9: delR(teacher,department)
```

Figure 4-23. Goal graph after learning for the interaction of Figure 4-18 (for an explanation of the representation, see Figure 4-20).

To summarize, this section has illustrated the application of the different components of LEADER on the domain of databases design. Examples have shown that LEADER can

enhance an initial domain theory about entity-relationship modelling by producing a goal rule usable by typical experts.

CHAPTER 5

Comparison with related work

The system LEADER presented in previous chapter can be described from different perspectives. These points of view correspond to different classes of systems that perform tasks similar to the ones given to LEADER. This chapter compares LEADER to other systems according to the following three classes of systems: program analyzers, planners and learning apprentice systems.

Section 5.1 presents the relationships between LEADER and the techniques used for program analysis. In particular, the similarity between the semantics of programming language statements and goal rules is discussed. Section 5.2 describes how LEADER can be seen as a planning system, and how it differs from several representative planners. In Section 5.3, LEADER is compared with an important class of systems, the learning apprentice systems, according to the main issues involved in these systems. Finally, Section 5.4 brings some additional remarks.

5.1 Comparison with program analysis techniques

As presented in Section 3.2.3.1 and illustrated by Figure 3-1, the concepts of hierarchical plans and conditions possess a number of similarities and differences with classical notions used for program analysis ; this section describes these relationships.

Both concepts of hierarchical plans and statements involve a notion of precondition that defines the prerequisite for their execution. For a hierarchical plan H , the notion of precondition refers to *the weakest precondition under executability of H* (Section 2.3). This is similar to the notion of the *weakest precondition for correctness of a statement H* as used in programming languages:

"The *total correctness formula* $\langle P \rangle H \langle Q \rangle$ is true with respect to interpretation I ($\models_I \langle P \rangle H \langle Q \rangle$) iff for every state σ , if predicate P holds for state σ under interpretation I , there exists a state σ' such that $(\sigma, \sigma') \in M[H]$ ³³ and Q must hold for σ' under I also" [Clarke (Jr) 1985](p.91).

However, in contrast with the definition of precondition needed in LEADER, the above notion guarantees neither the executability of H for all states where its precondition holds, nor the satisfaction of the postcondition in all states obtainable from the initial state. Thus, using such a definition would provide no guaranties that a given goal rule achieves the intended goal.

A more similar and appropriate notion is the *partial correctness* of a statement H :

"The *partial correctness formula* $\{P\}H\{Q\}$ is true with respect to interpretation I ($\models_I \{P\}H\{Q\}$) iff for all states σ and σ' under I , if predicate P holds for state σ under interpretation I and $(\sigma, \sigma') \in M[H]$, then Q must hold for σ' under I also" [Clarke (Jr) 1985](p.91).

Here, as required by LEADER, the postcondition is guaranteed to hold in every state reachable from the initial state.

Another important dimension links together both domains: in data-flow analysis as in LEADER, the semantics (precondition and postcondition) is determined prior to the execution time, and thus cannot be determined with perfect accuracy³⁴. Approximations must then be used, and are required to be *conservative* in the following sense: although the

³³ $M[H]$ is the relation defining all the pairs (initial state σ , final state σ') where σ' can be obtained by executing the statement H over σ .

³⁴ Programming languages use conditional branching, and the problem of deciding whether each path of statements may eventually be taken is undecidable.

information about H can be incomplete (missing something that happens), it cannot be a superset of what really happens. In data-flow analysis, it means to miss optimization opportunities to guarantee no changes in what the program computes; in LEADER, it means having eventually an overly strong precondition or an overly weak postcondition in order to guarantee the executability of the task and the truth value of the postcondition.

The same property of being conservative applies to the computation of available expressions (Section 3.2.3.1): to be conservative means here to ensure that every possible path will produce the given expression. Consequently, available expressions at a given point are obtained by computing the intersection of available expressions produced by each path. This is similar to the definition of hierarchical plan postconditions that also involves the intersection operator (Section 2.3), because the postconditions represent the maximal conditions obtained in every reachable final state.

5.2 Comparison with planners

In the previous section and in Section 3.2.3, the methods used by LEADER were compared with the techniques developed for program analysis. This section presents a comparison of LEADER with an important class of systems, the planners, according to the following point of view: the capability of enhancing a domain theory in order to have a better coverage of the domain. In LEADER, this means learning goal rules to explain a larger part of interactions; in planners, this means building plans to achieve new goals.

To establish this comparison, the following dimensions will be discussed:

- the representation used to represent goals and states
- the representation used to represent the time
- the representation used to represent actions and plans
- the method used to derive (search for) a plan that achieves a given goal
- the method used to prove or verify if a plan achieves a given goal.

Note that these topics may be inter-dependent. For instance, deriving a plan and proving its correctness are usually obtained simultaneously by the planner. As another example, the planner may use a method that depends widely on the representation used.

In the following sections, we will see how these points are handled in LEADER in comparison with some well-known planners such as STRIPS [Fikes and Nilsson 1971], NOAH [Sacerdoti 1975], TWEAK [Chapman 1987] and PRIAR [Kambhampati and Hendler 1990]. The intent is not to describe each system entirely, but to compare them with LEADER. Accordingly, the description of their relevant features is spread among the next sections.

5.2.1 Representation of goals and states

The representation of goal and states is crucial in planning: without goals every action is meaningless, without states no evolution is possible. The following section compares the representation used by systems from which LEADER is derived.

STRIPS [Fikes and Nilsson 1971] is one of the first problem-solving system built for planning. It is an adaptation to planning of GPS (General Problem Solver), a system intended to simulate human performance in search problems (e.g., puzzles, symbolic integration).

Both STRIPS and GPS represent states as sets of fully instantiated literals. A goal is represented as a set (conjunction) of literals that may contain variables; variables are interpreted existentially. Inside both goals and states, only positive and primitive literals are allowed³⁵.

The STRIPS representation has influenced the next generations of planners, including NOAH, TWEAK and PRIAR. STRIPS, TWEAK and PRIAR represented a goal as a set (conjunction) of literals. However, no inferential literals are used in these systems, allowing them to do efficient planning [Kambhampati and Hendler 1990].

This representation was not appropriate for LEADER, because of the use of inferential literals and of the explicit specification and use of postconditions. So, a goal in LEADER is represented by a pair of conditions $\langle P, Q \rangle$ specifying properties that are true before and after the goal achievement (this pair explicitly expresses the traditional $\langle$ initial state, final state $\rangle$ planning problem), and the condition P and Q can refer to inferential literals. As we saw earlier (Section 3.2), this increases the representation power of LEADER by allowing it to use disjunctions inside goals.

Although LEADER allows inferential literals inside goals, only primitive literals are used to describe states, as in the other systems. This allows an efficient maintenance of the states. The non-primitive properties are inferred only when required.

5.2.2 Representation of time

The representation of time is important in planning: it is required to represent that some changes occur over time when actions are executed. An interesting question is the nature of

³⁵ STRIPS were later on augmented to allow inferential literals: "primary relationships" were specified by actions, "secondary relationships" were deduced from the primary ones [Shavlik 1990].

the time, or more precisely, the nature required for the particular planning task [Shoham 1988]. What follows is a list of characteristics the time may have:

- acyclic (that is to say, a partial order) or cyclic
- discrete (between two time points there are a finite number of time points) or continuous
- bounded or unbounded (every point has both a predecessor point and a successor point)
- dense (between any two time points lies a third time point, as for rational numbers)
- complete (if a series of time points is bounded from above by a time point, the least upper point of this series exists and is a time point, as for real numbers)
- branching or linear.

Most systems assume acyclic and linear time. Although the early representations assumed discrete time, particularly in situation calculus, representations using continuous time are more common today [Shoham 1988].

The systems STRIPS, NOAH, TWEAK, PRIAR and LEADER make all use of discrete time, often in an implicit way: a time point is determined by the execution of an action.

In LEADER, the notion of time is required to maintain a linear order over the actions and the corresponding states, and to establish (and limit) the temporal distance between actions when causal links are retrieved to explain actions or to satisfy goals. Under these requirements, linear acyclic and discrete time is sufficient and assumed for LEADER: the state number (which is the action number) is used as the time indicator.

