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.

Building Software Specifications using Explanation-based Learning with Incomplete Theories

by

Jean Genest

Thesis submitted to
the School of Graduate Studies and Research
in partial fulfillment of the requirements for
the Master degree in Computer Science
offered by the Joint Program in Computer Science

University of Ottawa,
Ottawa, Ontario, Canada
1990

ISBN 0-315-60113-2

UNIVERSITÉ D'OTTAWA
UNIVERSITY OF OTTAWA

I hereby declare that I am the sole author of this thesis. I authorize the University of Ottawa to lend this thesis to other institutions or individuals for the purpose of scholarly research.

Jean Genest

I further authorize the University of Ottawa to reproduce this thesis by photocopying or by other means, in total or in part, at the request of other institutions or individuals for the purpose of scholarly research.

Jean Genest

Acknowledgement

I would like to thank my supervisor Stan Matwin for his guidance in the course of this research. The comments and suggestions of Robert C. Holte, Boris Plante and Daniel Charlebois were helpful in the preparation of this thesis and of other publications derived from it. The environment provided by the Ottawa Machine Learning Group allowed me to meet other researchers and provided an excellent platform to exchange ideas.

I would like to thank my wife Lise for reviewing my thesis for mistakes. Most importantly, I would like to thank her for the support and understanding she gave me throughout this project. As I was working long hours on this research, Lise carried and gave birth to our first born Jean-Philippe. *Mille Mercis Lise! Bienvenue Jean-Philippe!*

Contents

List of Figures

Introduction

This thesis presents several contributions in Machine Learning (ML) and in the application of Artificial Intelligence (AI) to software engineering.

A promising approach is presented to deal with the impeding problem of incomplete theories facing the explanation-based learning (EBL) community. This approach combines elements of abduction, analogical reasoning and case-based reasoning. Our approach not only provides an explanation when the domain theory usually fails, it also repairs the incomplete theory by adding the missing part into it.

Our approach was implemented in a system named LISE (Learning In Software Engineering). LISE is a system which translates informal and non-operational user requirements into formal and operational specifications. LISE keeps the individual specification of each operation in a global specification. The global specification is the specification of the software system being developed.

When LISE is presented with a new user requirement, it first verifies if the specification already includes it. If it does, LISE will translate the user requirement into an operational specification. To do so, LISE builds an explanation of the user requirement. When the new user requirement is not covered by the specification, the problem becomes similar to the problem of incomplete theories. LISE will include the new user requirement in the specification using our approach to deal with an incomplete theory.

As a brief example, consider that LISE contains the specification of a banking system. The banking system only contains two operations: `withdraw` and `debit` (fig I-1). The specification for `withdraw` is given in terms of a precondition and a procedure. The precondition requires that the person withdrawing owns an account. The procedure is to debit the amount from the account and to issue the money to the person. The `debit` operation is defined similarly. The specification of the banking system also contains goals for some of the operations. For example, the goal of the operation `subtract_from_balance` is `record_transaction`.

```
withdraw(Person,Amount)              debit(Account,Amount)
isa: TRANSACTION                     isa: ACTION
  precondition:                        precondition:
    account(Person,Account)              balance(Account,Balance)
    goal(account,identify_client)        goal(balance,protect_bank_interest)
  procedure:                             Balance > Amount
    debit(Account,Amount)                procedure:
    issue_money(Person,Amount)           subtract_from_balance(Account,Amount)
                                         goal(subtract_from_balance,record_transaction)
```

Fig I-1 A domain theory containing the specification for withdraw

Consider now that the training example of figure I-2 is presented. The training example represents an activity in which a person named bob borrows one thousand dollars. The features of borrow is that bob owns the account account_1, bob has a credit margin of 3500 dollars, a loan of 1000 dollars is recorded for bob and bob receives 1000 dollars.

```
The training example is: borrow(bob,1000).

The facts are:
  account(bob,account_1)
  credit_margin(bob,3500)
  record_loan(bob,1000)
  issue_money(bob,1000)
```

Fig. I-2 The training example for borrow(Person,Amount)

To verify if the specification of figure I-1 contains the operation borrow, EBL is used. EBL will not find an explanation for borrow because the specification of the banking system does not include that operation.

If we apply our approach to EBL, we will not only be able to explain the example of borrow, but we will also be able to augment the specification from the one shown on figure I-1 to the one shown on figure I-3. In this example, analogical reasoning allowed us to construct the specification of borrow and grant_loan using what we already know about withdraw and debit.

```
withdraw(Person,Amount)              debit(Account,Amount)
isa: TRANSACTION                     isa: ACTION
  precondition:                        precondition:
    account(Person,Account)              balance(Account,Balance)
    goal(account,identify_client)        goal(balance,protect_bank_interest)
  procedure:                             Balance > Amount
    debit(Account,Amount)                procedure:
    issue_money(Person,Amount)             subtract_from_balance(Account,Amount)
                                           goal(subtract_from_balance,record_transaction)

borrow(Person,Amount)                grant_loan(Person,Amount)
 isa: TRANSACTION                    isa: ACTION
   precondition:                        precondition:
     account(Person,Account)              credit_margin(Person,Credit_margin)
   procedure:                             Credit_margin > Amount
     grant_loan(Person,Amount)          procedure:
     issue_money(Person,Amount)           record_loan(Person,Amount)
```

Fig I-3 The domain theory produced by LISE

In addition to the innovative technique of applying ML to building system specifications, other contributions of this research are the use of abduction in EBL, a heuristic to select the best partial explanation, a method for combining multiple partial explanations, and the use of case-based reasoning as a back-up for an incomplete domain theory.

The following section presents the content of the thesis.

The software engineering issue of building the specification for a software system is discussed in the first chapter. This chapter also presents the origin and the motivation for the representation we developed to express a specification in a domain theory for EBL use. The usage of ML techniques to assist in the analysis phase of the software design life cycle is unique. Chapter 2 presents ML methods and justifies the choice of EBL for the task of building a system specification. Chapter 3 brings a deeper look at EBL and the incomplete theory problem. The novel approach we propose to deal with the problem of incomplete theories is also introduced there. The system LISE is presented in chapter 4. Two examples using a banking system and a fleet management system are presented. An analysis of the complexity of LISE (and our approach) is given in chapter 5. In chapter 6, we compare our results with related work. This is followed by a discussion of future research in chapter 7 and a conclusion.

The results of the research reported in this thesis were the object of two papers co-authored by the author of this thesis. The first paper, [Genest et al. 1990a], was

presented at the Canadian AI Conference CSCSI-90 held in Ottawa. The second paper, [Genest et al. 1990b], was presented at the 7th Annual Conference on Machine Learning held in Austin, Texas. The reviewers of both conferences offered very useful comments. Some of the suggestions are addressed in this thesis.

Chapter 1

System Specifications

In the software development life cycle, one of the early phases consists in transforming informal and non-operational user requirements into a more formal and operational software specification. This phase is called the analysis phase. Many factors, including the analyst's familiarity with the application area, his expertise and his creativity, have an impact on the specification that the analyst will elaborate for a user requirement [Shlear et al. 1988]. These factors explain why different specifications can be produced for the same requirement and, even worse, why the resulting specification can suffer from incompleteness and inconsistency.

A specification is incomplete if it contains undefined operations. We can ensure that a specification is complete by checking whether each operation is defined using primitive operations or using other operations which are defined using primitive operations.

A specification is inconsistent if there exists multiple and ambiguous ways to define an operation. The job of the analyst is to ensure that each operation has a unique and unambiguous definition within the scope of a given system..

The specification of a system is two-fold. It contains a description of the static component and the dynamic component of the system. The static component of the system, or the *data model*[1], represents the objects of the system. It is conventionally depicted using a diagram called Entity-Relationship Diagram (ERD) [Chen 1983]. The dynamic component of the system, or the *process model*, represents the operations. It is conventionally depicted using Data Flow Diagrams [Gane et al. 1979]. Recent literature on conceptual data modelling proposed a unified approach to document both the static part and the dynamic part of a system [Brodie et al. 1984]. Under this approach, an unified frame-

[1] Terms in italic are introduced in the text.

based representation is used to describe the objects and the operations. This approach is inspired from the knowledge representation field of artificial intelligence. The frame-based representation is preferred in our system because it allows us to manipulate the specification in a much more coherent manner. The next sections will present how the analyst goes about modelling a system using the ERD and the Data Flow Diagram. It will also show how the information of these two diagrams can be represented in a frame-based notation.

1.1 The data model

The analyst typically builds the data model first. The ERD is usually used to build the data model because it provides a good graphical representation of the objects and their relationships. The ERD is very simple to understand. The analyst can usually present the ERD directly to a user to obtain the approval of the design.

Each object of the system is mapped to an entity and a box is drawn for it. The attributes of the entity may or may not be annotated on the diagram. Diamonds are used on the diagram to depict relationships that may exist between some entities. Special relationships can also exist. For example, the *role* relationship is used to identify an entity as a specialization of another one.

The ERD is supplemented by a Data Dictionary [Gane and al. 1979]. The data dictionary contains an entry for each entity and relationship described on the diagram. The entry contains a description of the attributes of the entity or the relationship.

Fig 1.1 illustrates an ERD representing the entities `person`, `client` and `account`. The relationship `owns` connects the entity `account` to `person`. The special relationship `is-a` connects the entity `client` to `person`.

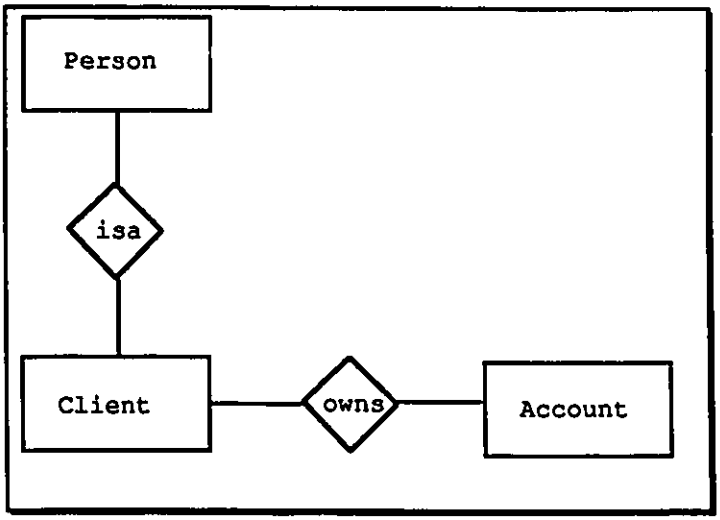

Fig 1.1 A data model using the Entity-Relationship model

Figure 1.2 shows the data model of figure 1.1 in a frame-based notation. Each entity is defined by a unique frame. A special slot called `isa` is used to identify the parent frame. It corresponds to the `is-a` relationship on the diagram. The other slots in the frame contain the attributes and the relationships of the entity. For example, the slot `phone-number` of the frame `person` indicates that a property of a `person` is to have a `phone-number`. On the other hand, the slot `account` of the frame `client` represents the relationship that exists between each `client` and and his/her `account`. The `account` itself is defined by its own frame named `account`.

```
person                  client                  account
   isa entity              isa person              isa entity
   name                    branch                  account_number
   address                 account                 balance
   phone-number
```

Fig 1.2 The data model of fig 1.1 represented using a frame-based notation

1.2 The process model

A software system implements one or more activity normally performed in an organization. The approach used by system analysts to produce the dynamic component of the specification of a software system consists in specifying externally observable

activities [Yau et al. 1986]. The specification of the activities are integrated together resulting in the specification of an entire software system.

To build the specification of an activity, the analyst takes an example of an externally observable activity expressed in some notation and extracts from it a generalized sequence of actions. Figure 1.3 shows an English description of an activity called `withdraw` as observed by an analyst.

```
Bob walks up to the teller and indicates that
he wishes to withdraw $100. The teller
verifies that Bob has an account. The teller
then ensures that the balance of Bob's account
- $150 - is greater that $100. The teller then
subtracts $100 from the balance of Bob's
account and gives the money to Bob. Bob walks
away.
```

Fig 1.3 Scenario of an activity as seen by the analyst

The analyst will identify all the relevant actions and the order in which they appear. He will then discard all the actions considered irrelevant according to his understanding of the domain. In the scenario above, the analyst discards `Bob walks up to the teller` and `Bob walks away`. The analyst will also generalize the actions so that they do not only apply to constants introduced in by the specific example.

To document the activity of figure 1.3, the analyst may use a Data Flow Diagram (DFD). A DFD is a graphical technique to describe processes where rectangular boxes are used to describe the processes and arcs are used to describe the flow of data between the processes. Figure 1.4 shows a DFD for `withdraw`. Each relevant action of `withdraw` is represented by a rectangular box. The square-shaped box containing `client` represents an external entity. The arcs between the rectangular boxes indicate the flow of data between the actions. The arcs between the square and the rectangular boxes show that the `client` must send the data `amount` to the process `check_account_ownership` and `client` receives the data `money` from `issue_money`. It is not necessary to show the teller

on the DFD of figure 1.4 because the teller is not a process, nor does the teller interact with that part of the system as an external entity.

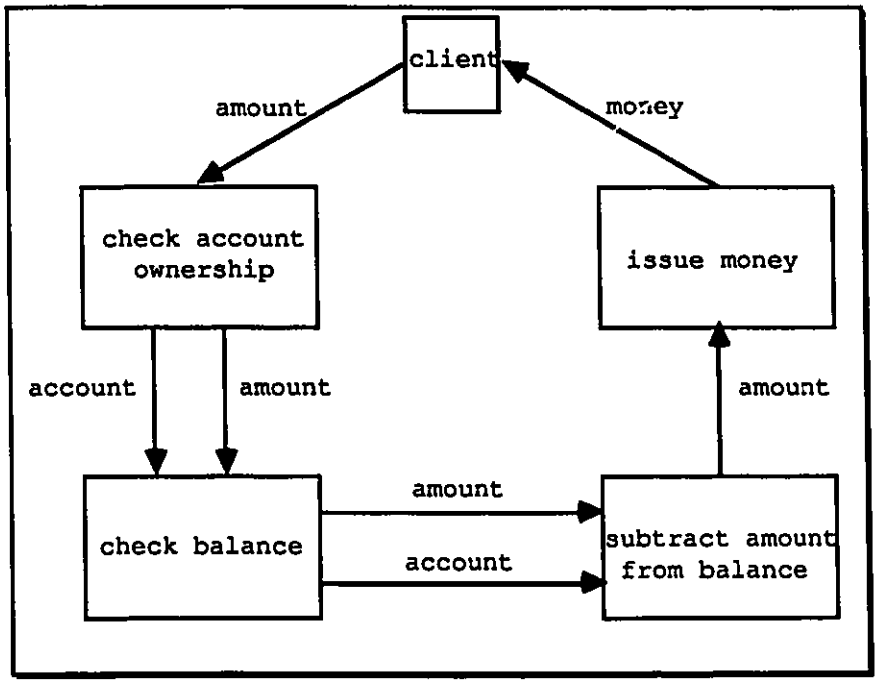

Fig 1.4 A Data Flow Diagram representing the scenario

The analyst could also describe the activity `withdraw` using a frame-based notation. A method for doing so is proposed in figure 1.5. The frame used to describe the activity `withdraw` contains three slots. The first slot is `isa`. The `isa` slot is used to indicate that the object being described is a TRANSACTION (for now, TRANSACTION can be considered as a synonym for activity). The two other slots contain a precondition and a procedure. The precondition outlines which actions must be executed first. It also requires that these actions succeed. The procedure contains a sequence of actions which may be considered as the body of the transaction.

The frame-based notation proposed to represent the data model and the process model will be preferred in our application for three reasons. First, the frame-based representation allows both the entities and the processes to be described in a uniform domain theory. Second, the frame-based representation allows the objects and the processes to be organized in a hierarchy using the `isa` relationship. Third, the frame-based notation used to describe the processes contains more information than the DFD. For example, in the DFD

```
withdraw(Person,Amount)
   isa TRANSACTION
   precondition:
      account(Person,Account)
      balance(Account,Balance)
      Balance > Amount
   procedure
      subtract_from_balance(Account,Amount)
      issue_money(Client,Amount)
```

Fig 1.5 A frame-based representation for the scenario

of `withdraw`, there is no distinction between the precondition and the procedure, as in the frame-based notation.

Chapter 2

Machine Learning

The objective of our research is to develop a method to synthesize the specifications of the operations of a system from examples of those operations, a task normally accomplished by an analyst. We believe that in his duty, the analyst undergoes a learning experience. As a result, we regard Machine Learning (ML) concepts as a foundation for our method.

The initial step was to find the most appropriate method of learning. The following step was to adapt the ML method selected to the task of building the specification. As our research progressed, the ideal method of learning came out to be a combination of Explanation-Based Learning, Abduction, Analogical Reasoning and Case-based reasoning. The overall result is not only a system that builds specifications but also a learning method applicable to more general planning tasks.

A classification of ML methods was first proposed during the 1985 International Workshop in Machine Learning [Kodratoff 1988]. The classification divides the methods into two groups: Explanation-based Learning (EBL) [DeJong et al. 1986],[Mitchell et al. 1986] and Similarity-based Learning (SBL) [Michalski et al. 1983]. These methods are also called Analytical and Empirical Learning methods, respectively.

2.1 Explanation-based Learning

Explanation-Based Learning (EBL) is an analytical learning method where a concept definition is learned from a single training example [DeJong et al. 1986],[Mitchell et al. 1986],[Ellman 1989]. The process of EBL uses a *domain theory*, i.e. a set of rules pertinent to the domain of the goal concept. The domain theory is used to construct an explanation, or a deductive proof, of how a training example is an instance of the *goal concept*. The goal concept is often expressed in terms which are not useful for the particular

expression of the concept definition. The explanation creates a new definition of the goal concept, expressed in terms which are operational, i.e. adequate for a given use of the concept. A concept definition is considered operational when it is expressed in terms which satisfy a stated *operationality criterion*. Figure 2.1 outlines the explanation-based learning problem.

```
Given:
   *  Goal Concept
      A definition of the concept to be          learned.
   *  Training Example
      An example of the goal concept.
   *  Domain Theory
      A set of rules and facts to be used in explaining how
      the training example is an example of the goal
      concept.
   *  Operationality Criterion
      Criterion specifying which features must be used to
      express the goal concept.

Determine:
      A generalization of the training example that is a
      sufficient concept definition for the goal concept and
      that satisfies the operationality criteria.
```

Figure 2.1 Explanation-Based Learning Problem

An assumption in EBL is that the domain theory be complete, consistent, correct and tractable. A complete domain theory contains enough rules of the domain to prove any training example. A consistent domain theory will produce a unambigous explanation for a training example. A correct domain theory produces explanations that are valid according to a domain expert. A tractable domain theory produce explanation within allocated time and resources. It is seldom the case that a domain theory will satisfy these four criteria in a real application.

2.2 Similarity-based Learning

Similarity-based Learning (SBL) designates learning methods which employ induction to produce a generalized concept definition from a set of positive and negative training examples. In SBL, a concept definition is generalized using the similarities between positive training examples and is specialized using the dissimilarities between the negative and the positive training examples.

In SBL, only a minimal amount of background knowledge is required since the concept is generated solely from a set of training examples. The minimal background knowledge is used to define an adequate representation of the training examples (i.e. the attributes used, their range, etc).

Figure 2.2 presents the problem of similarity-based learning as presented in [Michalski et al. 1983].

Many training examples, both positive and negative, and preferably exempt of noise, are required in SBL. SBL also needs a *preference criterion* or *bias* to guide the generalization towards a useful concept definition.

2.3 Choice of the ML method for building system specifications

We required a domain-independent method to learn a system specification from user requirements. The following questions were used in the selection process:

a. Is there one or many positive training examples available for each instance of user requirements?

b. Is there any negative training examples available for each instance of user requirements?

c. Is there any background knowledge?

```
given:
  * observational statements (facts)
    specific knowledge about some objects,
    situations, processes and so on
    representing positive and negative training
    example.
  *a tentative inductive assertion which may
      be null
  * background knowledge
    defines assumptions and constraints imposed
    on observational statements and generated
    domain knowledge. The latter includes a
    preference criterion characterizing the
    desirable properties of the sought
    inductive assertions.

  determine:
    an inductive assertion, or hypothesis, that
  tautologically or weakly implies the
  observational statements, and satisfies the
  background knowledge.
```

Figure 2.2 Similarity-based Learning Problem

As discussed previously, one instance of a user requirement corresponds to one externally observable activity. For each user requirement (i.e. positive training example), our system has to learn one single specification (concept to learn). The one-to-one relationship between a user requirement and its specification favors EBL. The absence of negative training examples also favors EBL.

The context of our learning task requires that a certain amount of background knowledge describing the data model of the system and the specification of the operations be at hand. Even though both EBL and SBL require background knowledge, the background knowledge of EBL is, in our view, more suitable to contain a specification because of its format.

The result of EBL is an operationalization of the goal concept. This is an important feature since the training examples in our application correspond to non-operational user requirements that have to be translated into operational specifications.

EBL systems do not only provide a method to build new operational concept definitions. [Ellman 1989] mentions that the research in EBL also addresses the problems of justified generalizations, chunking, operationalization and justified analogy. In view of that classification, we believe that our use of EBL for the task of building systems specifications from user requirements will provide interesting results in operationalization.

There is however a major drawback to EBL to deal with: the incomplete theory problem. The next chapter will present EBL and the problem of incomplete theories and will present our approach to deal with it.

Chapter 3

Explanation-based Learning and the Incomplete Theory Problem

3.1 Description of EBL

Explanation-Based Learning (EBL) is an analytical learning method where a concept definition is learned from a single training example [DeJong et al. 1986],[Mitchell et al. 1986],[Ellman 1989]. The process of EBL uses a *domain theory*, i.e. a set of rules pertinent to the domain of the goal concept. The domain theory is used to construct an explanation, or a deductive proof, of how a training example is an instance of the *goal concept*. The goal concept is proven if there is a rule having the goal concept as a consequent and where the antecedents unify with training example facts or where the antecedents are consequents of other rules for which antecedents are proven.

The goal concept is often expressed in terms which are not useful for the particular expression of the concept definition. The explanation creates a new definition of the goal concept, expressed in terms which are operational, i.e. adequate for a given use of the concept. A concept definition is considered operational when it is expressed in terms which satisfy a stated *operationality criterion*. The process of translating a non-operational goal definition into an operational one is called operationalization.

3.2 Limitations of EBL

EBL systems will not function if the domain theory is imperfect. A imperfect domain theory suffers from one or many of the following problems [Ellman 1989]:

a. **Incompleteness**: rules are missing making the explanation of some training example impossible,

b. **Inconsistency**: ambiguous explanations are produced for a training example,

c. **Incorrectness**: a training example is incorrectly explained, and,

d. **Intractability**: an explanation can not be built within allocated time and space resources.

Real applications usually suffer from one or more of the above problems. As examples of the incompleteness problem, [Mitchell et al. 1986] mentions the problems of predicting the stock market or the weather. The problem is that there is no theories of economics or meteorology complete enough so that we could prove why, for example, a stock has doubled over a twelve months period or why it will rain on the week-end.

The inconsistency problem is when inconsistent statements can be derived from the same theory. An example, also found in [Mitchell et al. 1986], is taken from a small theory about tables and object (the safe-to-stack example). The theory contains two rules which can be used to derive the weight of a table: one rule using a density-volume calculation and one a default rule based on the material of the table. An inconsistency will exist when both rules are used to calculate the weight and both rules produce a different value.

The incorrectness problem appears when a domain theory produces results which do not agree with a domain expert. As an example of an incorrect theory, [Mooney et al. 1989] shows that a domain theory about cups containing only rules on the structural aspect of cups will incorrectly classify a shot glass as a cup. [Mooney et al. 1989] breaks the inconsistency by extending the concept definition of the domain theory with a conjunction obtained inductively using positive and negative examples of cups.

An example of the intractability problem, cited in [Mitchell et al. 1986], is found in the chess theories. The theory of chess, which contains rules about the moves of the pieces, their values, etc, is complete. One can learn all these rules rapidly and play a game of chess with anyone else. The intractability problem is that there is no way with the chess theory to explain, for instance, why the opening move 'pawn to king four' is a member of the goal concept 'moves that lead to a win or draw for white'.

Recent research in EBL focused on means to circumvent the imperfect theory problem. This thesis presents an approach which deals with the incomplete theory problem.

3.3 The incomplete domain theory problem

The imperfection that we have to face in our particular application is the incomplete domain theory problem. The domain theory is incomplete when rules are missing.

An explanation in EBL is built using rules. The antecedents of the rules are satisfied using facts from the training example or using the consequents of other rules. When the domain theory is incomplete, rules that would be required to complete a particular explanation might be missing. If it is the case, EBL will produce one or many partial explanations.

Definition 3.1: **A partial explanation** is an explanation containing proven and unproven antecedents. An unproven antecedent is an antecedent for which no fact was found in the training example and for which no rule could be used to prove the antecedent.

3.4 Our approach to the incomplete domain theory

An incomplete domain theory is recognized when one or many partial explanations are produced instead of a complete explanation. Partial explanations are built using rules which have antecedents in common with the training example facts. The rules employed by EBL to build the partial explanations will be used in our approach to build a plausible explanation of the training example.

There are three steps in our approach to deal with the incomplete domain theory: the abduction step, the analogical reasoning step and the case-based reasoning step.

In the first step, a partial explanation providing the best coverage of the training example is selected using a heuristic. The partial explanation is transformed into a plausible explanation using abduction. Abduction is a logical inference which is, on one

hand, more flexible than deduction, but on the other hand, not as well-founded as deduction.

The second step in our approach is applied when abduction can not complete any partial explanation. In this step, the best partial explanation for the training example is also selected using the same heuristic as in the first step. New rules are created using analogical reasoning applied between the unproven antecedents of the partial explanations and the training example facts. The new rules created in this step allow the training example to be explained. The new rules are also added to the domain theory in order to repair the incompleteness problem.

The third step to deal with the incomplete domain theory involves the usage of a case-based system. It is applied only if the previous two steps failed to produce a complete explanation. In this step, the case-based system will retrieve a case and adapt it to the training example. The case-based system can be seen as providing an extension to the incomplete theory. It is used to classify a training example for which no explanation is possible.

3.4.1 Selecting a partial explanation

The two first steps used to deal with the incomplete domain theory (i.e. abduction and the analogical reasoning) require that one or more partial explanations be selected from among several produced. A heuristic was developed which ranks the partial explanations generated for a specific training example according to a score. The heuristic developed to calculate the score:

a. rewards a partial explanation for each feature it shares with the training example,

b. penalizes a partial explanation for each of its unproven leaves,

c. penalizes a partial explanation for each feature of the training example that was unaccounted for, and,

d. penalize slightly a partial explanation for each abductive inference that was used in its construction.

After each partial explanation has received a score, the entire set of partial explanations is sorted into a list where the partial explanation with the highest score is first and the partial explanation with the lowest score is last. The first partial explanation in the list is sometimes referred to as the 'best' partial explanation.

3.4.2 Abduction as a technique to complete partial explanations

Abduction is the generation of hypotheses which, if true, would explain observed facts. More precisely, if the rule `Q <- P` and the fact `Q` are given, then the abductive conclusion would be `P`. `P` can be characterized as being a hypothesis because there could exist another rule `Q <- P'` which could have been used to derive `Q`.

Abduction may be used to complete or improve the best partial explanation selected by our heuristic. Considering the training example features as facts, we will attempt to draw hypotheses from the unproven antecedents using abduction. If hypotheses can be drawn for each unproven antecedent, the partial explanation will be completed. Using the above example of the rule `Q <- P` and `Q`, `P` would be an unproven antecedent in the partial explanation and `Q` would be a fact given as a training example feature. If `P` is the only unproven antecedent, a hypothesis can be drawn to account for the occurrence of `Q` in the training example, and the partial explanation can be completed.

Figure 3.1 shows an example of an explanation for the goal concept `GC`. There are two rules in the domain theory. The first one is `GC <- P,R,S` and the second one is `Q <- P`. The training example facts are `Q,R` and `S`. When trying to prove `GC`, a partial explanation is obtained because `P` is not provable (the dashed lines from `P` to `GC` show that `P` is not provable). However, the partial explanation of `GC` can be completed the following way. Abductions can be used to transform `P` into a hypothesis using the rule `Q <- P` and the fact `Q`. Once `P` is transformed into a hypothesis, we can *abductively* infer `GC`.

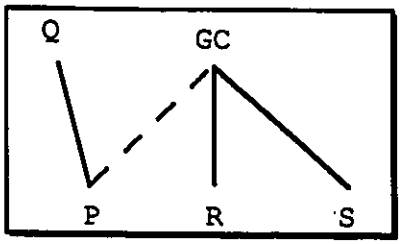

Figure 3.1 Abduction to complete a partial explanation

The process of abduction does not extend the domain theory. It only allows to produce a plausible explanation for a training example which would not have been explained by regular EBL.

The correctness of a plausible explanation is always guaranteed because abduction does not introduce new antecedents in the explanation nor new rules in the domain theory. It only turns the unproven antecedents into proven ones using a different logical inference. The explanation built using abduction is the same as the one which would have been obtained, should the training example contain the missing fact (P, above) instead of the consequent of that fact (Q, above).

3.4.3 Analogical reasoning as a technique to complete partial explanations

Analogical reasoning is the process of solving a problem using the solution to a previous and similar problem [Carbonell 1986]. This process involves searching for past problems similar to the problem to solve and to transform the solution of a past problem to the new problem. The search for previous similar problems and the transformation of past solutions are two distinct and challenging problems.

[Carbonell 1986] distinguishes between transformational and derivational analogy. In transformational analogy, cases are transformed solely on the basis of their features. No, or little background knowledge is used. On the other hand, derivational analogy requires that the knowledge used in the solution of a previous problem be transferred in the solution of a new problem. Classical case-based reasoning is equivalent to transformational analogy.

The search for previous similar problems consists in searching in memory for a problem sharing significant aspects with the problem to solve. The significant aspects may be some features occurring directly in both problems. They may also be different

features in each problem which are similar according to some background knowledge (also called matching knowledge in this context).

The transformation of a past solution to a new one can be done using transformation operators which translate an operation from the past solution into a new operation in the new solution. This is called analogical transformation in [Carbonell 1986] and structural adaptation in [Riesbeck et al. 1990].

The derivation process that was used to derive the solution of a previous problem can also be used to build the solution to a new and similar problem. Here, the derivation process is composed of the set of rules used in the derivation of the solution. This set of rules is applied to the new problem making its solution possible. This method is called derivational analogy in [Carbonell 1986] and derivational adaptation in [Riesbeck et al. 1990].

In our system, we use analogical reasoning to complete partial explanations. We use structural adaptation to transform the solution of a previous and similar problem to the solution of a new problem. The explanation of the previous problem is a complete explanation with respect to the previous problem and a partial explanation with respect to the new problem.

For example (figure 3.2), suppose that we have to explain a new training example NTE. We do not have a solution for the new training example because the domain theory is incomplete. However, we have the solution of a similar training example PTE. The solution of PTE, which is the explanation Expl, provides a partial explanation for NTE. In our system, we use structural adaptation to transform Expl into Exp2, which is the plausible explanation of NTE. The adaptation is made using the goal common to both the feature of Expl being replaced and the feature of NTE replacing it. Those two features are called analogous features.

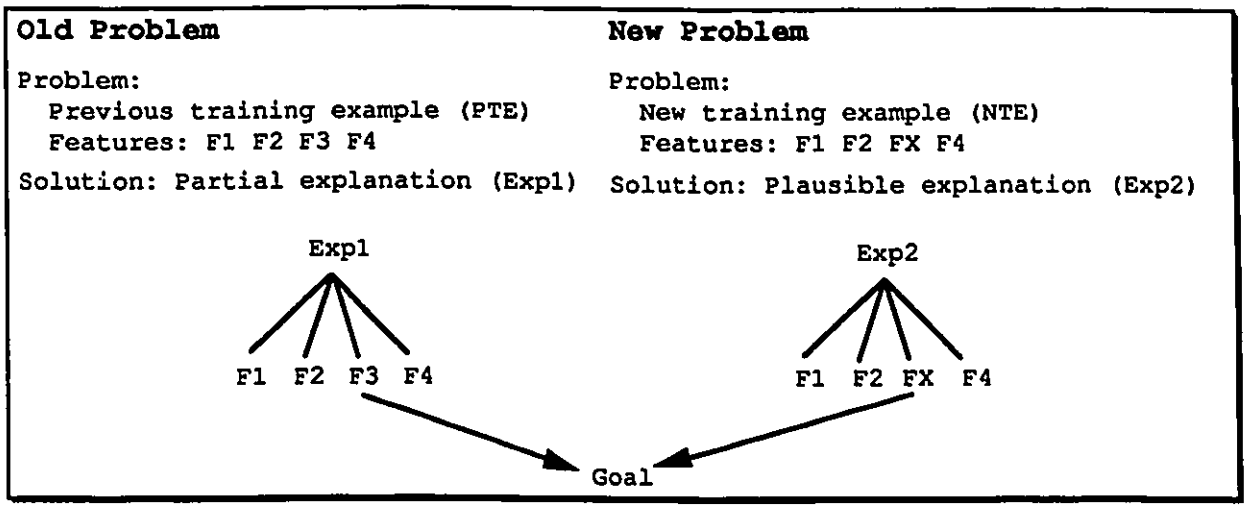

Figure 3.2 Analogical Reasoning applied to partial explanations

In practice, a plausible explanation in our system is built by first re-using the proven antecedents of the selected partial explanation. Second, the unproven antecedents of the partial explanation are replaced by *analogous* training example facts. An analogous training example fact is a fact which shares its goal with the unproven antecedents.

The resulting plausible explanation is used to create new rules. From these new rules, the new frames can be extracted and added to the domain theory. Thus, analogical reasoning allows us to repair an incomplete domain theory.

It is not uncommon to obtain for a training example a partial explanation having, as proven antecedents, a group of facts, and another partial explanation having, as proven antecedents, another group of facts. Instead of selecting the best partial explanation in all cases, our approach has a way to combine partial explanations together into a unique plausible explanation.

Our method of combining partial explanations works by re-using all proven antecedents and by applying analogical reasoning to the unproven antecedents. Consider the example of fig 3.3 where there are two partial explanations for the training example NTE. The first partial explanation, Expl, has two common features with NTE: F1 and F3. The second partial explanation, Exp2, has also two common features with NTE: F2 and F6. The plausible explanation Exp3 can be built by integrating Expl and Exp2 together. F5 and F4 are replaced by FX and FY because of the goals Goal1 and Goal2 respectively . As the

order is relevant, the order of the leaves in Exp3 is obtained using the order of the features in NTE.

When partial explanations are combined, it is common to find several unproven antecedents of the partial explanations that have no analogous features in the training example. An analysis of the goals of these unproven antecedents usually reveals that they can be removed mutually from the plausible explanations. An example of this situation is presented later.

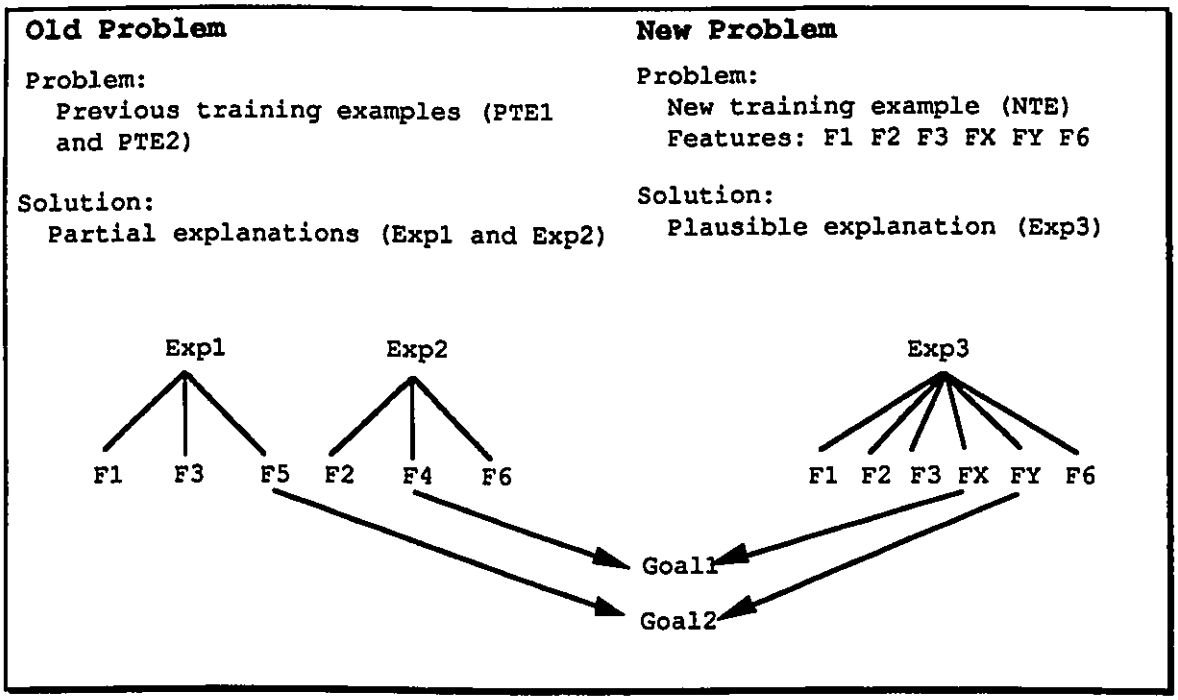

Figure 3.3 Combining partial explanations into a plausible one using analogical reasoning

3.4.4 Case-based reasoning applied to training example

Case-based reasoning systems solve new problems by adapting solutions that were used to solve old problems [Riesbeck et al. 1990]. The difference between case-based reasoning, as presented here, and analogical reasoning, as presented in the previous section (section 3.4.3), is that case-based reasoning manipulates only cases in the form of feature vectors. In the previous section, we used analogical reasoning to handle a more complex construct: the explanation.