5.2.3 Representation of actions and plans

Describing an action means providing its precondition and its effect. In any planner, both the precondition and the effect are described with a representation compatible with those used to describe goals and states, because they have to be compared to build the appropriate sequence of actions, that is to say, the plan. Assuming discrete time, one may distinguish several types of actions, when considering the information needed to describe the action:

- action whose effect depends on the initial condition

For instance, the action *increment the counter C by 1*:

if $\text{value}(C, N)$ then $\text{EFFECT} = \text{value}(C, N1)$, where $N1 = N + 1$

- conditional action, whose general form is:

if PRECOND is true then $\text{EFFECT} = E_1$ else $\text{EFFECT} = E_2$.

For instance, for the action *fire a gun*:

if gun_is_loaded is true

then $\text{EFFECT} = \text{noise}$ else $\text{EFFECT} = \neg \text{noise}$

A conditional action is always executed on any state; its effect depends on the truth value of the precondition when evaluated on the state. If the action is non-executable, its effect is usually empty (no changes).

- probabilistic action, whose general form is:

if PRECOND is true then $\text{EFFECT} = E_i$ with probability p_i .

The effect depends on a factor not described inside the action, and that modifies the effect according to some probability distribution.

- action having side effects

For instance, the action *turn off the computer*:

if computer_is_on then EFFECT = computer_is_off

Here, the action does not specify that the value of computer's counters becomes undefined. To incorporate this knowledge, thus making the action fully specified, axioms must be used, as in

$$\text{value}(\text{counter}_1, \text{undefined}) \leq \text{computer_is_off}$$

$$\text{value}(\text{counter}_2, \text{undefined}) \leq \text{computer_is_off}$$

$$\text{value}(\text{counter}_3, \text{undefined}) \leq \text{computer_is_off}.$$

Otherwise, all the derived literals must be integrated inside the action's effect, as in

if computer_is_on then

$$\text{EFFECT} = \text{computer_is_off} \wedge \text{value}(\text{counter}_1, \text{undefined})$$

$$\wedge \text{value}(\text{counter}_2, \text{undefined})$$

$$\wedge \text{value}(\text{counter}_3, \text{undefined}).$$

- simultaneous or non-simultaneous action

An action A may take more than one time step for its completion, allowing others actions to begin (and possibly terminate) their execution before the completion of A. For instance: calling an elevator, melting an ice cube.

- fully specified action

The description of the action provides all the information needed to determine its effect with complete accuracy.

All previous types of actions specify what changes result upon the execution of an action. The symmetrical problem, determining and specifying what is left unchanged by an action, is called the *frame problem* [McCarthy and Hayes 1981].

Using the above types of actions, relevant aspects regarding the representation of actions and plans in existing planning systems can now be presented and compared with LEADER.

In STRIPS, an operator (action) is described by its precondition (defining the minimal condition that must hold in the given state prior to its execution), an ADD list (specifying the set of formulas that are changed (added) in the resulting state), and a DEL list (specifying the set of formulas that may no longer be true in the resulting state). Thus, STRIPS uses fully specified actions. Typically, the precondition in STRIPS is an arbitrary formula, the DEL and ADD lists are restricted to a predetermined set of *allowable formulas*, and no actions have an effect that is not described inside the DEL and ADD lists (the *STRIPS assumption*). This guarantees the soundness of the representation [Georgeff 1987].

In TWEAK, each action is represented by a precondition (defining, as for STRIPS, its condition of applicability) and a postcondition (defining the strongest condition that is true in the state resulting from the action application). Both the precondition and the postcondition are assumed to be finite sets of positive and/or negative literals.

As in STRIPS, the TWEAK representation assumes that the domain actions can be described without conditional actions, derived side effects, or dependencies of effects on the initiating state. Despite the representation limitations it imposes, the TWEAK representation is used by a number of problem solvers, particularly planners, mainly

because it neatly bypasses the frame problem and because it leads to efficient planning [Chapman 1987], [Kambhampati and Hendler 1990].

Although the representation used by LEADER inherits from both systems (for instance, LEADER makes also use of the STRIPS assumption), it differs in several ways.

One of the problems with the STRIPS representation is that the formulas appearing in the DEL list are not necessarily false in the resulting model; their truth value is simply unknown. In LEADER, as in TWEAK, this distinction between a literal that is false and a literal whose truth value cannot be assessed is made clear by the explicit use of the postcondition. Instead on relying on the closed-world assumption, LEADER explicitly uses the precondition and postcondition of actions and tasks to compute the semantics of hierarchical plans.

As another example, the TWEAK representation is restricted to conjunctions of primitive literals. The representation used by LEADER allows inferential literals, and by the means of conjunction of literals, it also allows disjunctions inside goals.

5.2.4 Plan construction

This section compares the way LEADER builds hierarchical plans with methods used by planners. In fact, there is little to say because LEADER does not *actively* do planning, as explained below.

In contrast with *real* planners that tries to build a plan for a given goal by either creating it from scratch (by brute planning), by modifying a similar plan (by analogical, derivational or transformational planning) or by any other algorithmic search methods, LEADER

passively plans, as it observes the interaction built by an agent and extracts from it a plan that achieves an observed goal.

Consequently, LEADER does not deal actively with most of the problems, such as conjunctive planning (search for a single plan that achieves simultaneously several – possibly dependent – goals) well illustrated by the *Sussman anomaly* [Sussman 1973], and resource consumption minimization (search for a plan that minimizes resources available), as well as with related techniques of planners (such as non-linear planning and goal regression). Instead, LEADER relies on the interaction provided by the agent (and so on the agent) to solve the problems: LEADER will learn a plan only if the agent was successful in deriving it.

5.2.5 Plan verification

Once a plan is built, it might be required to verify if it really achieves the goal for which it was created. Indeed, in general, the proposed plan may fail to achieve the intended goal for various reasons. For example,

- the plan may contain probabilistic actions (Section 5.2.3) whose effects may vary from one execution to another
- the actions used may produce side effects unspecified in their description (Section 5.2.3)
- the planner may have used assumptions that are not present in the current execution of the plan

- the planner may have used an incomplete domain theory, so the plan is not guaranteed to be successful in achieving the goal in every situation where the plan precondition is satisfied.

For instance, in the case-based planning system CHEF [Hammond 1989] used to create recipes, the plan that is produced (by modifying another plan achieving a goal similar to the one desired) may fail for three reasons:

- undesirable side effects arise (for instance, the meal has a bad taste)
- a part of the desired goal is not achieved
- the precondition of one of its actions is not entirely satisfied.

In CHEF, the failure is first detected by a simulator (that uses a complete theory of the domain), then an explanation of the failure is provided by the simulator, and anticipations are added to the domain knowledge used by the planner in order to avoid that failure in future plans.

In LEADER, no such verification is required, again because of the way the plan is generated. To construct the plan, LEADER identifies relevant actions of a given interaction (that is thus executable), builds the corresponding hierarchical plan, and then computes its precondition and postcondition. This approach is guaranteed to produce correct plans because of the following:

- there are no probabilistic actions in the domain
- all the actions are fully specified

- the precondition of every action (and hierarchical plan) guarantees its complete execution
- the plan is built by extracting actions from an (observed) executable interaction. In particular, when the Action Executor (Section 3.1.2) executes the actions in order to create the sequence of contiguous states and the causal links, there is no need for monitoring the feasibility of the given actions.

5.3 Comparison with learning apprentice systems

To help the knowledge acquisition from experts, a class of promising knowledge-based systems has been introduced in the literature: the Learning Apprentice Systems (LAS). [Mitchell and Mabadevan 1990] defines these systems as:

"the class of interactive, knowledge-based consultants that directly assimilate new knowledge by observing and analyzing the problem-solving steps contributed by their user through their normal use of the system"
(p. 271).

In what follows, we discuss how LEADER fits in this class, and we compare features of LEADER with representative systems such as ARMS [Segre 1988], CAP [Dent, et al. 1992], CLERK [Campbell 1990], DISCIPLE [Tecuci and Kodratoff 1990], LEAP [Mitchell and Mabadevan 1990][Mitchell, Utgoff et al. 1983], ODYSSEUS [Wilkins 1990].

Again, the intent is not to describe each system entirely, but to establish a comparison with LEADER. So, the description of the systems is spread among the next sections, according to the following dimensions:

**PM-1 3½"x4" PHOTOGRAPHIC MICROCOPY TARGET
NBS 1010a ANSI/ISO #2 EQUIVALENT**

PRECISIONSM RESOLUTION TARGETS

- the nature of the initial knowledge base
- the nature of the interaction between the agent and the system
- the nature of the knowledge used by the system
- the methods used to acquire this knowledge.

5.3.1 Nature of the initial domain theory

The information initially provided to the learning system plays a crucial role in its ability to produce useful new knowledge. This information can be characterized according to several criteria, such as the nature of the initial knowledge (domain-specific or domain-independent information) and its quality (completeness, correctness, consistency, uniqueness). These criteria are discussed in the following paragraphs. Existing learning systems are used to illustrate similarities and differences with LEADER.

The initial information can be provided as domain theory or as domain-independent knowledge. Examples of domain-specific knowledge are: rules for VLSI design, *control knowledge* [Minton 1988] (information specifying which rule to choose among several ones whose precondition is satisfied) and definition of actions in a given domain for the use of a planning system. Examples of domain-independent knowledge are: general design methodologies, metarules and heuristics.

Usually, learning systems use both kind of knowledge. For instance, ODYSSEUS [Wilkins 1990], a learning apprentice system designed to refine a domain theory, uses domain-specific knowledge in the form of rules for medical diagnosis. Its domain-independent knowledge consists in metarules for explaining physician's behaviour, and in heuristic methods such as the *sociopathic reduction algorithm* used to extract from a domain theory a subset that gives better performance.

As another example, in the LEAP system [Mitchell and Mabadevan 1990], designed to acquire new knowledge in VLSI by observing design steps, the domain-specific knowledge takes the form of design rules associating a functional specification and input signals with a particular circuit that realizes the function, and of algebraic transformation rules used to show the equivalence between circuits. The domain-independent knowledge is essentially embedded within the system as explanation and generalization methods.

This separation of the domain-specific knowledge from the heuristic knowledge is also present in LEADER. The domain theory provides the semantics for actions as well as the specifications of goals and goal rules, and the domain-independent knowledge takes the form of the methods that analyze the interactions (Section 3.1) and of the different heuristics used by the learning module (Section 3.2). As for the previous systems, this clear separation allows LEADER to be used in various domains by modifying only the domain theory (Chapter 4).

Several parameters can be used to assess the quality of the domain theory, as shown by Table 5-1.

Parameter	Description
Completeness	Proportion of instances covered by the theory
Correctness	Proportion of instances correctly covered by the theory
Consistency	Proportion of instances covered without contradictions by the theory
Uniqueness	Proportion of instances not covered by two different parts of the theory

Table 5-1. Parameters for assessing the quality of a domain theory.

The main assumption made by LEADER about the quality of the domain theory is its *incompleteness*. Indeed, the purpose of the learning heuristics is to provide ways currently unknown from the domain theory to achieve goals: if the theory is complete, no learning can occur. This contrasts with pure explanation-based learning (EBL) systems [DeJong and Mooney 1986], or with some hybrid systems such as LEAP, where learning occurs only when the theory is complete enough to explain the training instance.

As LEADER does, the system ODYSSEUS also addresses the problem of incompleteness. In addition, the two other problems, incorrectness and inconsistency, are also of concern in ODYSSEUS. These problems are not taken into account by LEADER: if they occur in the theory, the knowledge proposed may be also incorrect or inconsistent. However, LEADER can still produce appropriate knowledge in these pathological situations, as observed in one experiment where the domain theory involuntarily contained an incorrect goal rule, and where LEADER proposed as new knowledge the correct version of the goal rule.

The uniqueness parameter is used in some learning systems. For instance, ODYSSEUS uses it as a bias to filter proposed rules prior their insertion in the domain theory. In LEADER, no automatic filtering is applied on the proposed knowledge: the rule filtering, enhancement and integration to the current domain theory is done by the expert during its verification step.

5.3.2 Nature of the interaction between the agent and the system

The interaction between the agent (the teacher) and the system may go from *rote teaching*, where the agent provides the information to the learner in a direct manner similar to programming, to *passive teaching*, the other extreme where the agent is assumed to be unaware of the learner's presence.

In some cases, learning is perturbed by the agent because it interferes with the knowledge acquired. For instance, although one can teach someone correct movements for playing tennis by holding the racket with the person while hitting the ball, the approach is not appropriate for teaching the correct force to apply on the racket, because part of the required force necessarily provided by the teacher cannot be measured by the learner. This knowledge must be learned by other means, for instance by hitting the ball without the physical assistance of the agent.

In some other cases, it is the learner that perturbs the agent because it interferes with the normal actions of the agent whose main task might be to solve problems, not to teach how to solve them. Ideally, learning should be totally unsupervised in order to avoid perturbing the agent. However, to reduce the complexity of the learning task, the learner usually relies on the agent to provide part of the learned knowledge. An important drawback is that the quality of the learning depends on the level of the agent's expertise; learning from a novice might produce poor knowledge.

In LEADER, the learning is totally unsupervised: the agent provides no other information than the interaction produced through normal operations. This type of learning is also present in the case-based learning system CHEF [Hammond 1989]; however, the independence of the learner relies on a considerable built-in domain knowledge. The domain theory is not so important for LEADER because having an incomplete theory might only make LEADER miss a learning opportunity (when the interaction contains a goal achieved in way similar to the missing rule): it will not *fail* to solve a given goal because no such goal is specified.

This unsupervised learning approach contrasts with most learning systems. For instance, the ARMS system for robot control [Segre 1988] requires that the teacher guides the robot arm through a series of movements that achieve a given goal.

5.3.3 Nature of the knowledge used by the system

The interaction of the agent with a learning apprentice system may provide three types of knowledge, called here the *what*, *how* and *why* knowledge. They represent, respectively, what the agent achieves (the goal), how he or she achieves it (the implementation), and why he or she does so (the motivation).

If the agent employs a goal structure to guide the interaction, then each node of this goal structure incorporates the three types of knowledge: the *what* knowledge corresponds to the node, the *how* knowledge is the children of the node (and possibly all its descendants), and the *why* knowledge corresponds to the structure above the node that justifies its presence, and particularly, it refers to the control knowledge [Minton 1988]³⁶.

In games, for instance, the *what* knowledge is provided by the rules specifying the winning and losing states, the *how* knowledge represents a particular move (or a short sequence of planned moves), and the *why* knowledge corresponds to a particular game hierarchical plan, that is to say, it is the motivation for the preferred move among legal options.

As another example, in the LEAP system [Mitchell and Mabadevan 1990], a training instance usually consists of the following three components: the functional specification of the desired circuit, a description of the input signals, and a circuit that realizes the given function. So, all three types of knowledge are provided by the agent: the *what* knowledge

³⁶ One may refer to the control knowledge as a *when* component, because it indicates the conditions where the application of the task is preferable. However, we include this notion of preferability in the *why* component, i.e., in the justification of the task steps.

is the functional specification, the *how* knowledge is the implementation of this function. and the *why* knowledge corresponds to the design choices done while achieving the implementation (e.g., why preferring a particular circuit to implement a given function, based on the input signals).

For a particular learning apprentice system, not all three types of knowledge may be provided by the agent. For instance, in CLERK [Campbell 1990], a system designed to learn tasks accomplished by an electronic-mail system user, the agent provides only the task name (the *what* part) and the sequence of actions defining the task (the *how* part). The *why* component is neither provided to the system nor learned by it.

In DISCIPLE [Tecuci and Kodratoff 1990] also, a multistrategy learning system that integrates explanation-based learning (EBL), learning by analogy, empirical learning and learning by questioning the agent, only the first two components are provided. DISCIPLE uses EBL to derive an explanation of the plan provided by the agent (the *how* part) in order to solve a given goal (the *what* part). Using the generalization of the plan and the explanation, the system looks in the knowledge base for similar plan instances and ask the agent to characterize them as positive or negative examples in order to use them to refine the initial solution. These instances and characterizations are a mix of *what* and *how* knowledge without the *why* component, because the agent does not justify its characterization.