We rely on case-based techniques to handle the training examples for which no explanation is possible. These training examples represent concepts that are not explainable with the domain theory because of their complexity or because they are exceptions to general rules. The case-based system can be seen as providing an extension to the incomplete domain theory.

In the case-based approach, the training example is considered only as a feature vector. The role of the case-based system is to find an appropriate case in the case-base and to apply it to the training example.

The case-based system will retrieve a case for a training example if there is a match between the case features and the training example features and if the order of the case features is preserved in the training example. There is a match between a case feature and a training example feature if the case feature name is the same as the training example name and the variables in the case unify with the constants in the training example. Matching features are unified, and the unification is propagated to other features of the case. There will also be a match when the constants in case features are associated with the same constants in the training example. Finally, there is a match when a feature in the case can be associated with an analogous feature in the training example as long as all the above conditions hold. When more than one case is retrieved, a heuristic similar to the one used to rank the partial explanations will select the most applicable one.

The cost of matching and the size of the case-base make the case-based approach secondary to the rule-based approach in our methodology to deal with the incomplete domain theory.

Chapter 4

Learning In Software Engineering (LISE)

4.1 EBL applied to the problem of system specification

We determined that EBL was the learning method the most suitable for the task of transforming informal and non-operational user requirements into formal and operational software specifications.

As shown on figure 4.1, the requirements of regular EBL directly map to the requirements of our task. The goal concept in LISE is the new specification to be learned. The training example is the description of an activity which corresponds to the goal concept. The domain theory contains the specifications of the individual operations already defined and also the definitions of goals. The operationality criterion in our task requires that the resulting specification produced for a goal concept be expressed in terms of primitive operations only.

The purpose of LISE is to produce a specification for each user requirement given as training example. Unfortunately, the domain theory may be incomplete when the analyst starts building the specification of a software system. The regular EBL process will not work with an incomplete domain theory. Consequently, we have to supplement EBL with our approach to deal with the incomplete domain theory.

4.2 The architecture of LISE

LISE is an integrated learning system using EBL augmented with our approach to deal with the incomplete domain theory. LISE is composed of three modules (Fig 4.2).

```
Given:
      *  Goal Concept
            A transaction's specification to be learned.
      *  Training Example
            An example of a user activity corresponding to the
            specification to be learned.
      *  Domain Theory
            The domain theory contains the system specification for the
            system developed (since the system is under development, the
            specification might not be complete). The domain theory also
            contains the goals for some operations.
      *  Operationality Criterion
            The transaction's specification must be given in terms
            of primitive operations.

Determine:
      *  If an explanation is possible, produce a definition of the
      training example in terms of primitive operations, otherwise,
      extend the domain theory (the specification) so that the
      transaction is covered and produce a definition of the training
      example in terms of primitive operations.
```

Figure 4.1 A statement of the problem in LISE

The first module of LISE is **ELLI**. ELLI stands for **Explanation-based Learning for LISE**. ELLI is the module which receives the training example from the user. The main engine of ELLI is the conventional EBL with the usual deductive logical inference. That engine uses the domain theory in order to build an explanation for the training example. The EBL version in ELLI is designed in such a way that if the regular EBL does not produce a complete explanation, a set of one or more partial explanations will be produced.

ELLI is designed to apply the first step of our approach to deal with the incomplete theory. For that purpose, ELLI includes two procedures. The first procedure is a selection heuristic used to score and rank the set of partial explanations. The second

procedure is an inference engine which uses abduction to solve the unproven antecedents of the partial explanations.

If ELLI is not able to build a plausible explanation, the training example and the set of partial explanations will be forwarded to the next module. The set of partial explanations is still ranked from the best to the worst. The heuristic needs not to be applied again.

The second module of LISE is **LARS**. LARS is an acronym for **LISE's Analogical ReaSoner**. LARS is designed to transform one or several partial explanations into a plausible explanation. LARS applies the second step of our approach to deal with the incomplete domain theory.

LARS builds a plausible explanation by re-using the proven antecedents of a partial explanation and by applying analogical reasoning to replace the unproven antecedents by analogous training example facts. LARS uses the goals from the domain theory to determine which facts are analogous to the unproven antecedents. The plausible explanation, if produced, is thereafter used to create new frames to be added to the domain theory.

If the procedure used by LARS does not succeed producing a plausible explanation for a training example, the training example is sent to the next module and the partial explanations are discarded.

The next and last module of LISE is **CARL (CAse-based Reasoning for LISE)**. CARL applies the third step of our approach to deal with an incomplete domain theory. It is applied when the two previous modules did not succeed to produce a plausible explanation.

CARL is a case-based reasoning system. When it receives a training example, CARL searches the case-base for the case providing the best match. The matching knowledge required to evaluate the match is the set of goals in the domain theory.

The retrieved case provides a specification for the training example. The dashed arrow from CARL to the domain theory on figure 4.2 indicates that the user has the option to augment the domain theory.

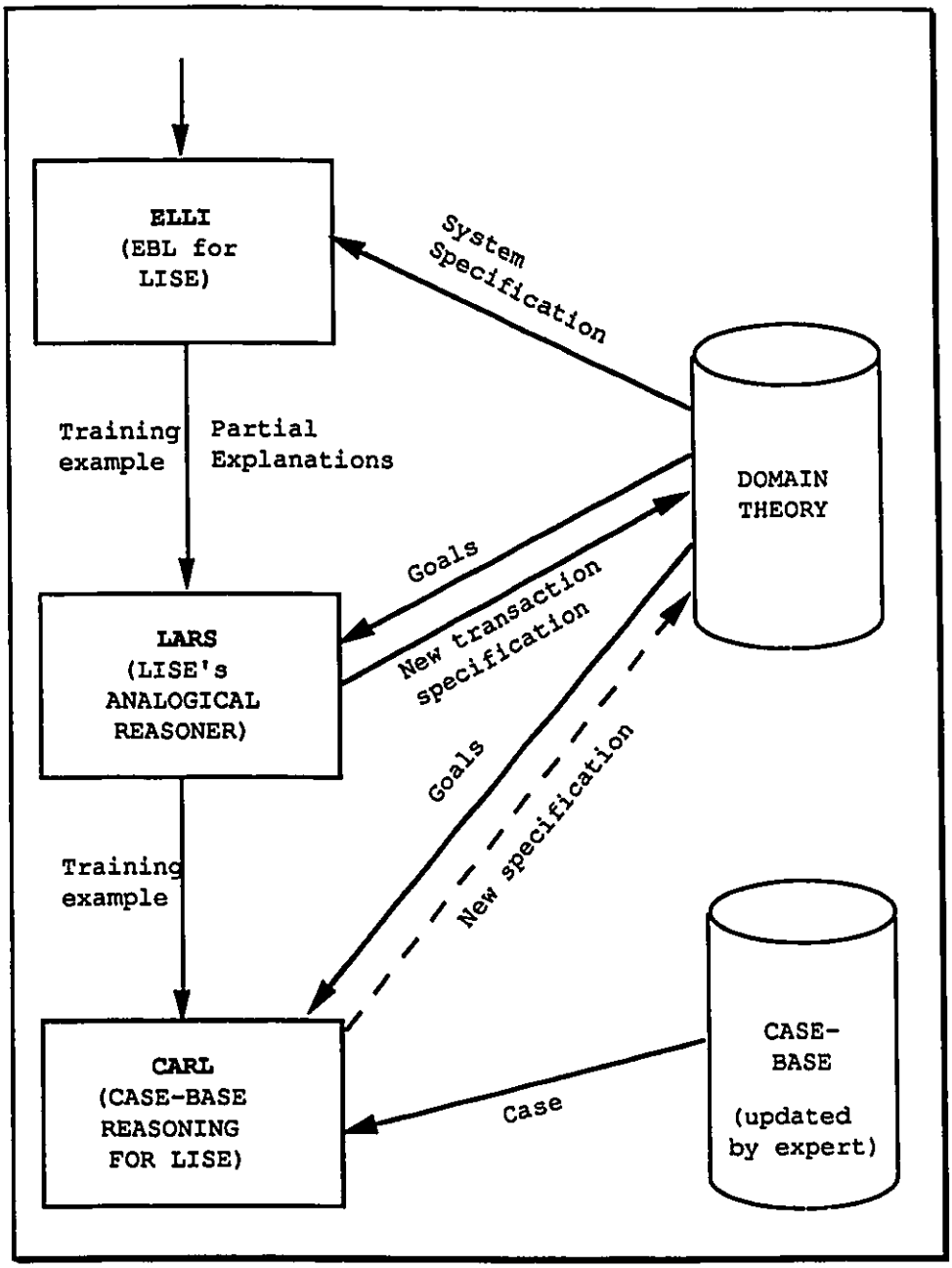

Fig 4.2 The architecture of LISE

4.3 The domain theory in LISE

In LISE, the domain theory represents the specification of a software system. The specification of the system is given in terms of objects and operations

applicable to objects. The objects represent the static properties of the system, whereas the operations represent the dynamic properties of the system.

The domain theory also includes goals used by LARS for analogical reasoning and by CARL for case matching.

4.3.1 Representing software specifications in the domain theory

The domain theory consists of frames arranged in a hierarchy and allowing multiple inheritance of properties. Each frame specifies an object or an operation using a set of properties. One common property of all frames is `isa`, which links a frame to its parents.

Frames representing objects are located under the top-level frame ENTITY. The hierarchy allows objects to be generalizations and specializations of other objects. As an example, figure 4.3 shows a domain theory for a banking system. It contains a frame named `person` and another frame named `client`. The frame `client` is a specialization of `person`.

Some frames describing objects have rules in their property slots. For example, the frame client of figure 4.3 has a property called `necessary_condition`. That property states that a person is a client if the person has an account or a safety-deposit box.

The frames representing operations are located under the frames ACTION and TRANSACTION. Operations specified as ACTIONs in the hierarchy are non-primitive operations. ACTIONs are defined by a precondition stating the condition under which a primitive operation can be applied, and a procedure which includes the single primitive operation. Primitive operations are atomic and they are not defined anywhere. Operations specified as TRANSACTIONs in the hierarchy are defined by a precondition and by a procedure containing primitive and non-primitive operations arranged in a fixed sequence.

```
person
isa:  ENTITY
  name
  address
  phone_number

client
isa:  person
  account
  goal(account,identify_client)
  credit_margin
  goal(credit_margin,protect_bank_interest)
  safety_box
  necessary_condition:
    (client(Person) <- account(Person,Account))
    or
    (client(Person) <- safety_box(Person,Safety_box))

withdraw(Person,Amount)
isa:  TRANSACTION
  precondition:
    account(Person,Account)
    goal(account,identify_client)
  procedure:
    debit(Account,Amount)
    issue_money(Person,Amount)

debit(Account,Amount)
isa:  ACTION
  precondition:
    balance(Account,Balance)
    goal(balance,protect_bank_interest)
    Balance > Amount
  procedure:
    subtract_from_balance(Accou it,Amount)
    goal(subtract_from_balance,record_transaction)

deposit(Person,Amount)
isa:  TRANSACTION
  precondition:
    account(Person,Account)
    goal(account,identify_client)
  procedure:
    receive_money(Person,Amount)
    credit(Account,Amount)

credit(Account,Amount)
isa:  ACTION
  precondition: nil
  procedure:
    add_to_balance(Account,Amount)
    goal(add_to_balance,record_transaction)
```

Figure 4.3 Frames of the domain theory for banking

The representation used in the domain theory for the system specification is inspired from the Extended Semantic Hierarchy Model (SHM+) [Brodie et al. 1984]. SHM+ is a conceptual data model where a unified frame-based notation is used to represent both the structural and the behavioral properties of a system. In SHM+, the primitive operations correspond to database operation such as INSERT, UPDATE, and DELETE. The primitive operations in LISE are also database oriented. They could be implemented by packages in ADA or mapped to any programming language or database manipulation language (DML).

With regard to the EBL operationality criterion, the primitive operations in the system specification are considered to be operational. In our context, this means that operational actions are to be implemented at a lower level of database design, or are already

available. The other operations described by the frames under TRANSACTION and ACTION are not primitive and therefore not operational. The EBL process will thus provide an operationalization mechanism to transform non-operational user requirements given as training examples into operational specifications.

4.3.2 Defining goals in the domain theory

During the analysis phase, the analyst has to take into account the objectives of the organization for which the system is being designed. The objectives of the organization are specified by goals. An example of a goal for a bank organization is `protect_bank_interest`.

An analyst writing a system specification has to be constantly aware of the goals of the organization. The domain theory, which contains the specification, must also include these goals.

A goal is attached to a property of an entity or an operation which is intended to enforce it. An example of a goal in the domain theory is:

```
goal(credit_margin(Person,Margin),protect_bank_interest)
```

which means that the goal `protect_bank_interest` is enforced by the property `credit_margin` of a person.

These goals are used by the analogical reasoning process used by LARS and by the case-base matching process used by CARL.

4.3.3 Transforming frames into rules

Before ELLI can build an explanation, the frames of the domain theory need to be transformed into rules. This change of representation is performed by LISE itself. A rule is formed from a frame by taking the frame name as the consequent of the rule and by taking the conjunction of the frame slots as the antecedents. A rule is also created to express the link between each frame and its parent frame. The rules contained within the frames remain as rules.

4.3.4 Definitions

Definitions of the major elements of the domain theory are following:

Definition 4.1: a **primitive operation** is an atomic operation. Thus, there is no specification for a primitive operation in the domain theory, However, the primitive operations are used in the definition of other non-primitive operations in the domain theory. A primitive operation is operational.

Definition 4.2: a **non-primitive operation** is an operation specified in the domain theory. A non-primitive operation is not operational, meaning that it must be decomposed into primitive operations for implementation purposes.

Definition 4.3: TRANSACTION. All the operations defined as TRANSACTIONs are non-primitive operations. They are defined in terms of a precondition and a procedure. The procedure contains a sequence of primitive and non-primitive operations.

Definition 4.4: ACTION. All the operations defined as ACTIONs are non-primitive operations. They are defined in terms of a precondition and a procedure. The procedure contains one primitive operation. ACTIONs provide a mean to attach a precondition to a primitive operation. The difference between TRANSACTIONs and ACTIONs is that the former can have several primitive and non-primitive operations in its procedure whereas the latter can have only one primitive operation in its procedure.

4.4 Presentation of LISE using the banking domain example

A large example, using the banking domain, was developed with LISE. The example will be used to show each module of LISE in action.

4.4.1 Explanation-based learning for LISE (ELLI).

The input to our learning system consists of positive training examples. Each training example corresponds to a user requirement. User requirements are activities, or sequence of actions, carried out by the users. Fig. 4.4 shows a training example for the user requirement for the activity `withdraw`. The training example is an actual withdrawal where a person named `bob` withdraws `100` dollars.

The training example in figure 4.4 is given as a sequence of four facts. The first fact, `account(bob,account_1)`, asserts that `bob` owns an account and identifies this account as being `account_1`. The second fact, `address(bob,101_Colonel_by)`, asserts that the address of bob is `101_Colonel_by`. The third fact, `balance(account_1,150)`, asserts that the balance of `account_1` is `150` dollars. The fourth fact, `subtract_from_balance(account_1,100)`, asserts that `100` dollars were subtracted from the `balance` of `account_1`. The last fact, `issue_money(bob,100)`, asserts that the bank issued `100` dollars to `bob`.

The ordering of the facts in the training examples is important because they represent the time. In the training example `withdraw(bob,100)`, it is important that the verification of the balance (i.e. `balance(account_1,150)`) takes place before the money is taken out of the client's account (i.e. `subtract_from_balance(account_1,100)`).

```
The training example is: withdraw(bob,100)
The facts are:
   account(bob,account_1)
   address(bob,101_Colonel_by)
   balance(account_1,150)
   subtract_from_balance(account_1,100)
   issue_money(bob,100)
```

Fig 4.4 The training example of `withdraw(bob,100)`

ELLI uses the domain theory to build an explanation of the training example of `withdraw(bob,100)`. The explanation is built using rules extracted from the frames of the domain theory. ELLI finds the rule:

```
Rule 1      withdraw(Person,Amount) <-
               account(Person,Account),
               debit(Account,Amount),
               issue_money(Person,Amount).
```

and tries to prove the first antecedent `account(Person,Account)`. The antecedent is proven and the binding becomes `bob/Person` and `account_1/Account`[2]. The second antecedent to prove is now `debit(account_1,100)`. ELLI finds the rule

```
Rule 2      debit(Account,Amount) <-
               balance(Account,Balance),
               Balance > Amount,
               subtract_from_balance(Account,Amount).
```

The bindings obtained in rule 1 are propagated to rule 2. Rule 2 creates the new bindings `150/balance` and `100/Amount`. ELLI then tries to prove `balance(account_1,150)`, `150 > 100` and `subtract_from_balance(account_1,100)`. The consequent `debit(account_1,100)` is proven since the three antecedents of the rule correspond to training example facts. ELLI resumes to prove the last antecedent of Rule `1`, `issue_money(bob,100)`, which also corresponds to a training example fact. Thus, the complete explanation of fig 4.5 is obtained.