As another example, in ODYSSEUS [Wilkins 1990], the *how* part is given in the form of a sequence of questions from a physician to a patient, along with the patient answers. Then, using metarules, the system builds an explanation to justify each question, and learning occurs by modifying the domain theory to take into account an explanation failure. The

what part is implicitly provided to ODYSSEUS because each session represents a single diagnosis and the system's goal is to explain every question that lead to *that* diagnosis.

The same level of implicit goals also appears in CAP [Dent, et al. 1992], an apprentice system for managing meeting calendars – which exhibits as LEADER a form of unsupervised learning as its learning components (decision tree learning modules and neural networks) are invoked offline. Indeed, in this system, *the* goal is primarily to establish (suggest) meeting parameters (duration, location, etc.), based on (training) instances consisting in the description of the parameters of meetings that have occurred.

In contrast with these systems, LEADER receives from the agent only the *how* component, that is to say, the actual interaction that may achieve several goals, with no annotations about the goals achieved (*what*) or the reasons for their achievement (*why*).

Furthermore, in contrast with the above systems that receive only one training instance at a time, LEADER receives an entire interaction that constitutes several training instances. So, LEADER has to extract from it the three knowledge components: the boundaries of the sub-interactions that delimit the training instances and within them the relevant actions that achieve the goals (*how*), the goals achieved (*what*) and decomposition of the goals into subgoals via the hierarchical plans (*why*).

5.3.4 Methods used to acquire this knowledge

One can identify several ways an instructable system may acquire knowledge to solve new tasks, as shown in Table 5-2 (taken from [MacDonald 1991]).

Method	Example
Specification	Drawing of an electric motor for assembly
Formal procedure	Assembly instructions for a motor
Program	Sorting code
Abstract plan	Sub-goals in an assembly task
Concrete plan	"Grasp BOLT.1, Align BOLT.TIP with HOLE.3"
Rules	"Prescribe ABC when you see XYZ"
Advice	"Don't use crescent wrenches"
Comments	"Aluminium has no fatigue limit"
Guiding	(Sequence of robot arm positions)
Show	(Watch a human assembling a motor)
Built-in	(Sequence to pick up an object)

Table 5-2. Methods for transferring new tasks.

Among these methods, learning by showing ways to solve typical problems is the method the most similar to the one used by LEADER. Indeed, LEADER learns by analyzing user interactions that solve problems (that is to say, achieve goals). The main differences are that a given interaction may solve several problems (or even no problems at all) and that no indications is provided to the learner as to which problems are solved and which part of the interaction solves them. LEADER must learn *what* is solved and *how* it is solved.

Other differences exist according to the way the training instances are used. For instance, in EBG, EBL and LEAP, the domain theory is assumed to be sufficient to explain the training instance, and learning occurs by generalizing the explanation of the instance. In contrast with these systems, in LEADER as in ODYSSEUS, learning occurs when the domain theory and the inference method are not sufficient to explain the training instance. Also, in usual learning systems, the generalization occurs over the objects manipulated by the plan. In LEADER and in [Prieditis 1990], the generalization occurs over the plan itself.

5.4 Concluding remarks

The previous section have compared LEADER with three classes of systems: program analyzers, planners and learning apprentice systems. This section brings additional issues involved in LEADER not discussed above.

LEADER can be compared to the systems that learn classification rules, by establishing the following correspondence: the class is associated to the goal, the class description corresponds to the task, and the training instance is a portion of the agent interaction. In such typical learning systems, the intention is to learn the description of a class, giving several instances with their classification. For instance, in Refiner [Sharma and Sleeman 1988], designed to infer prototype descriptions for a particular classification, given correctly classified training instances as being prototypical or non-prototypical for the given class, the class (goal) and the instances (the relevant parts of the interaction) are provided, and the description (task) is learned. In contrast with Refiner, LEADER has to identify the class (that is to say, the classification of the instances) before learning the description of the class, giving the instances.

LEADER does not address the *utility problem* [Minton 1988], that is to say, the problem of finding a compromise between two phenomena caused by an increase of the number of rules in the domain theory: an increase of the theory coverage, and a decrease of efficiency (in search time) by having more rules to process when searching for a solution. Although LEADER learns the condition of application of a goal rule (the precondition), it does not indicate when to prefer that rule to another applicable rule that achieves the same goal.

Another topic not addressed by LEADER is the *falling off the knowledge cliff problem*, that is to say, the rapid deterioration of the performance when a system is applied to

problems slightly beside the scope of its current knowledge. Let us mention two successful approaches to this problem: having a system that integrates different learning paradigms [Tecuci and Kodratoff 1990] (when one of them fails, the others may succeed), and having the possibility of building new terms to the theory, as in classifier systems [Holland 1986] or as in [Lapointe, Ling et al. 1993].

CHAPTER 6

Conclusion

This thesis has addressed the knowledge acquisition problem, that is to say, the problem of extracting domain knowledge from human experts for its use in computer systems. This practical problem is at the intersection of many fields such as planning, learning and advising. This multi-disciplinary aspect of the problem makes it important and interesting from the application perspective.

As an approach to the problem, the thesis has proposed non-obtrusive learning methods usable in situations where experts perform their tasks through sequences of actions resulting from the decomposition of goals into subgoals. Section 6.1 states the conclusions of the thesis, and Section 6.2 proposes directions for future research.

6.1 Conclusions

The conclusions of the present work are the following:

- The problem stated in Section 1.1 (achieving unsupervised learning based on an incomplete domain theory) has been solved by the means of a computer system that performed unsupervised and non-obtrusive learning of plans by analyzing instances of experts' interactions in two domains of application.
- Given a domain theory about goals and plans, explanation-based learning is an efficient way to extract pertinent information from experts' interactions.

- Regarding the TWEAK⁺ representation that we have introduced in Section 3.2.3.1, the following was obtained:

for the propositional logic without inferential literals, an efficient solution (linear time and space) in the form of bit vector equations is obtained for computing the description (precondition, postcondition, ADD and DEL lists) of hierarchical plans.

augmenting TWEAK⁺ with inferential literals where axioms are implication formulas in the disjunctive normal form $c \leq p_1 \vee p_2 \vee \dots \vee p_n$ can still be accommodated by the bit vector equations in certain situations. These situations are characterized by the absence of disjunctions in the definition of inferential literals as well as negations of inferential literals in the description of the semantics of actions.

in other situations (i.e., in the presence of disjunctions) the bit vector equations cannot be applied, and heuristic methods for computing precondition and postcondition of hierarchical plans can be used.

- Regarding the experiments, the following was obtained:

the application to the Office Tasks domain shows that LEADER can be used to enhance significantly a domain theory: it can increase the coverage of the theory (the number of situations taken into account by the theory) by about 10%.

the application to the Modeller domain shows that LEADER can produce useful enhancements to the domain theory in a practical domain.

Learning Heuristic 1 (learning a new hierarchical plan for the first task of a goal rule) is appropriate for the Office Tasks domain.

Learning Heuristic 2 (learning a new hierarchical plan for a goal when the corresponding goal rule is partially satisfied) is appropriate for the Modeller domain, but is not appropriate for the Office Tasks domain.

for the Office Tasks domain, the number of rules proposed by Learning Heuristic 1 increases almost linearly with the length of the interactions, in contrast with the relative coverage. The relative gain of coverage decreases with the number of rules proposed.

- Regarding the use of the DEL/ADD lists mechanism, the following was obtained:

negated literals can be avoided, inferential literals are useless, variable sharing and common literals are useful and dangerous, redundant literals are useful.

6.2 Future research

The thesis addresses to different extents the aspects of the knowledge acquisition problem presented in Section 1.1, which include: the construction of an initial knowledge base, the improvement of the existing knowledge, the automatic adaptation of the learning modules to users, and the elaboration of design principles and techniques. The following paragraphs discuss how LEADER could be enhanced according to these aspects.

As mentioned before, the first aspect (building an initial theory) is not covered by the thesis because this task requires a large amount of direct interaction with experts that does not fit

well with the unsupervised nature of LEADER. However, a front-end module could be added to LEADER to perform this task. For instance, the knowledge base produced for a domain using LEADER could be used by experts as a starting point for building the domain theory of similar domains. This might be possible, for instance, with high level design methodologies (such as *to achieve a given task, decompose it into simpler ones*).

The second aspect (improvement of the theory) divides itself into the detection of a deficiency, the suggestion of a repair for it and the validation of the repair. Among the possible deficiencies (incorrectness, inconsistency, incompleteness, inefficiency), only the problem of incompleteness is directly handled by LEADER, by the means of the proposal of new goal rules. As mentioned in Section 3.2.1 through the analysis of the achievement vector, there is a place in LEADER for integrating also the detection and proposal of missing goals, an enhancement that in turn will increase the proposal of goal rules (because goal rules are built by collecting the tasks achieving goals).

The problem of inefficiency could be addressed by detecting and removing overly specific (subsumed) goals. However, the removal of inefficient goal rules (e.g., those including establisher-clobberer tasks) could lead to a decrease of the coverage because the domain theory might no longer be able to recognize their expansions inside interactions. This decrease would be harmful to LEADER because learning is based on the explanation of interactions.

Further work is clearly possible on the plan of validating LEADER's proposals. We propose here three additional measures to evaluate the goal rules produced by LEADER. The first measure would determine the level of the rule, that is to say, the depth of the explanation tree (0 for the command level, 1 for the level above, and so on). This would provide a tool for assessing LEADER's capability for producing (high level) hierarchical

plans, a type of explanatory structure that people prefer to shallow explanations, even if the shallow ones have more coverage of the cases ([Gick and Matwin 1991]). The second measure would provide a quantitative assessment of rules' usefulness, by establishing statistics on the frequency utilization of the rules in real interactions. The third measure would provide a qualitative assessment of rules' usefulness, as follows:

Rank 2 : the rule is essential (it corresponds to a new approach)

Rank 1 : the rule is interesting (it is a variation of another rule)

Rank 0 : the rule is useless, redundant or inefficient.

In contrast with this first two measures that can be computed automatically, the last one requires the expertise of a human. However, it can be of a great help for assessing the increase in the quality of the domain theory upon learning.

To conclude, additional experiments with LEADER could be performed, in the two domains analyzed here as well as in other domains, as an attempt to answer general important questions about learning systems, such as

How many training instances are required for the system to produce a useful domain theory?

How well the system can adapt itself to evolving domains or experts?

How well the quality of rules automatically generated compares with hand crafted rules?

To what extent can human experts' role in learning be reduced?

REFERENCES

- Aho, A. V., J. E. Hopcroft and J. D. Ullman (1974), The Design and Analysis of Computer Algorithms, Addison-Wesley Publishing Company.
- Aho, A. V., R. Sethi and J. D. Ullman (1986), Compilers: Principles, Techniques, and Tools, Addison-Wesley Publishing Company.
- Campbell, B. (1980), "Office Clerk II", Final year project report, University of Calgary.
- Chapman, D. (1987), "Planning for Conjunctive Goals", *Artificial Intelligence*, vol. 32, pp.333-377.
- Clark, P. and T. Niblett (1989), "The CN2 Induction Algorithm", *Machine Learning*, vol. 3, no 4, pp.261-284.
- Clarke (Jr), E. M. (1985), "The Characterization Problem for Hoare Logics" in Mathematical Logic and Programming Languages, Prentice-Hall International, pp.89-106.
- DeJong, G. F. and R. Mooney (1986), "Explanation-Based Learning: An Alternative View", *Machine Learning*, vol. 1, pp.145-176.
- Dent, L., J. Boticario, J. McDermott, T. M. Mitchell and D. Zabowski (1992), "A Personal Learning Apprentice", *Proceedings of the Tenth National Conference on Artificial Intelligence*, San Jose, California, AAAI Press/The MIT Press, pp.96-103.
- Elkan, C. (1989), "Incremental, Approximative Planning", Rapport KRR-TR-89-12, Department of Computer Science, University of Toronto, November 1989.
- Fikes, R. E. and N. J. Nilsson (1971), "STRIPS: A New Approach to the Application of Theorem Proving to Problem Solving", *Artificial Intelligence*, no 2, pp.198-208.
- Filman, R. E. (1989), "Reasoning with Worlds and Truth Maintenance" in Artificial Intelligence Tools and Techniques, Alex Publishing Corporation, pp.107-149.
- Genest, J. and S. Matwin (1990), "Building Systems Specification using Explanation-Based Learning with Incomplete Theory", Technical Report TR-90-01, University of Ottawa, January 1990.
- Georgeff, M. P. (1987), "Planning", *Annual Review of Computer Science*, vol. 2, pp.359-400.
- Gick, M. and S. Matwin (1991), "The Importance of Causal Structure and Facts in Evaluating Explanations", *Proceedings of the Eighth International Workshop on Machine Learning*, Evanston, California, Morgan Kaufmann, pp.51-54.

- Ginsberg, A. (1989), "Knowledge Base Refinement and Theory Revision", *Proceedings of the Sixth International Workshop on Machine Learning*, Ithaca, N.Y., Morgan Kaufmann, pp.260-265.
- Ginsberg, M. L. and D. E. Smith (1988), "Possible Worlds Planning I + II", *Artificial Intelligence*, vol. 35.
- Guha, R. V. (1990), "Micro-theories and Contexts in Cyc, Part I: Basic Issues", Technical Report ACT-CYC-129-90, Microelectronics and Computer Technology Corporation, June.
- Hammond, K. J. (1989), Case-Based Planning: Viewing Planning as a Memory Task, vol. 1, Perspectives in Artificial Intelligence, Academic Press Inc.
- Holland, J. H. (1986), "Escaping Brittleness: The Possibilities of General-Purpose Learning Algorithms Applied to Parallel Rule-Based System" in Machine Learning, vol. II, pp.593-623.
- Kambhampati, S. and J. A. Hendler (1990), "A Validation Structure Based Theory of Plan Modification and Reuse", *Artificial Intelligence*.
- Kambhampati, S. and C. Jengchin (1993), "Relative Utility of EBG Based Plan Reuse in Partial Ordering vs Total Ordering Planning", *Proceedings of the AAAI-93*.
- Lapointe, S., X. C. Ling and S. Matwin (1993), "Constructive Inductive Logic Programming", *Proceedings of the Thirteenth International Joint Conference on Artificial Intelligence*, Chambéry, France, Morgan Kaufmann, (to appear).
- Levi, K. R., D. L. Perschbacher and V. L. Shalin (1988), "Learning Plans and Information Requirements for Pilot Aiding", *Proceedings of the Third Knowledge Acquisition for Knowledge-Based Systems Workshop*, Banff, Canada.
- Levi, K. R., D. L. Perschbacher and V. L. Shalin (1989), "Identifying Knowledge Base Deficiencies by Observing User Behavior", *Proceedings of the Sixth International Workshop on Machine Learning*, Ithaca, N.Y., Morgan Kaufmann, pp.296-301.
- Lewis, C. (1988), "Why and How to Learn Why: Analysis-based Generalization of Procedures", *Cognitive Science*, vol. 12, pp.221-256.
- Ling, X. and M. Valtorta (1991), "Revision of Reduced Theories", *Proceedings of the Eighth International Workshop on Machine Learning*, Evanston, California, Morgan Kaufmann, pp.519-523.
- MacDonald, B. (1991), "Instructable Systems", *Knowledge Acquisition*, no 3, pp.381-420.

- Matwin, S., F. Oppacher, U.-M. O'Reilly and B. Pelletier (1991), "Unsupervised Learning of Design Rules Aided by System Derived Heuristics", *Proceedings of the First International Conference on Artificial Intelligence in Design*, Edinburgh, U.K.
- McCarthy, J. M. and P. J. Hayes (1981), "Some Philosophical Problems from the Standpoint of Artificial Intelligence" in Readings in Artificial Intelligence, Los Altos, California, Morgan Kaufmann, pp.431-450.
- Michalski, R. S., I. Mozetic, J. Hong and N. Lavrac (1986), "The Multi-Purpose Incremental Learning System AQ15 and its Testing Application to Three Medical Domains", *Proceedings of the Fifth AAAI*, pp.1041-1045.
- Minton, S. (1988), Learning Search Control Knowledge: An Explanation-Based Approach, Knowledge Representation, Learning and Expert Systems, Kluwer Academic Publishers.
- Mitchell, T., P. Utgoff and R. Banerji (1983), "Learning by Experimentation" in Machine Learning, vol. I, pp.163-190.
- Mitchell, T. M., R. M. Keller and S. T. Kedar-Cabelli (1986), "Explanation-Based Generalization: A Unifying View", *Machine Learning*, vol. 1, no 1, pp.47-80.
- Mitchell, T. M. and S. Mabadevan (1990), "LEAP: A Learning Apprentice for VLSI Design" in Machine Learning, vol. III, pp.271-289.
- Navathe, S. B. (1988), "Conceptual and Logical Database Modelling: Survey, Applications and Research Issues".
- Nilsson, N. J. (1980), Principles of Artificial Intelligence, Los Altos, CA, Morgan Kaufmann Publishers.
- Pelletier, B. and S. Matwin (1992), "Building Macros in Deterministic and Non-Deterministic Domains", *Proceedings of the Ninth Canadian Conference on Artificial Intelligence*, Vancouver, B.C., Morgan Kaufmann, pp.15-21.
- Prieditis, A. E. (1990), "Discovering Algorithms from Weak Methods" in Machine Learning, vol. III, pp.351-359.
- Quinlan, J. R. (1990), "Learning Logical Definitions From Relations", *Machine Learning*, vol. 5, no 3, pp.239-266.
- Quinlan, J. R. (1993), C4.5: Programs for Machine Learning, San Mateo, California, Morgan Kaufmann.
- Quintus Computer Systems, Inc. (1987), Quintus Prolog Reference Manual. Version 10,

- Sacerdoti, E. D. (1975), "The Nonlinear Nature of Plans", *Proceedings of the Fourth Internationale Joint Conference on Artificial Intelligence*, Tbilisi, Georgia, USSR, pp.206-214.
- Segre, A. (1988), Machine Learning of Robot Assembly Plans, Kluwer Academic Publishers.
- Sharma, S. and D. Sleeman (1988), "REFINER: A Case-Based Differential Diagnosis Aide for Knowledge Acquisition and Knowledge Refinement", *Proceedings of the Third European Working Session on Learning*, Turing Institute, Glasgow, Pitman Publishing, pp.201-210.
- Shavlik, J. W. and G. F. Dejong (1990), "Acquiring General Iterative Concepts by Reformulating Explanations of Observed Examples" in Machine Learning, vol. III, pp.302-350.
- Shoham, Y. (1988), Reasoning About Change: Time and Causation from the Standpoint of Artificial Intelligence, The MIT Press Series in Artificial Intelligence, The MIT Press.
- Sussman, G. J. (1973), A Computational Model of Skill Acquisition, Ph.D. Thesis, M.I.T., Cambridge, MA.