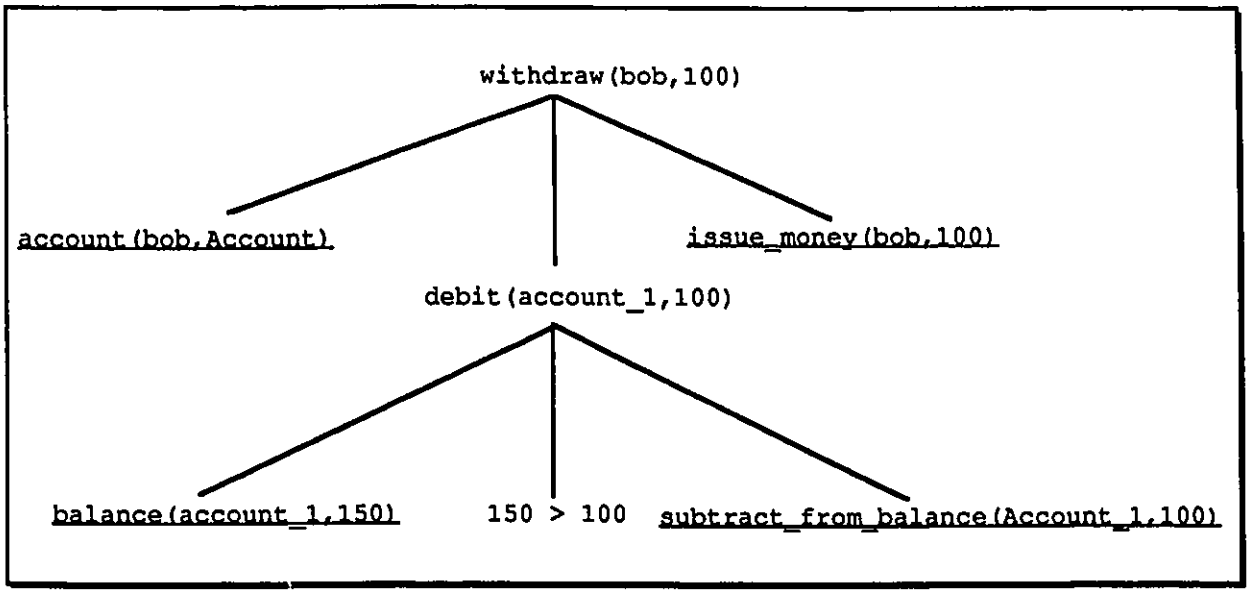

Fig 4.5 The explanation tree of `withdraw`

[2] Here we use the notation constant/variable to describe the binding.

In the explanation tree (figure 4.5), the underlined leaves correspond to antecedents of the rules used in the explanation for which there was a training example fact. Any training example fact that does not appear in the explanation tree (e.g. `address(bob,101_Colonel_by))` is an irrelevant fact. Irrelevant facts do not contribute in explaining how the training example is an instance of the goal concept.

There is also a leaf in the tree that is not underlined. That leaf, containing the fact `150 > 100`, was not given as part of the training example. The leaf was introduced by the antecedent `Balance > Amount` of rule 2 where `Balance` was unified to `150` and `Amount` was unified to `100`.

It is interesting to note that the leaves of the explanation tree are all primitive operations according to our domain theory. As the purpose of our system is to obtain a specification of the user requirement provided by the training example in terms of primitive operations, the leaves can be gathered in order to obtain a new definition of `withdraw(Person,Amount)` in terms of primitive operations. The order of the leaves in the specification must respect the order of the antecedents in the rules that introduced them. Fig. 4.6 illustrates the output provided by ELLI for the training example of fig 4.4.

```
Successful explanation
----------------------
Goal:
  withdraw(Person,Amount)
Result:
  account(Person,Account),
  balance(Account,Balance),
  Balance > Amount,
  subtract_from_balance(Account,Amount),
  issue_money(Person,Amount)
```

Fig. 4.6 Output of ELLI for a complete explanation

4.4.2 Using Abduction in ELLI.

When an explanation is not possible using the deductive inference, ELLI produces one or more partial explanations. If more than one partial explanation is produced, ELLI will apply our heuristic to select an explanation. Once the best partial explanation is identified, ELLI will apply abduction in order to complete it.

As seen in chapter 3, abduction is the generation of hypotheses, which, if true, would explain observed facts. More precisely, if the rule `Q <- P` and the fact `Q` are given, then the desired abductive conclusion would be `P`. `P` can be characterized as being a hypothesis because there could exist another rule `Q <- P'` which could have been used to derive `Q`.

Abduction is used to complete a partial explanation. Using the training example features as a set of facts, ELLI attempts to draw hypotheses from the unproven antecedents using abduction. If hypotheses can be drawn for each unproven antecedent, the partial explanation can be completed. Using the above example of the rule `Q <- P` and `Q`, `P` would be an unproven antecedent and `Q` would be a fact given as a training example feature. If `P` is the only unproven antecedent, then a hypothesis can be drawn to account for the occurrence of `Q` in the training example, and the partial explanation can be completed.

```
The training example is: withdraw(bob,100)
The facts are:
   client(bob)
   address(bob,101_Colonel_by)
   balance(account_1,150)
   subtract_from_balance(account_1,100)
   issue_money(bob,100)
```

Fig. 4.7 The training example on which ELLI uses abduction

The next scenario illustrates the usage of abduction to complete a partial explanation. The training example (Fig. 4.7) is the requirement for the transaction `withdraw(bob,100)` of Fig 4.4 where the fact `account(bob,account_1)` was replaced by `client(bob)`.

When ELLI is presented with the training example, it attempts to explain it using rule 1 and rule 2. ELLI was not able to prove the antecedent `account(Person,Account)` since the training example does not contain it as a fact and that `account(Person,Account)` is not the consequent of any rules in the domain theory. The antecedent `account(Person,Account)` was marked as unproven and ELLI proceeded with the proof as if nothing happened. If more than one partial explanation were produced, the heuristic would have been used to select the best one. In this example, we consider that a single partial explanation, shown in fig 4.8, is produced.

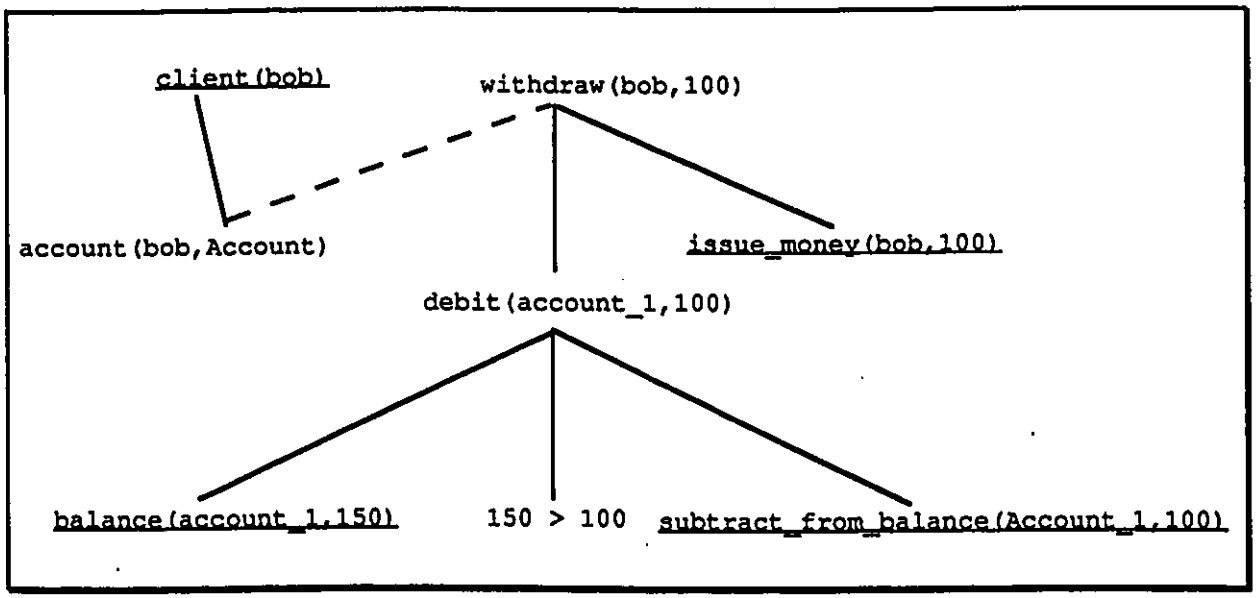

Fig. 4.8 Explanation tree resulting from using abduction

The partial explanation contains a single unproven antecedent - `account(Person,Account)`. ELLI will use abduction to complete the partial explanation. In abduction, we need a candidate rule `Q <- P` and a fact `Q` in order to abduce `P`. `P` here is the unproven antecedent - `account(Person,Account)` - that we wish to transform to a hypothesis. ELLI finds the candidate rule `client(Person) <- account(Person,Account)` in the domain theory and will try to find the fact `client(Person)` in the training example. The fact `client(bob)` is in the

training example and can be used in conjunction with the candidate rule to raise the hypothesis `account(bob,Account)`. The partial explanation can be transformed into a plausible explanation with the new hypothesis.

Reasoning abductively is very common for humans. In the situation above, the teller reasons abductively when he/she sees a client he/she knows to hypothesize that the client owns an account.

The leaves of the plausible explanation can be gathered to produce the specification for the training example in terms of primitive operations. Fig 4.9 shows the output that will be presented by ELLI to the user.

```
Successful explanation
----------------------

MESSAGE from LISE concerning the next explanation

Abduction was used with hypothesis on rule:
   [client(bob) <- [account(bob,Account)]]
   Other possibilities are:
   [client(bob) <- [safety_box(bob,Safety_box)]]

Goal:
  withdraw(Person,Amount)
Result:
  account(Person,Account),
  balance(Account,Balance),
  Balance > Amount,
  subtract_from_balance(Account,Amount),
  issue_money(Person,Amount)
```

Fig. 4.9 Output of ELLI for the example of abduction

Abduction only allows to make a hypothesis for a certain fact `P` using a rule `Q <- P` because there may exist another rule `Q <- P'`, which would have been used with the fact `P'` to deduce the fact `Q`. In the example considered here, our hypothesis is: bob is a client because bob owns an account. However, the domain theory contains the rule `client(Person) <- safety_box(Person,Safety_box)`. This rule states that bob is a client if bob owns a safety-deposit box. ELLI will include a message in its output to indicate to the user that a hypothesis was raised in the construction of the specification.

As with the previous example, the irrelevant feature `address(bob,101_Colonel_By)` was not used in the explanation and is not added to the specification of `withdraw(Person,Amount)`.

4.4.3 LISE's analogical reasoner (LARS)

As mentioned earlier, ELLI will generate one or more partial explanations for a training example when the domain theory does not contain the rules required to explain it. Each partial explanation is built using rules which contain antecedents expressed using predicates that are also used in the training example.

When many partial explanations are produced, a heuristic, described in section 3.4.1, will be used to determine the best one. Abduction will then be applied to the best partial explanation in order to transform it into a plausible explanation. In the event that abduction does not work, the best partial explanation will be forwarded to LARS where analogical reasoning will be used in an attempt to build a plausible explanation.

To build the plausible explanation, LARS will take the best partial explanation and will keep its proven antecedents. LARS will then take each unproven antecedent of the partial explanation and will replace it by an *analogous* training example fact. An unproven antecedent and a training example fact are analogous if their predicates have the same goal according to the domain theory. If each unproven antecedent can be replaced by an analogous training example fact, the resulting explanation is called a plausible explanation.

The leaves of the plausible explanation can be extracted to produce the specification of the user requirement in terms of primitive operations. The order of the leaves is important and must be replicated in the specification.

Knowledge-level learning [Dietterich 1986] takes place when a learning system augments the deductive closure of its domain theory. Consequently, our approach permits knowledge-level learning since the module LARS augments LISE's domain theory.

LARS will be demonstrated using the training example for `borrow(bob,1000)` shown in figure 4.10.

```
The training example is: borrow(bob,1000).

The facts are:
  account(bob,account_1)
  credit_margin(bob,3500)
  record_loan(bob,1000)
  issue_money(bob,1000)
  car(bob,car_of_bob)
  value(car_of_bob,12000)
```

Fig. 4.10 The training example for `borrow(bob,100)`

The domain theory is incomplete because the specification of the transaction `borrow` required to explain the training example is missing. Consequently, two partial explanations will be produced. Fig. 4.11 illustrates the output presented by LARS for the training example of fig 4.10. In the output, the first message indicates that there was no successful explanation and that one or more partial explanations were obtained instead.

The first partial explanation was produced using the specification of the transaction `withdraw(Person,Amount)`. LARS indicates that there are two common features to `withdraw` and `borrow`: `issue_money(Person,Amount)` and `account(Person,Account)`. LARS goes on to display that the unexplained features of withdraw were `subtract_from_balance(Account,Amount)`, `balance(Account,Balance)` and `Balance > Amount`. Then, as seen in figure 4.11, LARS displays the facts of the training example that were not used in the explanation which are: `credit_margin(bob,3500)` and `record_loan(bob,1000)`.

The second partial explanation in the output of LARS was produced using the specification of `deposit(Person,Amount)`. The common feature between `deposit` and `borrow` is `account(Person,Account)`. The unexplained features of deposit are: `receive_money(Person,Amount)` and `add_to_balance(Account,Amount)`. Finally, the features of the training example of `borrow` that were not used in the partial explanation produced using `deposit` are `credit_margin(bob,3500)` `record_loan(bob,1000)` and `issue_money(bob,1000)`.

The last lines on the output show the result of applying our heuristic to rank the partial explanations. The partial explanation obtained using `withdraw` is the best partial explanation since it received the highest value.

```
NO successful explanation; Partial explanation(s) follow:

PARTIALLY explained using transaction: withdraw(Person,Amount)

Common features: [issue_money(Person,Amount),account(Person,Account)]

Unexplained features of withdraw(Person,Amount) :
[subtract_from_balance(Account,Amount),balance(Account,Balance),
 Balance > Amount]

Features of the training example not used in explanation :
[credit_margin(bob,3500),record_loan(bob,1000)]

PARTIALLY explained using transaction: deposit(Person,Amount)

Common features: [account(Person,Account)]

Unexplained features of deposit(Person,Amount) :
[receive_money(Person,Amount),add_to_balance(Account,Amount)]

Features of the training example not used in explanation :
[credit_margin(bob,3500),record_loan(bob,1000),
 issue_money(bob,1000)]

There were two explanations found...

The partial explanations are, from the best to the worst:
      Score: -3.0   Transaction name: withdraw(Person,Amount)
      Score: -4.0   Transaction name: deposit(Person,Amount)
```

Fig 4.11 Output produced by LARS for the training example `borrow(bob,1000)`

Another way to picture the information in the output produced by LARS is by using the tree notation. Fig 4.12 shows the partial explanation produced using `withdraw`. The underlined antecedents, `account(bob,account_1)` and `issue_money(bob,1000)`, are proven since they correspond to training example features.

The unproven antecedents are linked using dashed lines. They correspond to the three antecedents of rule 2, the rule for `debit(account,amount)`. Fig. 4.13 illustrates a partial explanation obtained using the specification of the transaction `deposit(Person,Amount)`. There is only one proven antecedent, `account(bob,account_1)`, in that partial explanation.

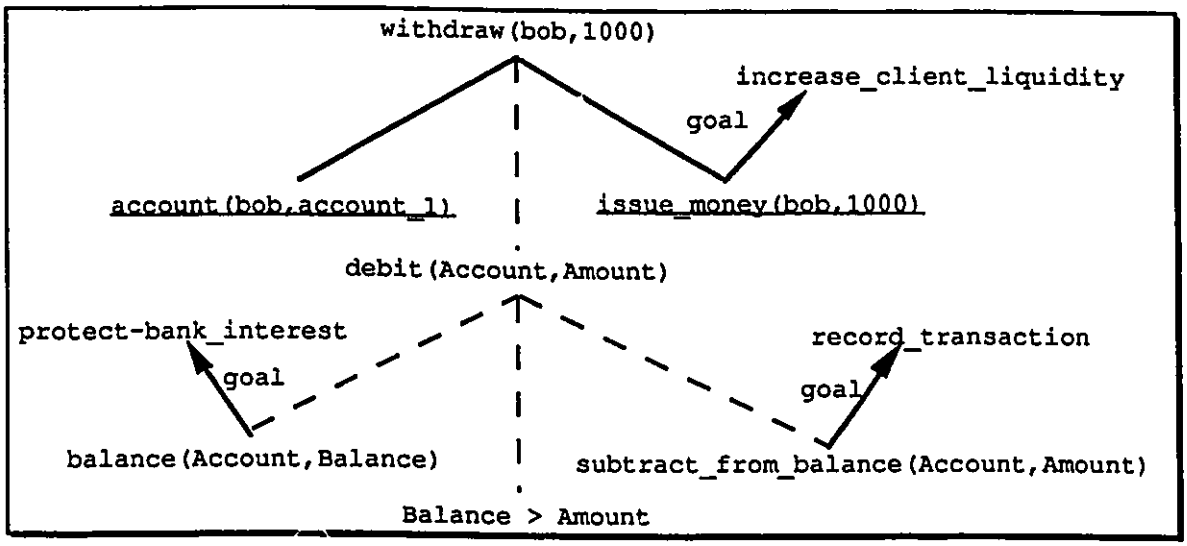

Fig. 4.12 Partial explanation for `borrow` produced using `withdraw`

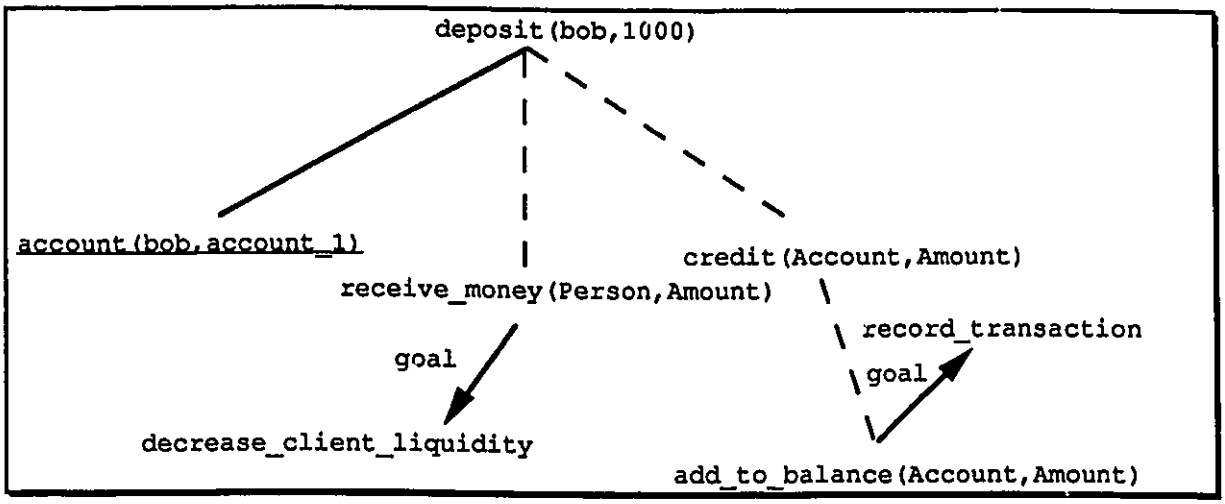

Fig. 4.13 Partial explanation for `borrow` produced using `deposit`

To build a plausible explanation of `borrow`, LARS first creates an explanation tree with `borrow(bob,1000)` in the root. LARS will then insert all the subtrees of the root of the partial explanation under the root of the plausible explanation. At this point, the plausible explanation is similar to the partial explanation of figure 4.12 except for the label of the root node which is `borrow(bob,1000)`.