- Tauzovich, B. (1990), "An Expert System for Conceptual Data Modelling" in Entity-Relationship Approach to Database Design and Querying, Elsevier Science Publisher, B.V. [North-Holland], pp.205-220.
- Tecuci, G. and Y. Kodratoff (1990), "Apprenticeship Learning in Imperfect Domain Theories" in Machine Learning, vol. III, pp.514-551.
- Wilkins, D. C. (1990), "Knowledge Base Refinement as Improving an Incorrect and Incomplete Domain Theory" in Machine Learning, vol. III, pp.493-513.

Appendix A

Terminology

This glossary provides definition of main terms and notations used in the thesis.

Achievement vector

For any goal $G = \langle P_0, Q_0 \rangle$, any conditions P and Q , and any states S and S' , the corresponding achievement vector is the 4-tuple $AV = (P(S), P_0(S), Q(S'), Q_0(S'))$.

Action

An action is a command whose arguments are fully instantiated.

Axiom

An axiom is an implication formula on the form " $L \Leftarrow L_1 \wedge L_2 \wedge \dots \wedge L_n$ ", where L (called the consequent) is a positive literal, and L_i 's (called the antecedents) are positive or negative literals. The current set of axioms is denoted AXMS.

AXMS

AXMS is the set of axioms of the current theory.

Causal link

Let $GG = \langle N = \{n_1, n_2, \dots, n_k\}, PC \rangle$ be a goal graph. A causal link for GG according to the current set of axioms AXMS is a 4-tuple $\langle n_i, e, c, n_j \rangle$, where

- (1) $n_i < n_j$
- (2) $e \in \text{postcond}(n_i)$
- (3) $e \rightarrow c$
- (4) $c \in \text{precond}(n_j)$.

For each literal u needed for the precondition of the task n_j and assumed under the closed-world assumption, the causal link $\langle a_0, u, u, n_j \rangle$ is created. The set of all causal links for GG is denoted $CL(GG)$, or simply CL .

For a particular goal $G = \langle P_0, Q_0 \rangle$, a special action a_G is added to the goal graph to represent the realization of G . The precondition and postcondition of action a_G are both Q_0 . This action is inserted right after the last action a_{iG} achieving Q_0 : $a_{iG} < a_G < a_{iG+1}$.

The causal link $\langle n_i, e, c, a_G \rangle$ is created for each node $n_i \in N$ such that

- (1) $n_i < S_{iG}$
- (2) $e \in \text{postcond}(n_i)$
- (3) $e \rightarrow c$
- (4) $c \rightarrow Q_0$.

Command

The agent modifies the state via *commands*. A command C is represented by " $C = \langle \text{precond}(C), \text{DEL}(C), \text{ADD}(C), \text{postcond}(C) \rangle$ ", where C specifies the name of the command and its arguments, where $\text{precond}(C)$ (the precondition of C) specifies the "weakest" condition that must hold in the current state for C to be executable, and where $\text{postcond}(C)$ (the postcondition of C) specifies the "strongest" condition obtained in the state after the execution of C . The components $\text{DEL}(C)$ and $\text{ADD}(C)$ provide, respectively, the literals to be deleted or added to the current state to produce the state resulting from the execution of the command.

Condition

(see Property)

Direct action

An action is *direct* if it appears in the expansion of a goal rule used to achieve a goal. The action is *indirect* if it is required to achieve the precondition of a goal rule used to achieve a goal (some of the postcondition of the action appears in the goal rule precondition).

Directly supporting action

An action is *directly supporting* a condition Q if some effect of it contributes to Q . An action is *indirectly supporting* a condition Q if it is directly supporting Q or if it contributes to the precondition of some action that is directly supporting Q .

Distance

(see Sequence)

Expansion

An expansion of a hierarchical plan H is a hierarchical plan obtained by expanding all tasks and derived sub-tasks of H down to the level of commands. An expansion of a goal rule $G \Leftarrow \langle P, H, Q \rangle$ is a goal rule where the hierarchical plan is an expansion of H .

A goal rule (or a hierarchical plan) is deterministic if it has only one expansion.

Goal

The semantics of a goal G is a pair $\langle P_0, Q_0 \rangle$ (denoted $G = \langle P_0, Q_0 \rangle$), where P_0 and Q_0 are properties. P_0 , called the goal feasibility property, represents the relevant facts about G allowing its satisfaction; Q_0 , called the goal achievement property, represents those facts that are true after its achievement.

Goal graph

A goal graph GG over the interaction I is a pair $\langle N, PC \rangle$, where N is a set of tasks (nodes), and PC defines the parent-child relations between elements of N according to the goal rules in the theory.

For a particular goal $G = \langle P_0, Q_0 \rangle$, a special action a_G is added to the goal graph $GG = \langle N, PC \rangle$ to represent the realization of G . The precondition and postcondition of action a_G are both Q_0 . This action is inserted right after the last action a_{iG} achieving Q_0 : $a_{iG} < a_G < a_{iG+1}$

Goal rule

A goal rule is a rewrite rule whose left part is a goal and whose right part has three components: a precondition, a hierarchical plan and a postcondition. A goal rule for a goal G is denoted $G \Leftarrow \langle P, H, Q \rangle$, where H , P , Q are, respectively, a hierarchical plan that achieves the goal G , the precondition of the goal rule, and the postcondition of the goal rule. In order for G to be desirable, we have $P \Rightarrow P_0$; in order for G to be achieved, we have $Q \Rightarrow Q_0$.

Goal tree

A goal tree for an interaction I and a goal rule R is a tree contained in the goal graph of I , and whose root is (unifies with) the left part of R . Tasks, goal rules, hierarchical plans, preconditions and postconditions used in that goal tree are said to be realized by the sub-interaction formed by the ordered leaves of the goal tree; these leaves are then explained by the goal tree. A goal graph for an interaction I is said to explain I if each action of I (excluding a_0) is explained by a goal tree.

Hierarchical plan

A hierarchical plan is an ordered sequence of tasks. The semantics of a hierarchical plan H is described by the triple $\langle P, H, Q \rangle$, where P and Q are, respectively, the precondition and the postcondition of H .

Indirect action

(see Direct action).

Indirectly supporting action

(see Directly supporting action).

Interaction

An interaction is an ordered sequence of actions. The interaction is denoted $I = (a_1, a_2, \dots, a_k)$, where a_i 's are the actions. The membership of the action a_i in I is denoted $a_i \in I$ ($0 \leq i \leq k$). (see Sequence)

Literal

A literal is a predicate (positive literal), or the negation of a predicate (negative literal). The negation is denoted by the connective " $\neg$ ". A literal is called "inferential" if it has the same functor and the same number of arguments as the left part of some axiom. Otherwise, it is called "primitive".

Plan

A plan is a hierarchical plan containing only commands.

Postcondition

The postcondition of a task (or a hierarchical plan) T is the strongest condition obtained after the execution of T , under its precondition.

Precondition

The precondition of a task (or a hierarchical plan) T is the weakest condition that must hold for T to be executable.

Predicate

A predicate is composed of two parts: a name (called functor) and possibly some ordered arguments.

Proof tree

A proof tree for a literal c is a tree whose root is c and showing how c is proven in terms of other literals, using the current set of axioms AXMS.

Given a literal c and a proof tree PT for c , a literal u *contributes* to c , denoted " $u \rightarrow c$ ", if u is a node in PT .

Property

A property (also called condition) is a conjunction of literals. The conjunction is represented by the connective " $\wedge$ ".

Satisfaction

A hierarchical plan is a -satisfied ($0 \leq a \leq 1$) if the proportion of its tasks that are realized is a . A condition is a -satisfied ($0 \leq a \leq 1$) if the proportion of its conditions that are satisfied is a .

A hierarchical plan (or a condition) is partially satisfied if it is a -satisfied, with $a < 1$. It is almost satisfied if it is a -satisfied, with $1/2 \leq a < 1$. A goal rule is a -satisfied if its hierarchical plan is a -satisfied, regardless of its conditions.

An action that is not part of a goal tree is said to be unexplained by the goal graph. A goal rule R is partially realized by an interaction if its precondition and/or hierarchical plan and/or postcondition are/is "almost satisfied".

Semantics

(see Goal, Hierarchical plan).

Sequence

An ordered sequence (ordered set) $T = (t_1, t_2, \dots, t_k)$ is denoted $T = (t_1, t_2, \dots, t_k)$, where

- (1) $\text{length}(T) = k$
- (2) $T(i)$ is the i^{th} element of T ($1 \leq i \leq k$)
- (3) $T(i..j)$ is the ordered sequence (ordered subset) $(t_i, t_{i+1}, \dots, t_j)$ ($1 \leq i \leq j \leq k$)
- (4) The *distance* between element t_i and element t_j is $\text{abs}(i-j)$,
where $\text{abs}()$ is the absolute value function
- (5) The empty ordered sequence is denoted $()$.

The ordered sequence of states corresponding to the interaction $I = (a_1, a_2, \dots, a_k)$ is $(S_0, S_1, S_2, \dots, S_k)$, where

- (1) S_0 is the state in which a_1 is executed. It is the state corresponding to the initial conditions; it may be empty.
- (2) S_i is the state resulting from the execution of a_i in state S_{i-1} ($1 \leq i \leq k$)
- (3) A special action, denoted a_0 , produces the state S_0' (note the prime) when executed in the empty state S^e . This action is not explicitly executed by the agent and is not shown in the enumeration of actions in I , although it is part of it.

The ordering among actions and states is specified as follows ($0 \leq i, j \leq k$):

- (1) $a_i < a_j \iff i < j$ (similarly for $\leq$, $>$ and $\geq$)
- (2) $S_i < S_j \iff i < j$ (similarly for $\leq$, $>$ and $\geq$)
- (3) $a_i < S_j \iff i \leq j$ Note: states are in between actions
- (4) $S_j < a_j \iff i < j$
- (5) If T contains either commands or states (but not both simultaneously), then
 $\text{first}(T) = m$ such that $m \leq s$ for all $s \in T$
 $\text{last}(T) = m$ such that $s \leq m$ for all $s \in T$.

State

A state describes a world situation. It is represented by a set of fully instantiated literals. The empty state is denoted S^e , i.e., $S^e = \{ \}$.

Task

A task is either a goal or a command.

Terminate(H)

$\text{Terminate}(H)$ is the set of models in which the hierarchical plan H is executable.

Appendix B

Domain theory for actions of the Office Tasks domain

```

action(say(T,M,R), [has(T,info,M), in_contact(T,R,verbal), tellable(M)],
  [], [has(R,info,M)],
  [has(T,info,M), in_contact(T,R,verbal), tellable(M), has(R,info,M)]).

action(ask(T,M,R), [has(R,info,M), in_contact(T,R,verbal), tellable(M)],
  [], [has(T,info,M)],
  [has(R,info,M), in_contact(T,R,verbal), tellable(M), has(T,info,M)]).

action(fax(T,M,R), [has(T,info,M), faxable(M), in_contact(T,R,fax)],
  [], [has(R,info,M)],
  [has(T,info,M), faxable(M), in_contact(T,R,fax), has(R,info,M)]).

action(look_up_phone_book(T,R,Phone_No), [phone_book(T,R/Phone_No)],
  [], [has(T,info,R/phone_no/Phone_No)],
  [has(T,info,R/phone_no/Phone_No), phone_book(T,R/Phone_No)]).

action(look_up_fax_book(T,R,Fax_No), [fax_book(T,R/Fax_No)],
  [], [has(T,info,R/fax_no/Fax_No)],
  [has(T,info,R/fax_no/Fax_No), fax_book(T,R/Fax_No)]).

action(redial_phone(T,No,R), [phone_free, not in_contact(T,R,verbal),
  phone_mem(T,No,R)],
  [phone_free],
  [in_contact(T,R,verbal)],
  [not phone_free, in_contact(T,R,verbal), phone_mem(T,No,R)]).

action(call_phone(T,No,R), [phone_free, has(T,info,R/phone_no/No)],
  [phone_free, phone_mem(T,_,_)],
  [in_contact(T,R,verbal), phone_mem(T,No,R)],
  [in_contact(T,R,verbal), has(T,info,R/phone_no/No),
  phone_mem(T,No,R), not phone_free]).

action(redial_fax(T,No,R), [fax_free, not in_contact(T,R,fax), fax_mem(T,No,R)],
  [fax_free],
  [in_contact(T,R,fax)],
  [in_contact(T,R,fax), fax_mem(T,No,R), not fax_free]).

action(call_fax(T,No,R), [fax_free, has(T,info,R/fax_no/No)],
  [fax_free, fax_mem(T,_,_)],
  [in_contact(T,R,fax), fax_mem(T,No,R)],
  [in_contact(T,R,fax), has(T,info,R/fax_no/No),
  fax_mem(T,No,R), not fax_free]).

action(hang_off_phone(T,R), [(not phone_free), in_contact(T,R,verbal)],
  [in_contact(T,R,verbal)],
  [phone_free],
  [phone_free, not in_contact(T,R,verbal)]).

action(hang_off_fax(T,R), [in_contact(T,R,fax), not fax_free],
  [in_contact(T,R,fax)],
  [fax_free],
  [not in_contact(T,R,fax), fax_free]).

```

Appendix C

Domain theory for goals and plans of the Office Tasks domain

```
a_goal( transmit_info(T,M,R),
        [has(T,info,M)],
        [has(T,info,M),has(R,info,M)]
      ).
a_goal( get_phone_no(T,R,No),
        [],
        [has(T,info,R/phone_no/No)]
      ).
a_goal( get_fax_no(T,R,No),
        [],
        [has(T,info,R/fax_no/No)]
      ).
a_goal( transmit_mes_phone(T,M,R,No),
        [has(T,info,M), tellable(M)],
        [has(T,info,M), tellable(M), has(R,info,M)]
      ).
a_goal( transmit_mes_fax(T,M,R,No),
        [fax_free, has(T,info,M), faxable(M)],
        [fax_free, has(T,info,M), faxable(M), has(R,info,M)]
      ).
a_goal( get_ph_connection(T,No,R),
        [phone_free, not in_contact(T,R,verbal)],
        [not phone_free, in_contact(T,R,verbal)]
      ).
a_goal( get_fax_connection(T,No,R),
        [fax_free, not in_contact(T,R,fax)],
        [not fax_free, in_contact(T,R,fax)]
      ).
```

```

/* The transmitter T transmits the message M to the receiver R */

gr( trans_info1, (transmit_info(T,M,R) :- [
    [has(T,info,M),tellable(M)],
    [has(T,info,M),tellable(M),has(R,info,M)],
    get_phone_no(T,R,No),
    transmit_mes_phone(T,M,R,No)]))).

gr( trans_info2, (transmit_info(T,M,R) :- [
    [has(T,info,M),faxable(M)],
    [has(T,info,M),faxable(M),has(R,info,M)],
    get_fax_no(T,R,No),
    transmit_mes_fax(T,M,R,No)]))).

/* T looks in the phone book to find the phone number No of R */

gr( get_ph_no1, (get_phone_no(T,R,No) :- [
    [phone_book(T,R/No)],
    [phone_book(T,R/No), has(T,info,R/phone_no/No)],
    com(look_up_phone_book(T,R,No)) ])) ).

/* T gets the phone/fax connection */

gr( get_ph_con1, (get_ph_connection(T,No,R) :- [
    [phone_free, not in_contact(T,R,verbal)],
    [not phone_free, has(T,info,R/phone_no/No),
in_contact(T,R,verbal)],
    com(call_phone(T,No,R)) ])) ).

gr( get_fax_con1, (get_fax_connection(T,No,R) :- [
    [fax_free, not in_contact(T,R,fax)],
    [not fax_free, in_contact(T,R,fax)],
    com(call_fax(T,No,R)) ])) ).

/* T looks in the fax book to find the fax number No of R */

gr( get_fax_no1, (get_fax_no(T,R,No) :- [
    [fax_book(T,R/No)],
    [fax_book(T,R/No), has(T,info,R/fax_no/No)],
    com(look_up_fax_book(T,R,No)) ])) ).

gr( trans_mes_ph, (transmit_mes_phone(T,M,R,No) :- [
    [phone_free, has(T,info,M), tellable(M)],
    [phone_free, has(T,info,M), tellable(M), has(R,info,M)],
    get_ph_connection(T,No,R),
    com(say(T,M,R)),
    com(hang_off_phone(T,R)) ]))).

/* T transmits the message M to R by fax */

gr( trans_mes_fax, (transmit_mes_fax(T,M,R,No) :- [
    [has(T,info,M), faxable(M)],
    [has(T,info,M), faxable(M), has(R,info,M)],
    get_fax_connection(T,No,R),
    com(fax(T,M,R)),
    com(hang_off_fax(T,R)) ]))).

```

Appendix D

Domain theory for implemented actions of the Modeller domain

```
action(addE(En), [not entity(En)],  
      [],  
      [entity(En)],  
      [entity(En)]).  
  
action(delE(E), [entity(E)],  
      [entity(E), relation(E,_,_)],  
      [],  
      [not entity(E), not relation(E,_,_)]).  
  
action(addR(E1,E2,R), [entity(E1), entity(E2), not related(E1,E2)],  
      [],  
      [relation(E1,E2,R)],  
      [entity(E1), entity(E2), relation(E1,E2,R)]).  
  
action(delR(E1,E2), [entity(E1), entity(E2), relation(E1,E2,R)],  
      [relation(E1,E2,R)],  
      [],  
      [entity(E1), entity(E2), not related(E1,E2)]).  
  
action(chgRT(E1,E2,R), [entity(E1), entity(E2), relation(E1,E2,_)],  
      [relation(E1,E2,_)],  
      [relation(E1,E2,R)],  
      [entity(E1), entity(E2), relation(E1,E2,R)]).
```


Appendix E

Domain theory for implemented goals and plans of the Modeller domain

```

a_goal( gen_entity(E1,E2,Gen),
        [entity(E1), entity(E2), not entity(Gen), relation(E1,E2,R)],
        [entity(E1), entity(E2), entity(Gen), not related(E1,E2),
         relation(Gen,E2,R), depend_on(E1,Gen)]) ).

a_goal( make_dependent_on(E1,E2),
        [entity(E1), entity(E2), not depend_on(E1,E2)],
        [entity(E1), entity(E2), depend_on(E1,E2)]) ).

/* Create a more general entity. */

gr( gen_entity, (gen_entity(E1,E2,Gen) :- [
    [entity(E1), entity(E2), not entity(Gen), relation(E1,E2,R)],
    [entity(E1), entity(E2), entity(Gen), not related(E1,E2),
     relation(Gen,E2,R), depend_on(E1,Gen)],
    com(addE(Gen)),
    make_dependent_on(E1,Gen),
    com(addR(Gen,E2,R)),
    com(delR(E1,E2))]) ) ).

/* Create a dependance between two entities. */

gr( role_dep1, (make_dependent_on(E1,E2) :- [
    [entity(E1), entity(E2)],
    [entity(E1), entity(E2), relation(E1,E2,role)],
    com(chgRT(E1,E2,role))]) ) ).

gr( role_dep2, (make_dependent_on(E1,E2) :- [
    [entity(E1), entity(E2), not related(E1,E2)],
    [entity(E1), entity(E2), relation(E1,E2,role)],
    com(addR(E1,E2,role))]) ) ).

```

Appendix F

Example of a session with LEADER

```
| ?- in_dom(mod).
% compiling file /a/kaml1/usr4 ml/bertrand/c/mod_actions.p
* Clauses for action/5 are not together in the source file
% mod_actions.p compiled in module user, 1.067 sec 0 bytes
Processing goal rules...
    gen_entity(_7494,_7511,_7528)
    make_dependent_on(_7494,_7511)
    gen_entity
    role_dep1
    role_dep2

yes
| ?-
| ?- in_ses(mm3).
% compiling file /a/kaml1/usr4 ml/bertrand/c/mod_ses
* Clauses for initial_user_sequence/2 are not together in the source file
* Clauses for user_sequence/2 are not together in the source file
% mod_ses compiled in module user, 0.750 sec 12 bytes

yes
| ?- analyze.
Computing initial state...
Computing states...
    10
Computing cycles...