LARS will then visit each leaf of the plausible explanation and do the following:

1. if the leaf is a proven antecedent, LARS will keep it in the plausible explanation, and,

2. if the leaf is an unproven antecedent, LARS will consult the domain theory and will attempt to replace it by an analogous feature taken from the training example.

The first leaf visited is `account(bob,account_1)`. This leaf is a proven antecedent and is kept in the tree. The second leaf visited is `balance(Account,Balance)`. This leaf is an unproven antecedent. LARS will consult the domain theory to find the goal of `balance(Account,Balance)`. The goal found is `protect_bank_interest`. LARS will search the training example for a fact having the same goal. LARS finds that the goal of `credit_margin(bob,3500)` is also `protect_bank_interest` and will then replace `balance(Account,Balance)` by `credit_margin(bob,3500)`.

The next leaf encountered by LARS is `Balance > 1000`. The substitution of `balance(Account,Balance)` by `credit_margin(bob,3500)` indicated that the variable `Balance` can be replaced by `3500`. The leaf now becomes `3500 > 1000`. This antecedent becomes proven and is kept in the tree.

The next leaf visited is `subtract_from_balance(account_1,1000)`. This leaf is an unproven antecedent. LARS will search the training example for a fact having the same goal as `subtract_from_balance`, which is `record_transaction`. The feature `record_loan(bob,1000)` of the training example is found to have the same goal. Thus, `subtract_from_balance(account_1,1000)` is replaced by `record_loan(bob,1000)`.

The last leaf visited is the proven antecedent `issue_money(bob,1000)` and it is kept in the plausible explanation.

The training example facts `car(bob,car_of_bob)` and `value(car_of_bob,12000)` did not appear in the partial explanation and were not selected by LARS. Consequently, they were singled out as irrelevant features. The plausible explanation built by LARS for `borrow(bob,1000)` is illustrated on figure 4.14.

The next step is to extract the new specification corresponding to `borrow(Person,Amount)`. The plausible explanation of fig 4.14 contains all the primitive operations but still needs to be generalized. The generalization process consists in substituting constants of the training example for variables. The domain theory and the unification of the proven antecedents with training example facts are used to determine which variable to use for every constants.

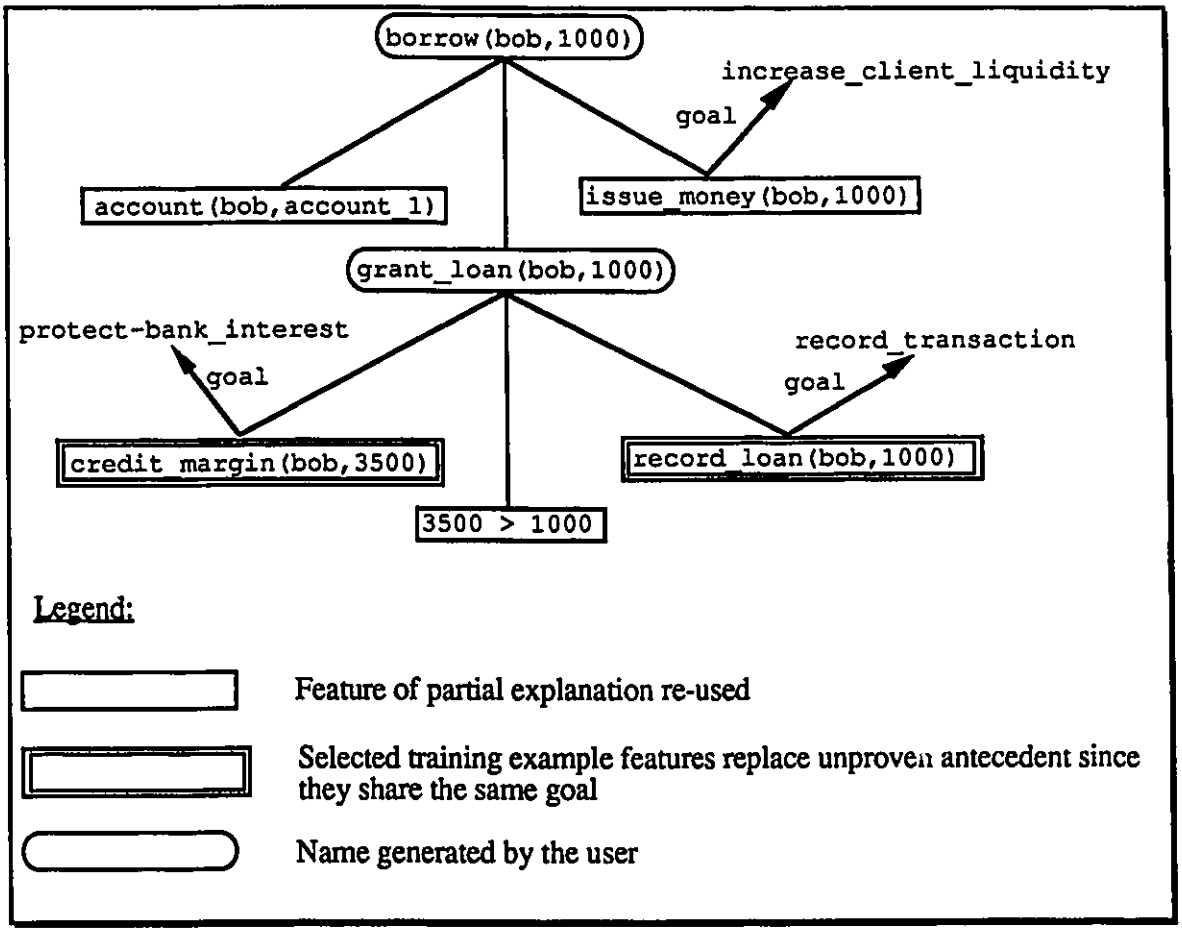

Fig. 4.14 The plausible explanation for `borrow` produced using `withdraw`

We need to enter a new name for a node in a tree when one or several of its nodes were changed. This ensures that the domain theory will not contain any ambiguous

rules and will thus preserve the consistency of the specification. In our system, the user is required to provide the name of the new node and the system verifies for uniqueness. In the example of `borrow`, the node `debit(Account,Amount)` needs to be renamed because its subtree was changed. The user selected `grant_loan` as a new name for the node. Our system takes care of selecting the parameter for the new action names. For this example, the system selected `(Person,Amount)` as the new parameters.

The next and the last step is to extract the new frames from the plausible explanation tree in order to augment the domain theory. Each subtree translates into a frame. The name of the subtree's root becomes the name of the frame and the offsprings of the root become the slots. The root is mapped onto a TRANSACTION and the internal nodes are mapped onto ACTIONs. To determine if an antecedent is to be inserted in the precondition or in the procedure of the TRANSACTION or the ACTION, LARS uses the

```
Name: grant_loan(Person,Amount)
 isa: ACTION
   precondition:
     credit_margin(Person,Credit_margin)
     Credit_margin > Amount
   procedure:
     record_loan(Person,Amount)

Name: borrow(Person,Amount)
 isa: TRANSACTION
   precondition:
     account(Person,Account)
   procedure:
     grant_loan(Person,Amount)
     issue_money(Person,Amount)
```

Fig. 4.15 The frames for `grant_loan` and for `borrow`

structure of the frame that was used to build the partial explanation. For example, `credit_margin(Person,Credit_margin)` replaced `balance(Account,Balance)`,

which was part of the precondition of `debit(Account,Amount)`. Thus, `credit_margin(Person,Credit)` will be inserted in the precondition of the frame replacing `debit(Account,Amount)`, namely `grant_loan(Person,Amount)`. The resulting frames are the one displayed on figure 4.15.

Inserting a predicate at the same location as its analogous predicate ensures that the relationship between a precondition and a procedure within a TRANSACTION or an ACTION be respected. For example, in the transaction `withdraw`, the goal of the precondition predicate `balance` was `protect_bank_interest` and the goal of the procedure predicate `subtract_from_balance` was `record_transaction`. In other words, the goal `record_transaction` is a precondition to the goal `protect_bank_interest`. The same relationship is preserved in the new transaction `borrow`.

4.4.4 Combining multiple partial explanations in LARS

In the previous example, a single partial explanation, obtained using `withdraw`, was sufficient to build the plausible explanation of `borrow`. However, it is not always possible to build a plausible explanation for a training example using only one partial explanation. The training example `transfer(Person,Amount)` (fig 4.16) will be used to illustrate such a case.

```
The training example is: transfer(bob,100).

The facts are:
   account(bob,account_1)
   account(bob,account_2)
   phone(bob,992-2318)
   balance(account_1,350)
   subtract_from_balance(account_1,100)
   add_to_balance(account_2,100)
```

Fig. 4.16 The training example for the transaction `transfer(Person,Amount)`.

The training example is a transfer of 100 dollars from an account to another one. It contains six facts. The first two facts are that `bob` owns two accounts, `account_1` and `account_2`. The following fact gives the phone number of `bob`. The two following facts assert that the balance of `account_1` was `350` and that `100` dollars was taken out of that account. The last fact asserts that `100` dollars was added to `account_2`.

Intuitively, the reader familiar with the `transfer` transaction may recognize that the transaction `transfer` is, to a certain degree, the transaction `withdraw` followed by the transaction `deposit`. There are some features of `transfer` that remind us of `withdraw` and some other features that remind us of a `deposit`. We will now see how LARS will build a plausible explanation for that training example.

The domain theory is incomplete with regard to the transaction `transfer` because it does not include the frame of `transfer`. Should there be a specification in the domain theory for `transfer`, ELLI, the first module of LISE, would deliver its specification in terms of primitive operations. ELLI will generate partial explanations for the training example. The output of ELLI is presented in fig 4.17.

```
NO successful explanation; Partial explanation(s) follow:

PARTIALLY explained using transaction: withdraw(Person,Amount)
Common features:
[subtract_from_balance(Account,Amount),balance(Account,Balance),
 account(Person,Account)]

Unexplained features of withdraw(Person,Amount) :
[Balance > Amount,issue_money(Person,Amount)]

Features of the training example not used in explanation :
[account(bob,account_2),add_to_balance(account_2,100)]

PARTIALLY explained using transaction: deposit(Person,Amount)
Common features:
[add_to_balance(Account,Amount),account(Person,Account)]

Unexplained features of deposit(Person,Amount) :
[receive_money(Person,Amount)]

Features of the training example not used in explanation :
[account(bob,account_1),substract_from_balance(account_1,100),
 balance(account_1,150)]

There was 2 partial explanation(s) found
The explanations are, from the best to the worst:
      Score: 1.0  Transaction name: withdraw(Person,Amount)
      Score: -2.0 Transaction name: deposit(Person,Amount)
```

Fig 4.17 The output of ELLI produced for `transfer(Person,Amount)`

Two partial explanations were produced for `transfer(bob,100)`. The first partial explanation, shown in figure 4.18, was obtained using `withdraw`. The proven antecedents of `withdraw`, which correspond to training example facts, are `subtract_from_balance(Account,Amount)`, `balance(Account,Balance)`, and `account(Person,Account)`. The unproven antecedent of `withdraw` is `issue_money(Person,Amount)`. The unused facts of the training example are `account(bob,account_2)` and `add_to_balance(account_2,100)`.

The second partial explanation, shown in figure 4.19, is produced using `deposit`. The proven antecedents are `add_to_balance(Account,Amount)` and `account(Person,Account)`. The unproven antecedent of `deposit` is `receive_money(Person,Amount)`. The unused training example facts are `account(bob,account_1)`, `substract_from_balance(account_1,100)` and `balance(account_1,150)`.

The heuristic was applied to the partial explanations and the one selected as the best one was the partial explanation of `withdraw`. ELLI tried to apply abduction to complete the best partial explanation with no success. ELLI tried to apply abduction to the second best partial explanation (the only other one) again with no success. Thus, the partial explanations are sent to the module LARS. Figure 4.16 and 4.17 illustrate the explanation tree of the two partial explanations.

To create the plausible explanation, LARS will proceed as discussed in section 4.4.3. A plausible explanation will be created with the label `transfer(bob,100)` in the root. Next, LARS will copy each subtree of `withdraw` as offsprings of `transfer`. Then, LARS will visit the leaves of the plausible explanation, leaving the proven antecedents (underlined in fig 4.18) untouched and replacing the unproven antecedents by analogous training example facts. At the end of this process, the plausible explanation will still have an unproven antecedent: `issue_money(Person,Amount)`.

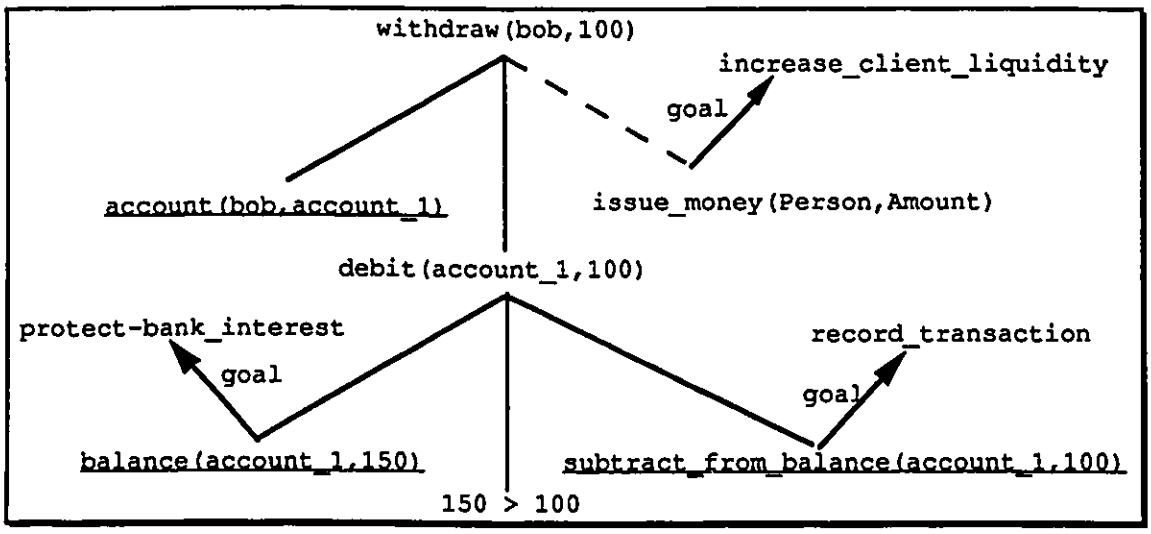

Fig. 4.18 Partial explanation for `transfer` produced using `withdraw`

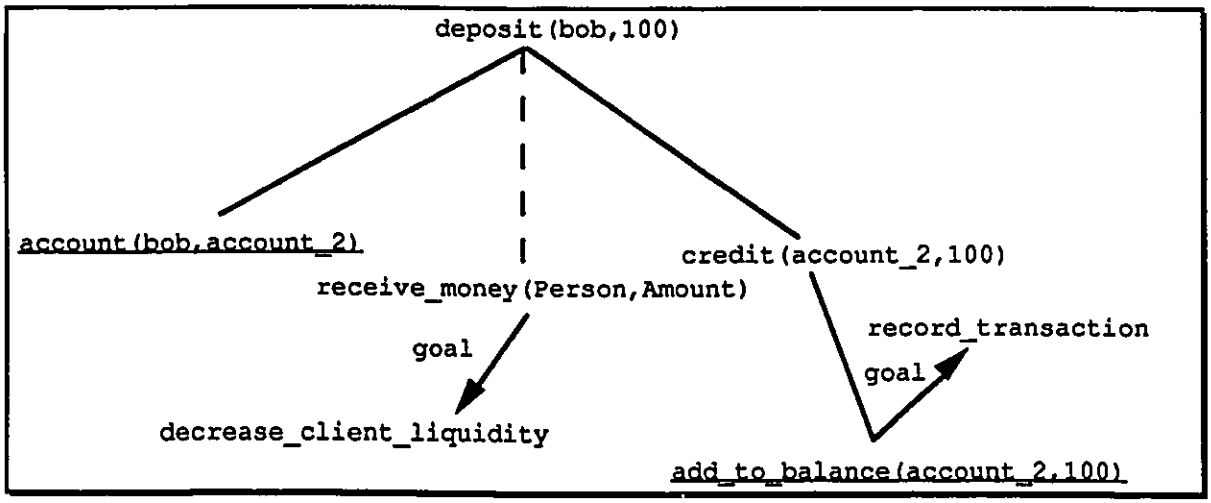

Fig. 4.19 Partial explanation for `transfer` produced using `deposit`

LARS will then examine the second best partial explanation to see if it contains training example facts that were not used in the current plausible explanation. Indeed, it finds that there are two facts, `add_to_balance(Account,Amount)` and `account(Person,Account)`, which were covered in the partial explanation of `deposit` and were not used in the plausible explanation. LARS will attempt to combine the partial explanations together so that no unproven antecedent remains.

To combine the partial explanations, LARS starts with the current plausible explanation of `transfer`. Each subtree of the second partial explanation's root (`deposit(bob,100)`) will be inserted as offspring of the root of the plausible explanation.

(`transfer(bob,100)`). The order selected for the position of the new subtrees is based on the location of the features corresponding to the proven antecedents of the subtree in the training example. For example, the subtree `account(bob,account_2)` of the second partial explanation will be inserted between the subtree `account(bob,account_1)` and the subtree `debit(account_1,100)` because, in the training example, `account(bob,account_2)` is immediately after `account(bob,account_1)`.