*** New Cycle ***

Testing gen_entity ...
Testing role_dep1 ...
Testing role_dep2 ...      g11

*** New Cycle ***

Testing gen_entity ...
Testing role_dep1 ...
Testing role_dep2 ...
yes
| ?-
| ?- list_all_trees.
Computing nodes used...
    make_dependent_on(department,university)    g11, role_dep1
    [1,10,10]    [2,4]    [2,4]    [10]
        [entity(department)]
        [entity(university)]
        [not_related(department,university)]
        addR(department,university,role)    g11

    addE(dept_employee)    (c6)
    addE(university)    (c4)
    addR(teacher,department,related)    (c3)
    addE(department)    (c2)
    addE(teacher)    (c1)

yes
```

```

| ?- list_states.
State 0

State 1    addE(teacher)
           entity(teacher)/[1]

State 2    addE(department)
           entity(department)/[2]    entity(teacher)/[1]

State 3    addR(teacher,department,related)
           relation(teacher,department,related)/[3]    entity(department)/[2]
           entity(teacher)/[1]

State 4    addE(university)
           entity(university)/[4]    relation(teacher,department,related)/[3]
           entity(department)/[2]    entity(teacher)/[1]

State 5    addE(student)
           entity(student)/[5]    entity(university)/[4]
           relation(teacher,department,related)/[3]    entity(department)/[2]
           entity(teacher)/[1]

State 6    addE(dept_employee)
           entity(dept_employee)/[6]    entity(student)/[5]
           entity(university)/[4]    relation(teacher,department,related)/[3]
           entity(department)/[2]    entity(teacher)/[1]

State 7    addR(teacher,dept_employee,char)
           relation(teacher,dept_employee,char)/[7]    entity(dept_employee)/[6]
           entity(student)/[5]    entity(university)/[4]
           relation(teacher,department,related)/[3]    entity(department)/[2]
           entity(teacher)/[1]

State 8    addR(dept_employee,department,related)
           relation(dept_employee,department,related)/[8]    relation(teacher,dept_employee,char)/[7]
           entity(dept_employee)/[6]    entity(student)/[5]
           entity(university)/[4]    relation(teacher,department,related)/[3]
           entity(department)/[2]    entity(teacher)/[1]

State 9    delR(teacher,department)
           relation(dept_employee,department,related)/[8]    relation(teacher,dept_employee,char)/[7]
           entity(dept_employee)/[6]    entity(student)/[5]
           entity(university)/[4]    entity(department)/[2]
           entity(teacher)/[1]

State 10   addR(department,university,role)
           relation(department,university,role)/[10]    relation(dept_employee,department,related)/[8]
           relation(teacher,dept_employee,char)/[7]    entity(dept_employee)/[6]
           entity(student)/[5]    entity(university)/[4]
           entity(department)/[2]    entity(teacher)/[1]

<end_of_states>

yes
| ?- commands_used.
addE(teacher)    (c1)
addE(department)    (c2)
addR(teacher,department,related)    (c3)
addE(university)    (c4)
addE(dept_employee)    (c6)
addR(department,university,role)    (c10)

yes

```

```
| ?- commands_not_used.
addE(student)      (c5)
addR(teacher,dept_employee,char)  (c7)
addR(dept_employee,department,related)  (c8)
delR(teacher,department)  (c9)
```

```
yes
| ?- indirect_commands.
addE(teacher)      (c1)
addE(department)   (c2)
addR(teacher,department,related)  (c3)
addE(university)   (c4)
addE(dept_employee) (c6)
```

```
yes
| ?- direct_commands.
addR(department,university,role)  (c10)
```

```
yes
| ?- learning_method(heuristic2).
```

```
yes
| ?- learn.
*****
Testing gen_entity ...
```

```
Initial goal rule.
gen_entity(En1,En2,En3)
  Preconds:
    [entity(En1)]
    [entity(En2)]
    [not entity(En3)]
    [relation(En1,En2,Type1)]
  Postconds:
    [entity(En1)]
    [entity(En2)]
    [entity(En3)]
    [not related(En1,En2)]
    [relation(En3,En2,Type1)]
    [depend_on(En1,En3)]
  Strategy:
    com(addE(En3))
    make_dependent_on(En1,En3)
    com(addR(En3,En2,Type1))
    com(delR(En1,En2))
```

```
Partially satisfied goal rule:
gen_entity(teacher,department,dept_employee)
  Preconds:
    [entity(teacher)]
    [entity(department)]
    [not entity(dept_employee)]
    [relation(teacher,department,related)]
  Postconds:
    [entity(teacher)]
    [entity(department)]
    [entity(dept_employee)]
    [not related(teacher,department)]
    [relation(dept_employee,department,related)]
    [depend_on(teacher,dept_employee)]
  Strategy:
    com(addE(dept_employee))
```

```

make_dependent_on(teacher,dept_employee)
com(addR(dept_employee,department,related))
com(delR(teacher,department))

First state      : 5
Last state      : 9
Substitution     : [Type1/related,En3/dept_employee,En2/department,En1,teacher]
New First,Last States : 5,9

Satisfied subgoals : [c+5,c+3,c+9]
Satis. subgoal index : [1,3,4]
Unsat. subgoal index : [2]
Unexplained nodes : [7]
Inverse substitution : gen_entity(En1,En2,En3)
Inverse subst. PrePos : [entity(En1),entity(En2),not
entity(En3),relation(En1,En2,Type1)], [entity(En1),entity(En2),entity(En3),not
related(En1,En2),relation(En3,En2,Type1),depend_on(En1,En3)]
Computa. of conditions: [success,success]

New goal rule:
gen_entity(En1,En2,En3)
Preconds:
[not related(En3,En2)]
[entity(En2)]
[not entity(En3)]
[entity(En1)]
[not related(En1,En3)]
[relation(En1,En2,Type1)]
Postconds:
[relation(En3,En2,Type1)]
[entity(En3)]
[relation(En1,En3,char)]
[entity(En1)]
[entity(En2)]
[not related(En1,En2)]
Strategy:
com(addE(En3))
com(addR(En1,En3,char))
com(addR(En3,En2,Type1))
com(delR(En1,En2))
***** The new rule is asserted.

*****
Testing role_dep1 ...

*****
Testing role_dep2 ...

yes
| ?- analyze.
Computing initial state...
Computing states...
10
Computing cycles...

*** New Cycle ***

Testing new_gr+1 ...      gl2
Testing gen_entity ...
Testing role_dep1 ...
Testing role_dep2 ...

*** New Cycle ***

```

```

Testing new_gr+1 ...
Testing gen_entity ...
Testing role_dep1 ...
Testing role_dep2 ...
yes
| ?- list_all_trees.
Computing nodes used...
make_dependent_on(department,university) (g11, role_dep2)
[1,10,10] [2,4] [2,4] [10]
[entity(department)]
[entity(university)]
[not related(department,university)]
addR(department,university,role) (c10)

gen_entity(teacher,department,dept_employee) (g12, new_gr+1)
[1,6,9] [1-3] [1-3,6] [6-9]
[not related(dept_employee,department)]
[entity(department)]
[not entity(dept_employee)]
[entity(teacher)]
[not related(teacher,dept_employee)]
[relation(teacher,department,related)]
addE(dept_employee) (c6)
addR(teacher,dept_employee,char) (c7)
addR(dept_employee,department,related) (c8)
delR(teacher,department) (c9)

addE(university) (c4)
addR(teacher,department,related) (c3)
addE(department) (c2)
addE(teacher) (c1)

yes
| ?- commands_not_used.
addE(student) (c5)

yes
| ?- commands_used.
addE(teacher) (c1)
addE(department) (c2)
addR(teacher,department,related) (c3)
addE(university) (c4)
addE(dept_employee) (c6)
addR(teacher,dept_employee,char) (c7)
addR(dept_employee,department,related) (c8)
delR(teacher,department) (c9)
addR(department,university,role) (c10)

yes
| ?- indirect_commands.
addE(teacher) (c1)
addE(department) (c2)
addR(teacher,department,related) (c3)
addE(university) (c4)
addE(dept_employee) (c6)

yes
| ?- direct_commands.
addE(dept_employee) (c6)
addR(teacher,dept_employee,char) (c7)
addR(dept_employee,department,related) (c8)
delR(teacher,department) (c9)
addR(department,university,role) (c10)

yes

```

END

0 8-1 1-94

FIN