The combination of subtrees from two partial explanations into a single plausible explanation is possible because each subtree of the root in the partial explanations corresponds to an operation in the domain theory. The operation is primitive if the subtree is made up of only one node, and is non-primitive if the subtree is a tree itself.

The order of the subtrees in the plausible explanation is also very important and must reflect the order of the features in the training example. This is because the training example corresponds to the user requirement. The user requirement not only dictates the operations to carry out in a specification, but also dictates the order in which they must occur.

After all the subtrees of the second partial explanation have been inserted in the plausible explanation, LARS will revisit each leaf. After the second visit in our example, LARS still has two unproven antecedents. The plausible explanation at this point is shown in figure 4.20.

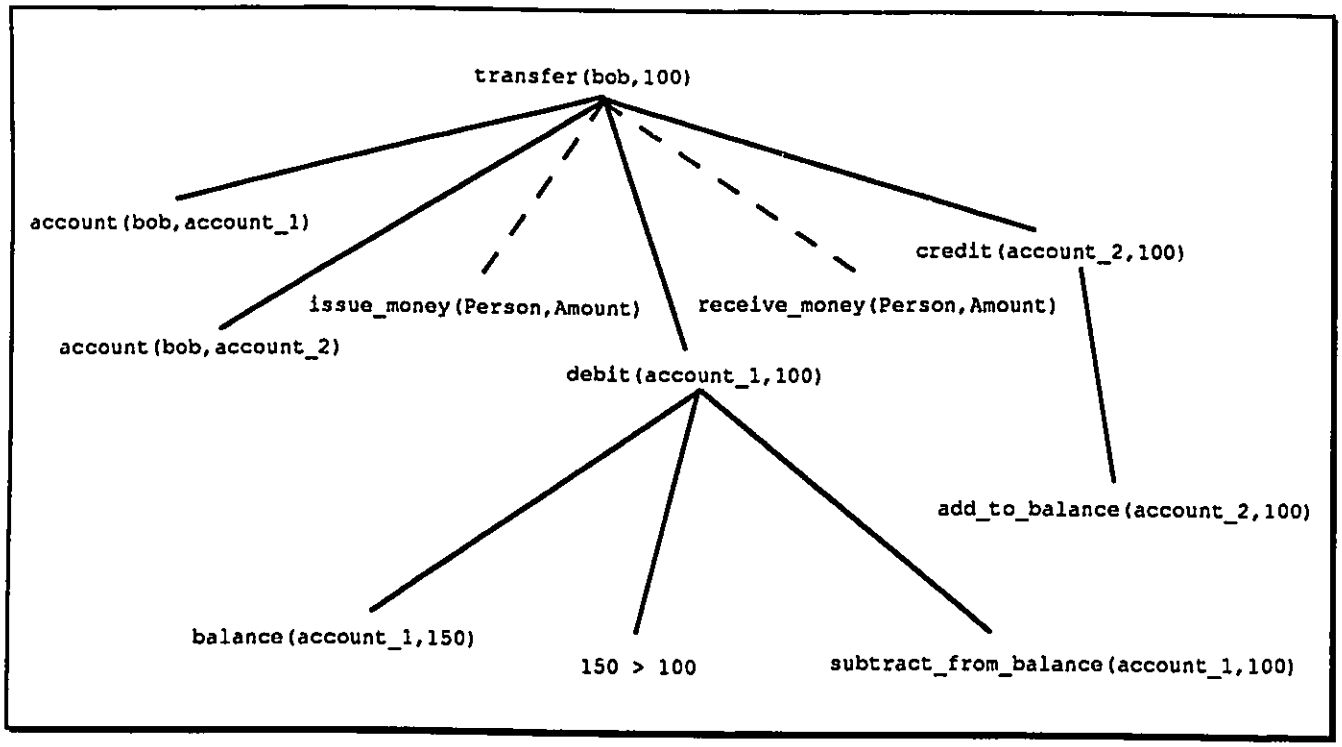

Fig 4.20 The explanation obtained by combining withdraw and deposit

The plausible explanation obtained after the combination (fig 4.20) still contains two unproven antecedents: `issue_money(Person,Account)` and `receive_money(Person,Account)`. LARS can remove both these actions because their goals are complementary. In other words, the analysis of the goals reveals that if the bank issues `100` dollars (i.e. the value to which `amount` is unified in the explanation) to a customer and receives `100` dollars back from the customer within the same transaction, then both actions can simultaneously be removed.

After having removed the two unproven antecedents from the explanation tree of fig 4.20, LARS obtains the plausible explanation of fig 4.21.

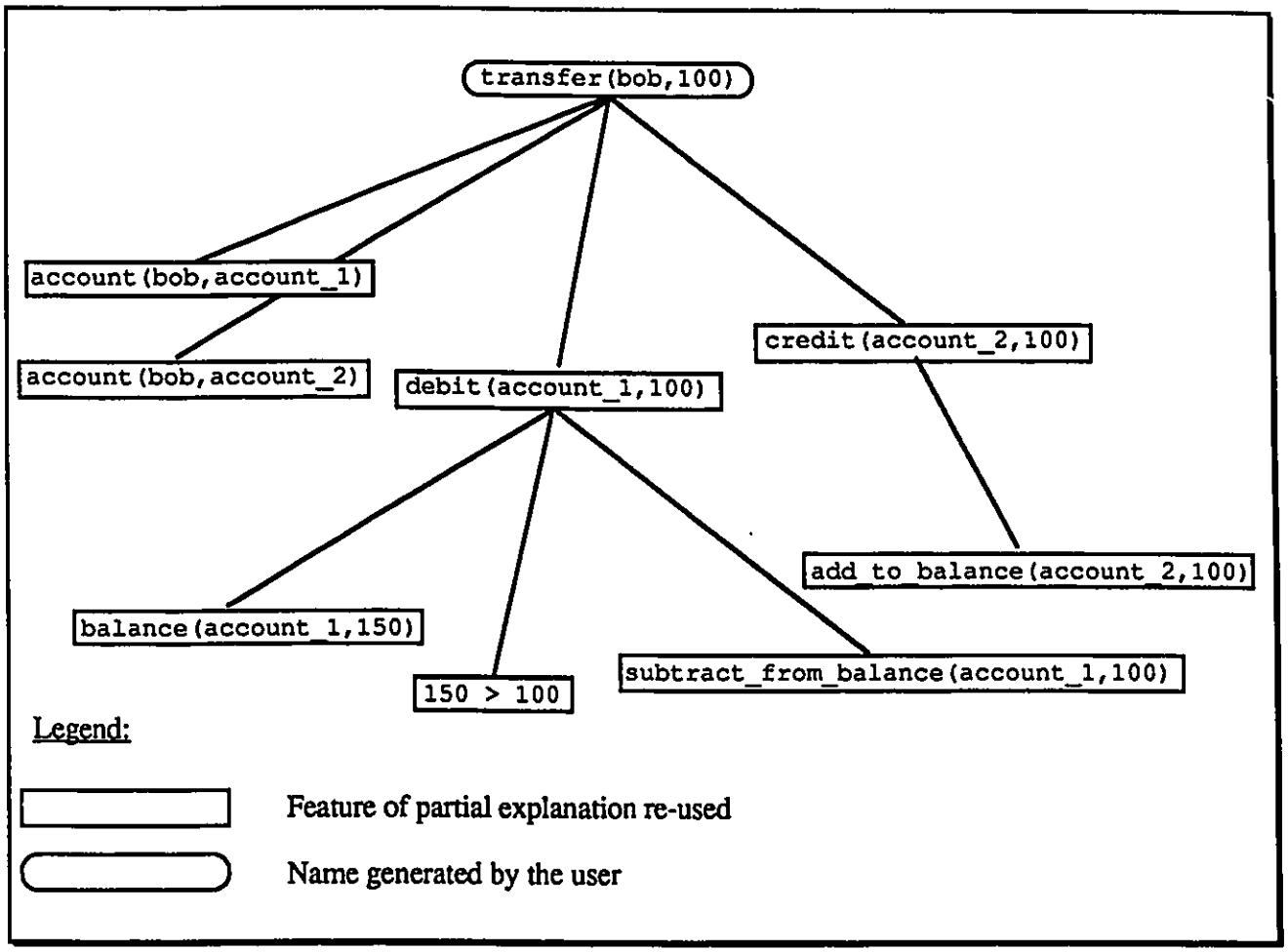

Fig. 4.21 The plausible explanation for `transfer`

After the plausible explanation of figure 4.21 has been produced, the constants must be generalized to variables. Then, the new frames must be synthesized from the explanation tree and added in the domain theory. In this example, the only frame extracted from the plausible explanation is the frame of `transfer(Person,Amount)` (see fig 4.22) because the subtrees of `debit(Account,Amount)` and `credit(Account,Amount)` were re-used from the partial explanation of `withdraw` and `deposit` respectively.

```
Name: transfer(Person,Amount)
 Parents: [transaction(Person)]
   precondition:
     account(Person,Account1)
     account(Person,Account2)
   procedure:
     debit(Account1,Amount)
     credit(Account2,Amount)
```

Fig. 4.22 The new frame for `transfer` added to the domain theory.

4.4.5 Case-based reasoning for LISE (CARL)

The domain theory in LISE contains rules which normally enable EBL to explain many positive training examples. However, some instances of concept are not readily explainable using these rules. These instances may be considered as exceptions to the general rules. They correspond to exceptions and nearly representative examples [Berdagano et al. 1988]. In the inductive setting, such instances are usually covered by the small disjuncts in [Holte et al. 1989]. In the EBG setting, they are referred to as marginals [Matwin et al. 1990]. In our method, a case in the case-base is an extension of the concept definition of the domain theory. It may be used to define instances of concepts which would be too complex to describe using rules in the domain theory. The module of LISE which relies on case-based reasoning is called CARL (Case-based reasoning for LISE).

CARL will attempt to apply a case to a training example when EBL does not work, abduction fails to complete a partial explanation, and analogical reasoning does not permit to replace all the unproven antecedents by training example facts.

The cases in the case-base represent TRANSACTIONs as defined in section 4.3.4. As for the TRANSACTION of the domain theory, they are composed of a precondition and a procedure containing predicates. The predicates found in the precondition and the procedure are called case features. A case will be retrieve for a training example if there is a match between the case features and the training example facts and if the order of the case features is preserved in the training example. There is a match between a case feature and a

training example fact when the name of the case feature (i.e. property name or operation name) is the same as the training example fact name and when the variables in the case feature unify with the constants of the training example fact. The unification is propagated to the other case features. There will also be a match between a case feature and a training example fact when the case feature contains constants that are associated with the same constants in the training example fact. Finally, when the case feature name is not the same as the training example fact name, there may be a match if there is a fact in the training example analogous to the case feature.

There may be more than one case selected for a training example. When this happens, a heuristic similar to the one described in section 3.4.1 will select the case providing the best match. The case providing the best match will be the one leaving the minimum amount of irrelevant facts in the training example.

The training example `get_a_mortgage(bob,95000)` of figure 4.23 will be used to illustrate how CARL applies a case to a training example. The training example `get_a_mortgage` represents an activity where a person named `bob` gets a mortgage for `95000` dollars. The sequence of facts in the training example are that `bob` owns an account identified as `account_1`. Next, `credit_margin(bob,3500)` tells us that `bob` has a credit margin of `3500` dollars. The next fact says that `bob` owns some real-estate valued at `99000` dollars. The monthly income of `bob` is `3300` dollars according the the fourth fact. The fifth fact mentions that, for a `95000` dollar mortgage, with a rate of `12.0` percent and a term of `25` years, the installment is `980.31` dollars. The sixth fact is that a loan of `95000` is recorded at the bank and the last fact is that `bob` is issued the amount of `95000` dollars.

```
The training example is: get_a_mortgage(bob,95000)
The facts are:
  account(bob,account_1).
  credit_margin(bob,3500)
  real_estate(bob,property_1)
  value(property_1,99000)
  monthly_income(bob,3000)
  installment(95000,12.0,25,800)
  record_loan(bob,95000)
  issue_money(bob,95000)
```

Fig 4.23 The training example get_a_mortgage(Person,Amount)

The first step in the system LISE is to use the module ELLI to attempt to explain the training example with EBL. Considering that the domain theory of figure 4.3 is used, two partial explanations are produced because the domain theory does not include the specification of `get_a_mortgage`.

ELLI starts by attempting to complete the partial explanations obtained for `get_a_mortgage` using abduction. The attempt fails because the domain theory contains no rules which, in conjunction with training example facts, can abductively transform the partial explanation into a plausible one.

The second attempt to build a plausible explanation from the partial explanations is made by LARS. In this case, LARS fails to build a plausible explanation because the domain theory does not contain all the goals necessary to analogically replace each unproven antecedent of a partial explanation by an analogous feature from the training example. It is interesting to note that if the `credit_margin` of `bob` was greater that `95000` dollars, LARS would have been able to build a plausible explanation as it did for the `borrow` transaction (the only difference between `get_a_mortgage` and `borrow` is that the amount the client wants to `borrow` exceeds his/her `credit_margin`, the reader might convince himself by comparing the training examples of fig 4.9. and fig. 4.23.). The resulting specification would have been a valid one because, according to the domain theory, when someone as an appropriate `credit_margin`, there is no necessity to verify his/her salary and assets.

```
case: loan_to_country(Country,Amount),
  isa: transaction(Country)
  precondition:
    account(Country,Account)
    gold_reserve(Country,Gold_reserve)
    value(Gold_reserve,Value_of_gold_reserve)
    Value_of_gold_reserve > Amount
    monthly_gnp(Country,Monthly_GNP)
    installment(Amount,Rate,Term,Installment)
    Affordable_installment is Monthly_GNP * 0.3
    Installment < Affordable_installment])
  procedure:
    record_loan(Country,Amount)
    issue_money(Country,Amount)
```

Fig 4.24 The previous case `loan_to_country(Country,Amount)`

As no plausible explanation is obtained using ELLI and LARS, the training example is sent to the module CARL. Suppose that the case-base at this point contains the case `loan_to_country` described on figure 4.24. As for the TRANSACTIONs of the domain theory, this case has a precondition and a procedure. The first feature of the precondition requires that the country asking for a loan has an account. The second, third and fourth features demand that the country has gold reserves, and that the value of the goal reserves exceeds the Amount requested for the loan. The fifth feature of the precondition is the monthly Gross National Product (GNP) which, in this case, plays the role of the country's revenue. The sixth precondition feature determines the installments for the requested loan. The two last features of the precondition require that the installment for the loan be less than 30 percent of the country's `monthly_gnp`. The two features of the procedure say that the loan must be recorded and only then the money is issued to the country.

CARL will attempt to apply the case `loan_to_country` to the training example `get_a_mortgage`. Figure 4.25 illustrates the output produced by CARL as it retrieves and adapts the case to the training example in order to build the

```
Building a new spec using case:
loan_to_country(Country,Amount)

Feature of TE is proven antecedent of partial_explanation :
: account(bob,account_1)

Training example fact adapted by analogy:
Training example fact: real_estate(bob,property_1)
case feature: gold_reserve(Country,Gold_reserve)
Goal shared: assets

Common feature kept: value(property_1,99000)

Constraint from case kept: Real_estate_value > Amount

Training example fact adapted by analogy:
case feature :monthly_gnp(Country,Monthly_gnp)
Training example feature :monthly_income(bob,3300)
Goal shared:monthly_revenue

Common feature kept:
installment(Principal,Int_rate,Duration,Installment)

Constraint from case kept :
Affordable_installment is Monthly_income * 0.3

Constraint from case kept :
Installment < Affordable_installment

Common feature kept: issue money(Person,Amount)
```

Fig 4.25 Adaptation of the case `loan_to_country` to the training example of `get_a_mortgage`

specification of `get_a_mortgage`. The first message indicates that the case `loan_to_country` is used. The next message is that the case feature `account(bob,account_1)` is kept in the new specification of `get_a_morgage`. The next message says that the case feature `gold_reserve(Country,Gold_reserve)` is replaced by the analogous training example fact `real_estate(bob,property_1)`. The goal common to both predicates is that they are both used to identify assets of an individual. The next messages say that the feature `value(property_1,99000)` and `99000 > 95000` are both kept in the new specification. The sixth message says that the case feature `monthly_gnp(Country,Monthly_gnp)` was replaced by the analogous training example fact `monthly_income(bob,3300)` because the goal of both is `monthly_revenue`. The four last messages say that some common features between the case and the training example were integrated in the new specification.

The resulting specification of `get_a_mortgage(Person,Amount)` is shown in fig 4.26. CARL determined that the training example fact `credit_margin(bob,3500)` was irrelevant and did not integrate it in the specification.

```
The new transaction frame is:

Name: get_a_mortgage(Person,Amount)
 Parents: [transaction(Person)]
   precondition:
     account(Person,Account)
     real_estate(Person,Real_estate)
     value(Real_estate,Real_estate_value),
     Real_estate_value > Amount
     monthly_income(Person,Monthly_income)
     installment(Principal,Int_rate,Duration,Installment)
     Affordable_installment is Monthly_income * 0.3
     Installment < Affordable_installment
   procedure:
     grant_loan(Person,Amount)
     issue_money(Person,Amount)
```

Fig 4.26 The result of case-based applied to `get_a_mortgage`

The new specification produced by CARL is in the format used by the domain theory and can be directly integrated without any further changed. The next time the training example of fig. 4.23 is presented, a complete explanation will be obtained.

4.5 Another example: the trucking domain

To show that LISE is domain independent, we shall present another example. The specification in this example is for a subset of the Base Automated Transport Operation System (BATOPS) which is a fleet management system developed by the Canadian Armed Forces to computerize transport operations on all the bases across Canada and abroad.

Another goal of this part is to show the user interaction between LISE and the user.

The domain theory used as a starting point contains one transaction and two actions (figure 4.27). The transaction `hire_a_vehicle` is used when a client needs a car for a certain period. The precondition of the transaction is to authorize the request of the client. The procedure is to:

assign a vehicle (ACTION `assign_a_vehicle`),
pick-up the vehicle (primitive operation),
return the vehicle (primitive operation),
prepare the bill for vehicle(ACTION `prepare_vehicle_bill`) , and,
send the bill to the client (primitive operation).

In that transaction, there are two non-primitive actions. The first non-primitive action, `assign_a_vehicle`, is defined in the second frame of figure 4.27. Its precondition is to find an available vehicle (primitive operation `available_vehicle`) and its procedure is `book_the_vehicle`. The last frame of fig 4.27 is the frame of the non-primitive operation `prepare_vehicle_bill`. The precondition is `get_meter_reading` and the procedure is `calculate_vehicle_cost`.

Most non-primitive and primitive actions in the domain theory for BATOPS have goals. The notation used for the goals in the domain theory is the two-ary predicate `goal`. The general form is `goal(operationX,goalX)` and should be read the goal of `operationX` is `goalX`. These goals will be used extensively in our example.

```
hire_a_vehicle(Client,Date,Time_start,Time_end)
 isa transaction(Client)
 precondition:
     authorize_request(Client)
  procedure:
     assign_a_vehicle(Date,Time_start,Time_end,Vehicle)
     goal(assign_a_vehicle,assign_resources)
     vehicle_pickup(Client,Date,Time_start,Vehicle)
     goal(vehicle_pickup,resource_issue)
     vehicle_return(Client,Date,Time_end,Vehicle)
     goal(vehicle_return,resource_return)
     prepare_vehicle_bill(Vehicle,Bill)
     send_bill(Client,Bill)

assign_a_vehicle(Date,Time_start,Time_end,Vehicle)
 isa:action
 precondition:
     available_vehicle(Date,Time_start,Time_end,Vehicle)
     goal(available_vehicle,ensure_resource_availability)
  procedure:
     book_vehicle(Date,Time_start,Time_end,Vehicle)
     goal(book_vehicle,record_resource_commitment)

prepare_vehicle_bill(Vehicle,Bill)
 isa:action
 precondition:
     get_meter_reading(Vehicle,Meter_reading)
     goal(get_meter_reading,record_usage)
 procedure:
     calculate_vehicle_cost(Vehicle,Meter_reading,Bill)
     goal(calculate_vehicle_cost,calculate_cost)
```

Fig 4.27 The domain theory for the trucking example.

The training example presented to the system is shown in fig 4.28. It represents an activity where a certain individual, capt_genest, hires a driver from 1100

hours to 1600 hours on the day 89200. The training example shows that the request of `capt_genest` was authorized. `driver_1` was found available for the period requested and was booked for the task. The driver was sent on the start time to accomplish the task and returned on the end time. The operation `get_driver_time_sheet` was used to obtain the data required for the next operation, `calculate_driver_cost`. The last operation was to send the bill to the client.

```
training_example(hire_a_driver(capt_genest,89200,1100,1600))

authorize_request(capt_genest)
available_driver(89200,1100,1600,driver_1)
book_driver(89200,1100,1600,driver_1)
send_driver(capt_genest,89200,1100,driver_1)
driver_return(capt_genest,89200,1600,driver_1)
get_timesheet(driver_1,time_sheet)
prepare_driver_bill(driver_1,time_sheet,bill_1)
send_bill(capt_genest,bill_1)
```

Fig 4.28 The training example for hire_a_driver

The training example `hire_a_driver` was given as input to LISE. The domain theory at that moment was the one illustrated on fig 4.27. The domain theory is incomplete with regard to `hire_a_driver`, thus LISE produces a partial explanation of `hire_a_driver` using the specification of `hire_a_vehicle`. Figure 4.29 shows the output produced by LISE when the training example is processed.

The partial explanation has two proven antecedents `send_bill(Client,Bill)` and `authorize_request(Client)`. The proven antecedents in figure 4.29 contain variables that were copied from the specification of the domain theory used to build the partial explanation. It does not mean that LISE has generalized the proven antecedent yet. At this point, LISE could also have displayed the proven antecedents as `send_bill(capt_genest,bill_1)` and `authorize_request(capt_genest)`. It was a design decision to display the variables.

There are also five unproven antecedents in the partial explanation and five unused facts in the training example. The module ELLI has attempted to apply abduction to complete the training example to no avail.

```
NO successful explanation; Partial explanation(s) follow:
-------------------------------------------------------

PARTIALLY explained using transaction:
hire_a_vehicle(Client,Date,Time_start,Time_end)

Common features:
[send_bill(Client,Bill),authorize_request(Client)]

Unexplained features of hire_a_vehicle(Client,Date,Time_start,Time_end):
[book_vehicle(Date,Time_start,Time_end,Vehicle),
available_vehicle(Date,Time_start,Time_end,Vehicle),
vehicle_pickup(Client,Date,Time_start,Vehicle),
vehicle_return(Client,Date,Time_end,Vehicle),
calculate_vehicle_cost(Vehicle,Meter_reading,Bill)]

Features of the training example not used in explanation :
[available_driver(89200.0,1100,1600,driver_1),
book_driver(89200.0,1100,1600,driver_1),
send_driver(capt_genest,89200.0,1100,driver_1),
driver_return(capt_genest,89200.0,1600,driver_1),
calculate_driver_cost(driver_1,time_sheet,bill_1)]

Hit any key to continue...

 *** MESSAGE from LISE ***
Do you want to build a new transaction
from the partial explanations? (y/n)

There was 1 partial explanation found -- no ranking will be performed
```

Fig 4.29 The output displayed by LISE for the partial explanation

As indicated by the first message of LISE (fig 4.30a), the module LARS will attempt to construct a plausible explanation for the training example of `hire_a_driver`

using the partial explanation `hire_a_vehicle`. The first antecedent of the partial explanation is `authorize_request`. It will be re-used since it is a proven antecedent. The second antecedent, `available_driver`, is unproven. It is replaced by the analogous training example fact, `available_vehicle`, since they share the goal `ensure_resoure_availability`. The next unproven antecedent, `book_driver`, is also replaced by analogy. The following message is a request for a new name because LARS needs to create the specification of a new action which will replace the action `assign_a_vehicle` in the plausible explanation. The boldface characters show that the user entered `assign_a_driver` as a name for the action. Similarly, the user was asked to provide a new name for another new action to be included in the plausible explanation. The user chose the name `prepare_driver_bill` (figure 4.30b).

```
Building a new frame using:
hire_a_vehicle(Client,Date,Time_start,Time_end)

Feature of TE is proven antecedent of partial_explanation :
authorize_request(Client)

Training example fact adapted by analogy:
Training example fact:
available_driver(89200,1100,1600,driver_1)
Unproven antecedent:
available_vehicle(Date,Time_start,Time_end,Vehicle)
Goal shared: ensure_resource_availability

Training example fact adapted by analogy:
Training example fact: book_driver(89200,1100,1600,driver_1)
Unproven antecedent:
book_vehicle(Date,Time_start,Time_end,Vehicle)
Goal shared: record_resource_commitment

A new action was found, its properties are:
precondition:
      available_driver(89200,1100,1600,driver_1)
procedure:
      book_driver(89200,1100,1600,driver_1)
Please enter a name for the new action without parameters:
assign_a_driver
Training example fact adapted by analogy:
Training example fact:
send_driver(capt_genest,89200.0,1100,driver_1)
Unproven antecedent:
vehicle_pickup(Client,Date,Time_start,Vehicle)
Goal shared: resource_issue
```

Fig 4.30a LISE building the plausible explanation for `hire_a_driver`

```
Training example fact adapted by analogy:
Training example fact:
driver_return(capt_genest,89200.0,1600,driver_1)
Unproven antecedent:
vehicle_return(Client,Date,Time_end,Vehicle)
Goal shared: resource_returned

Training example fact adapted by analogy:
Training example fact: get_timesheet(driver_1,time_sheet)
Unproven antecedent:
get_meter_reading(Vehicle,Meter_reading)
Goal shared: record_usage

Training example fact adapted by analogy:
Training example fact:
calculate_driver_cost(driver_1,bill_1)
Unproven antecedent: calculate_vehicle_cost(Vehicle,Bill)
Goal shared: calculate_cost

A new action was found, its properties are:
precondition:
      get_timesheet(driver_1,timesheet)
procedure:
      calculate_driver_cost(driver_1,timesheet,bill_1)
Please enter a name for the new action without parameters:
prepare_driver_bill

Feature of TE is proven antecedent of partial_explanation :
send_bill(Client,Bill)

Hit any key to continue...
```

Fig 4.30b Continuation of LISE building the plausible explanation for `hire_a_driver`

After the plausible explanation is built, LARS generalizes by turning the constants into variables. The result of learning `hire_a_driver` presented to the user is a set of three specifications (fig 4.31) for `assign_a_driver`, `prepare_driver_bill` and `hire_a_driver`. These new frames will be added to the incomplete domain theory.

```
A new action frame was created:

Name: assign_a_driver(Date,Time_start,Time_end,Driver)
 isa: [action]
   precondition:
         available_driver(Date,Time_start,Time_end,Driver)
   procedure:
         book_driver(Date,Time_start,Time_end,Driver)

A new action frame was created:

Name: prepare_driver_bill(Driver,Bill)
 isa: [action]
   precondition:
         get_timesheet(Driver,Timesheet)
    procedure:
         calculate_driver_cost(Driver,Timesheet,Bill)

The new transaction frame is:

Name: hire_a_driver(Client,Date,Time_start,Time_end)
 isa: [transaction]
   precondition:
         authorize_request(Client)
   procedure:
         assign_a_driver(Date,Time_start,Time_end,Driver)
         send_driver(Client,Date,Time_start,Driver)
         driver_return(Client,Date,Time_end,Driver)
         prepare_driver_bill(Driver,Bill)
         send_bill(Client,Bill)
```

Fig 4.31 The new frames obtained for `hire_a_driver`

The main component of LISE used in this demonstration was LARS. The specification of a new transaction, `hire_a_driver`, was learned using analogical reasoning applied to the already known specification of `hire_a_vehicle`. Two actions were also learned analogically. `assign_a_driver` was learned analogically from `assign_a_vehicle` and `prepare_driver_bill` was learned analogically from `prepare_vehicle_bill`.

This section showed a second example of LISE applied to the domain of fleet management. The intention was not only to show the type of interaction between LISE and its user, but also to show that LISE is truly domain independent.

Chapter 5

Complexity analysis of LISE

One of the challenges in Computer Science in general and in Machine Learning in particular is to develop approaches to current problems that are not too expensive in time and in computing resources. Our approach was developed with this criterion as a priority. This chapter will present an analysis of the complexity of the three modules of LISE and of LISE as a whole.

5.1 Analysis of the complexity of ELLI

The first process applied by ELLI is EBL. Being a backward chaining, goal reduction problem solver, EBL's cost depends on the depth to which the search is allowed. In our system, a level of depth 1 corresponds to chaining from the goal concept to the antecedents of the rule having the goal concept as its consequent. If additional chaining is required from the antecedents of that rule, the next level will be depth 2 (see figure 5.1).

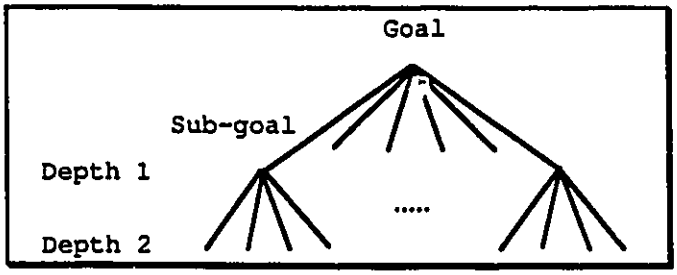

Figure 5.1 Search space of LISE

In our domain theory, the specifications of the TRANSACTIONs and ACTIONs are transformed into rules. These rules have the TRANSACTIONs or ACTIONs names as consequents and the contents of the precondition and the procedure as antecedents.

EBL is always started with a TRANSACTION name as the goal concept. Consequently, a search at depth 1 allows us to chain from the TRANSACTION name

(consequent) to the ACTIONs used in its specification (antecedents). For the antecedents representing primitive actions, there will be no more chaining. Each antecedent will be proven if there is a corresponding training example fact. For the antecedents representing non-primitive actions, one additional chaining will be required from the action name to the content of its precondition and the procedure, which in turn, contains only primitive actions. Thus, the non-primitive actions will require a search of depth 2. At depth 2, if an antecedent does not correspond to a training example fact, then the antecedent is flagged as an unproven antecedent and the explanation being built will be a partial explanation.

The number of partial explanations we obtain for a training example containing n facts is p where p is the number of transactions defined in the domain theory. This is because we can only use the definition of the transactions contained in the domain theory to partially explain a training example. Even though the number of partial explanations processed using abduction and analogical reasoning will be p-e where e is the number of empty partial explanations, p will be used for the rest of our analysis.

The cost involved in ELLI to build a set of one or many partial explanations corresponds to traversing p (not p-u) trees to a maximum level of depth-2. This is done in order to mark the proven and unproven antecedents. A lower cost will be obtained only if ELLI encounters a complete explanation - an explanation made up of proven antecedents exclusively. In that case, ELLI will stop searching and will deliver the complete explanation.

At this point, if no complete explanation is produced, the set of p partial explanations needs to be scored and ranked using our heuristic. The scoring, done first, costs a unit for each partial explanation. The ranking, coming next, is performed using quicksort. The total cost of our heuristic, which is applied only once, is `p + p log p`.

To evaluate the cost of abduction, let us now suppose that the partial explanations contain an average of u unproven antecedents. The aim of abduction is to search for a rule in the domain theory having, as its own antecedent, an unproven antecedent of the partial explanation, and having, as its consequent, a training example fact. Suppose that there are r rules of that kind, the cost of abduction will be at most `r*u`. The cost of abduction is minimized because the application of the rules is limited to one chaining.

In the worst case, abduction may work only with the last partial explanation. Thus, the cost of the module ELLI is, at worst, `O(p + p log p + (p*r*u))` or

```
O(p(log p + r*u)).
```

5.2 Analysis of the complexity of LARS

The second module LARS attempts to replace the unproven antecedents of a partial explanation by analogous training example facts. The search for an analogous training example fact is made as following: from the unproven antecedent, we obtain the goal directly, and from the goal, we have to search in the training example for an analogous fact. We do not allow multiple goals for one predicate.

Recalling that there are `p` partial explanations, `n` training example facts, and an average of `u` unproven antecedents in the training example, the cost of LARS is, at worst:

```
p*u*n.
```

5.3 Analysis of the complexity of CARL

The third module, CARL, will search through a case-base for a case which can be applied to the training example. Supposing that there are `c` cases in the case-base having an average of `f` features, the cost of matching the cases to the training example is, at worst:

```
c*f*n.
```

If more than one case matches the training example, our heuristic is used to score the match and extract the best one. The heuristic costs `c + c log c`.

The total cost of CARL is at worst:

```
O(c(log c + f*n)).
```

This cost is the most important so far since there may be a rather large number of cases c, which in turn can have a moderately large number of features f. However, if we consider that the cost of matching cases to training example is an NP-complete problem (because of the combinatorial explosion), the polynomial cost of our variant of case-based reasoning is very reasonable. The polynomial cost is achieved by imposing that the order of the case features be respected by the training example facts.

5.4 Complexity of LISE as a whole

The cost of LISE described in this analysis is the pessimistic cost. It depends on the quality of the domain theory. A good domain theory will produce a fair number of partial explanations (medium p) containing only a few unproven antecedents (small u). A domain theory containing a good amount of background knowledge - or goals - makes the odds that analogical reasoning produces a plausible explanation very good. This is desirable since it eliminates the need to use the case-based system.

The overall cost of LISE will be, at worst:

```
O(p(log p + u(r+n) + 1)+ c(log c + f*n + 1)).
```

where:

- p: number of partial explanations,
- u: number of unproven antecedents (average),
- r: number of rules in the domain theory,
- n: number of training example facts,
- c: number of cases, and,
- f: number of case features (average).

The analysis depicted here does not contain any costs related to the generalization of the plausible explanations accomplished by turning constants into variables. The reason is that the EBL algorithm used in the module ELLI builds a generalized version of each partial explanation in the initial pass discussed on section 5.1. When a proven antecedent is re-used, the variables are kept. When an unproven antecedent is analogically substituted by a training example fact, the required variables are immediately determined.

Chapter 6

Comparison with related work

The research presented in this thesis concerns three distinct domains of computer science. First, it addresses an important Machine Learning issue: the incomplete theory problem. Second, it is an non-trivial application of Artificial Intelligence (AI) to the design of software systems. Third, it can be regarded as a programming-by-example tool. In this chapter, we will compare the results of our research with related work along the three distinct domains of interest.

Section 6.1 will compare LISE with two different approaches to deal with the incomplete theory. In section 6.2, we will compare LISE with two systems applying AI to the design of software systems. Section 6.3 will provide a comparison with the system Query-by-example, a database system driven by examples.

6.1 Related work in the incomplete theory area

[Ellman 1989] divides the methods to handle incomplete domain theories based on partial explanations into analytical methods and empirical methods. Both these methods generate partial explanations by explaining the training examples as much as they can using the incomplete theory. Rules are conjectured to fill the holes in the explanations. Analytical methods use background knowledge to validate and refine the conjectured rules whereas the empirical methods use multiple training examples to validate and refine the conjectured rules.

According to [Ellman 1989]'s division, our approach is analytical because of our usage of background knowledge - the rules in abduction, the goal hierarchy in LARS and the cases in CARL - in order to construct plausible explanations of our training examples.

In this section, we will compare our work with an analytical and an empirical approach. The analytical approach, compared in section 6.1.1, is called *Learning by failing to explain* [Hall 1988]. Section 6.1.2 will compare our work with an empirical approach called *Learning from plausible explanations* [Fawcett 1989].

Comparing our approach to [Dietterich et al. 1988]'s Induction Over Explanation (IOE) and to the most recent [Mooney et al. 1989]'s Induction Over the Unexplained (IOU) will only be done briefly because both these systems address the problem of inconsistent theories (see section 3.2) whereas our work addresses the incomplete theory problem.

A theory is inconsistent when several mutually incompatible explanations are produced for the same training example. [Dietterich et al. 1988]'s IOE system repairs the inconsistent theories by specializing the domain theory so that a unique explanation is produced for each training example. The specialization process is purely syntactic since it involves removing disjuncts, replacing many variables by unique ones, and replacing constants by variables.

[Mooney et al. 1989]'s IOU deals with the inconsistent theory using an inductive learning component. An inconsistent theory produces overly-general concept definitions. An overly-general concept definition includes negative examples of the concept. IOU uses induction to add a conjunction to the concept definition. The conjunction is intended to specialize the concept definition and eliminate its coverage of the negative examples.

6.1.1 Our approach vs. Learning by failing to explain

[Hall 1988] describes an analytical approach to deal with an incomplete domain theory which was applied to the design of digital circuits. Similarly to our system, Hall 's system is an application of Machine Learning to design. The approach, called *precedent analysis*, was implemented in a system called PA. A precedent is composed of two similar objects. Here, the two similar objects are digital circuits having the same functionality (i.e. producing the same output for the same inputs), but with slightly different topologies.

The digital circuits are built upon low-level modules such as AND gates, OR gates, XOR gates, delays, and other modules defined by a design grammar. The system PA builds a partial match between the two circuits by matching similar modules and by matching modules which are equivalent according to the design grammar. After the partial parse is built, some modules which can not be matched according to the design grammar, end up being isolated at the same location of the circuits. New rules are then conjectured to the effect that these corresponding modules have the same functionality.

6.1.1.1 An example of PA

Figure 6.1 is a simple example of a session with PA where two digital circuits having the same functionality F (for each set of values that X, Y and Z can take, the same value of R is obtained on each circuit) constitute the precedent. Figure 6.2 shows a rule in the design grammar stating that two modules of type C and D in sequence are equivalent to one module of type E. The partial match is built as following. The modules A of circuit 1 can be matched to module A of circuit 2 because they are located at the same place (they have X and Y as inputs and α as output). According to the design grammar, the sequence of modules C and D is equivalent to the module E and they match because they are located at the same place, with β as input and R as output. No more matching is possible and there are two subcircuits left in both circuits with the same inputs and the same outputs. Thus, a new design rule (figure 6.3) is created to specify that module B is equivalent to module W.

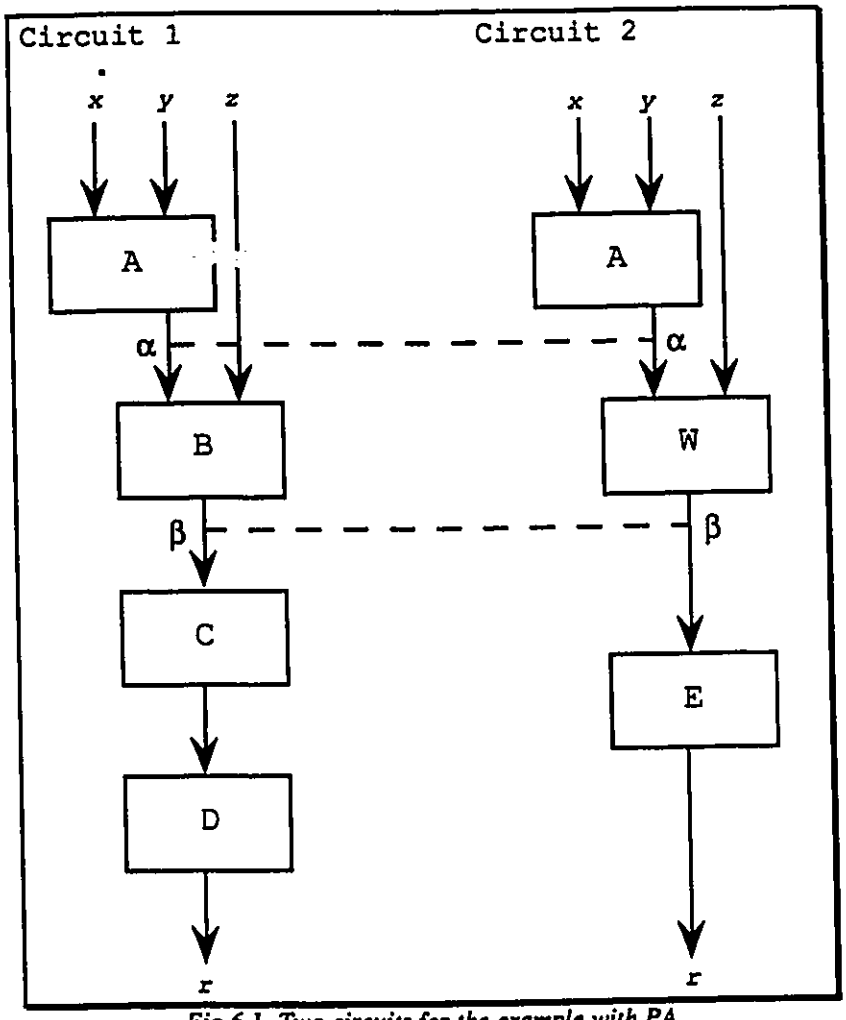

Fig 6.1 Two circuits for the example with PA

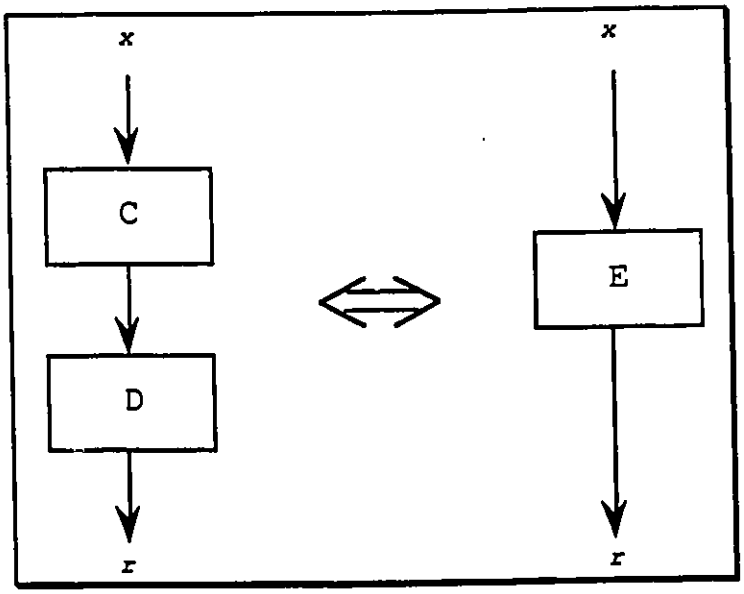

Fig 6.2 The grammar rule used in PA

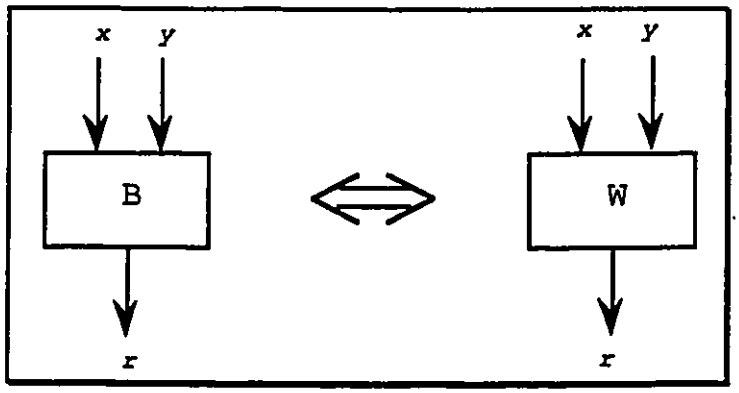

Fig. 6.3 The new design grammar rule learned by PA

After each new rule is built, PA applies a process called *rule reanalysis* which ensures that no overly-specific rules are produced by the system. Ideally, the difference between two circuits in a precedent should be attributable to only one unknown rule (this corresponds to the idea of a near-miss in [Winston 1984]). However, PA can accept precedents where the difference between two circuits are caused by the application of more than one rule ([Hall 1988] calls these cases far-miss). In that case, PA would synthesize a unique overly-specific new rule. The rule reanalysis approach consists in applying precedent analysis to the new overly-specific rules in order to extract more general rules. It is more likely that a general rule be used in the design than an overly-specific rule.

6.1.1.2 Discussion

There are some similarities between our approach to deal with the incomplete theory and Hall 's. In [Hall 1988], partial parses are used to isolate unknown portions of training examples. In our work, the partial parses are replaced by partial explanations. In [Hall 1988], a unique partial parse of training example X is made with respect to a second training example Y. There is an assumption that X and Y are both available, and that X and Y are functionally equivalent. Because the functionality of a training example is unique in LISE, we must use the entire domain theory available to build partial explanations, and we must apply a heuristic to determine which one is the best. For the same reason, we can not conclude that the unproven part of the best partial explanation has the same functionality as the corresponding training example facts. Our assumption is that, although their functionality are different, the (relevant) training example facts should have the same goals as the unproven antecedents. This assumption is made on the basis that

the specification of all the transactions in a system should respect the same set of goals as determined by the organization.

In summary, the difference between our work and Hall's lies in the difference in the inputs of the learning task. The design tasks in [Hall 1988] require a design grammar, which is augmented as more precedents are presented. In our system, the design grammar counterpart is our domain theory.

6.1.2 Our approach vs. Learning from plausible explanations

In this section, we will compare our approach with the approach presented in [Fawcett 1989]. Fawcett proposes an empirical approach to augment a domain theory based on the usage of partial explanations. In this case, a partial explanation is quite similar to a partial explanation in LISE: it is an explanation containing both proven and unproven antecedents. A small difference is that Fawcett allows an explanation having no proven antecedents (i.e. the unproven goal concept by itself) to also be called a partial explanation.

6.1.2.1 Description of Fawcett's work

Fawcett's approach is to generate a set of one or many partial explanations. A heuristic is then used to select the best partial explanation, which he calls *most plausible explanation*. A rule is created for each unproven antecedent of the most plausible explanation. The rule consequent is the unproven antecedent and the rule antecedents are the training example facts. The rule creation is a form of abduction since there is an attempt to use known facts (training example features) as hypotheses for unknown facts (unproven antecedents). As we shall see later, our usage of abduction is different.

There may be many rules created from a single plausible explanation. These rules are, upon their creation, very specific. The aim is to use subsequent training examples to refine these rules. One type of refinement, not as yet implemented according to [Fawcett 1989], consists in eliminating antecedents which are not repeated in all the rules having the same consequent. For example, consider rule 1: ball(X) <- sphere(X) & red(X) and rule 2: ball(X) <- sphere(X) & green(X). Those rules were extracted from two plausible explanations built from two training examples. The refinement, which is a generalization, would transform these two rules into rule 3: ball(X) <- sphere(X).

6.1.2.2 Discussion

The generation of partial explanations in [Fawcett 1989] is similar to LISE's. Both systems use different heuristics in order to speed up the generation of partial explanations. Fawcett 's heuristic restricts the space of partial explanations by making some relations (e.g. is_a and part_of) mandatory (i.e. can not be left unproven) and by accepting no partial explanations containing unproven antecedents that are proven in other partial explanations. LISE's heuristic limits the search space by imposing that the partial explanations leaves be in the same order as the training example facts.

The heuristic used to select the best partial explanation is similar in both systems. In LISE, abduction can be used as an inference to contribute to a partial explanation by explaining some of its unproven antecedents. That heuristic is not used in [Fawcett 1989] because abduction is not used in the construction of an explanation. A criterion in Fawcett's heuristic that LISE does not support is to consider the depth of an unproven antecedent in the explanation tree as a measure of the quality of the partial explanation. We feel that, unless a great care was taken in arranging the rules so that the depth of a rule in an explanation tree relates to its generality, this heuristic might not be of a strong help.

The domain theory in [Fawcett 1989] starts with rules given by a domain expert. These rules have a high confidence value. The few first partial explanations are built using these rules. At some point, new rules, created abductively, start being used in the construction of the partial explanations. We do not think that these newly created rules should have the same weight as the rules given by the expert. Their weight should reflect both their confidence and their *usefulness*.

Before covering the differences between the usage of abduction in LISE and [Fawcett 1989], a brief recap of abduction is in order (for more details, see section 3.4.2). According to [Pople 1973], abduction is the generation of hypotheses which, if true, would explain observed facts. There should be a rule `Q <- P` and a fact `Q` in order to raise `P` as a hypothesis. In LISE, `Q` is a training example fact, `Q <- P` is a rule in the theory and `P` is the unproven antecedent that we need as a hypothesis.

In [Fawcett 1989], the "fact" `Q` is the unproven antecedent and the rule `Q <- P` does not exists. `P` is a training example fact. The rule `Q <- P` is conjectured on the

hypothesis that a relation exists between P and Q. In our view, that inference is more typical of induction since a rule is conjectured to explain a relationship between two events.

The main difference between our approach and Fawcett's system is the availability of a single training example versus multiple training examples. Although the requirement for a single training example seems more economical, our system requires that some background knowledge be available to resolve the partial explanation analytically. Nevertheless, the analytical adaptation of a partial explanation into a plausible one does not jeopardize the domain theory because no inductive leap occurs.

6.2 LISE vs related work in AI applied to Software Engineering

In this section, we will compare LISE to systems which apply AI techniques to the design of software systems. The first system, called LASR (Learning Apprentice for Software Reuse), not to be confused with our module LARS, uses explanation-based learning (EBL) in order to test the reuse potential of abstract data types. The second system, WATSON, transforms informal "scenarios" into formal specifications using a generalization process and theorem-proving.

6.2.1 LISE vs LASR (Learning Apprentice for Software Reuse)

LASR (A Learning Apprentice for Software Reuse) is an experimental learning system developed at Hewlett-Packard Laboratories [Hill 1988]. It uses explanation-based learning in order to test the reuse potential of abstract data types. When reuse is possible, LASR generalizes the interconnections between abstract data types in order to increase their reuse potential.

```
THEORY Stack

SORTS stack element

OPS empty: -> stack
      push: element stack -> stack
      pop: stack -> stack
      top: stack -> element
      empty?: stack -> boolean

VARS e: element
      s: stack

EQNS  pop(empty)= empty
      pop(push(e,s))= s
      top(push(e,s)= e
      empty?(empty)= true
      empty?(push(e,s))= false
```

Figure 6.4. Data Theory of Stacks

6.2.1.1 Description of LASR

In modern software engineering practice, formal theories of data abstraction play an important role in designing and validating software and in promoting reuse. These formal theories, also called *data theories*, play the role of domain theory for EBL. The data theories are equivalent to the concept of abstract data types. An example of the data theory of Stacks is illustrated in figure 6.4. As for the abstract data types, the data theories are divided into two parts, the SORTS part contains the syntactic information of the data type and the EQNS part contains the semantic information.

In order to apply data theories to the design of software, a mechanism called *theory morphism* is necessary. A *theory morphism* from a theory T1 to a theory T2 maps sorts and operations of T1, respectively, to sorts and operations of T2. After the morphism is applied, the axioms of T1 can be deduced in T2. In that scenario, T1 is called *source theory* and T2 is called *target theory*.

Figure 6.5 shows an instance of morphism from the theory of stacks to the theory of arrays. That morphism implements the stack data type using the array data structure.

```
MORPHISM Stack-with-Offset Stack => I-Array

SORTS (Stacks => i-array)

VARS e: element
  a: array
  n: nat

OPS     (empty => <new-array,0>)
  (push(e,<a,n>) => <assign(a,n+1,e),n+1>)
  (pop(<a,n>) => <a,n-1>)
  (top(<a,n>) => a[n])
  (empty?(<a,n>) => if n=0 then true else false)
```

Figure 6.5. Stack Implementation Morphism

The training example is the mapping M from theory T1 to theory T2. The domain theory is the data theory T2. The goal concept is a morphism M' from T1 to a subtheory T2' of T2. The operationality criterion is achieved by having the axioms defining the target theory expressed in terms of primitive operations of the data theory T2.

The explanation in LASR is a proof or a validation that the morphism M transforms the axioms of the source theory T1 into axioms in the target theory T2. Considering the morphism from the stack data theory (T1) to the array data theory (T2), the proof is:

```
assign(a,j,e)[i] = if i=j then e else a[i]
assign(a,n+1,e)[i] = a[i] if not(i=n+1)
assign(a,n+1,e)[i] = a[i] if i<=n
<assign(a,n+1,e),n> = <a,n>
pop(<assign(a,n+1,e),n+1>) = <a,n>
pop(push(e,<a,n>)) = <a,n>
```

Equational reasoning rules (reflexive, symmetric, and transitive laws, together with substitution and instantiation) are used to obtain the proof steps above. Equational reasoning is discussed in [Liskov et al. 1986].

The generalization process then consists in regressing the goal (in the example above, it is the axiom in the source theory: pop(push(e,<a,n>)) = <a,n>) through the explanation to eventually obtain a formula in the target theory. The reader is referred to [Hill 1988] for a more complete example of goal regression in that context.

6.2.1.2 Discussion

LISE and LASR share the same goal of promoting re-use of a specification. The specification in LARS are more general than LISE's because they can be used in any type of application and they are more specific that LISE's because they are defined using very low-level primitives.

The main difference between LISE and LASR is that LISE addresses the early analysis phase of software development whereas LASR is directed to the implementation phase (or programming phase) of software development. Both systems are complementary.

Finally, EBL in LISE is used as an operationalization process. In LASR, EBL is used as a theorem proven.

6.2.2 LISE vs WATSON

Watson is an automatic programming system that converts informal and incomplete requirements into formal and consistent specifications that can be executed.

The informal requirements are given to WATSON in the form of scenarios. A scenario is a description of the behavior of a system. The behavior of the system is a sequence of episodes. Figure 6.6 illustrates a scenario describing the behavior of a simple telephone service [Kelly et al. 1988].

```
First, Joe is on-hook,
and Joe goes off-hook,
then Joe starts getting dial-tone.
Next, Joe dials Bob;
then Bob starts ringing
and Joe starts calling Bob.
and Joe starts getting ringback.
Next, Bob goes off-hook,
then Joe gets connected to Bob.
Next Joe goes on-hook,
then Joe stops being connected to Bob.
```

Figure 6.6: The telephone service scenario

6.2.2.1 Description of WATSON

The procedure used by WATSON is the following. A scenario is given by the user and enters the *Scenario Generalizer*. The role of this module is to convert the scenario into a set of rules. These rules are then processed by the *Consistency and Completeness Analyzer* which repairs rule contradictions, eliminates unreachable or dead-end model states, infers missing rules and ascertains that all stimuli are handled in every states of the *world*. The analyzer requires interaction with the user to authorize changes and also with a set of domain axioms used to provide "common-sense" knowledge. In the domain axioms, we find domain independent and domain specific rules.

The scenarios are composed of episodes. Each episode consists of three parts:

a. the antecedents, which are assertions known to be true of the agents,

b. a stimulus, similar to an event, having implicit side-effects, and,

c. explicit consequents, which are assertions becoming true after the stimulus.

The scenario of figure 6.6 contains four episodes. Each episode is represented by one or many rules, depending on the number of consequents. The rules are expressed in a two-tense logic where the antecedents are in the present and the consequent in the immediate-future tense. Figure 6.7 shows the rules that were obtained from the scenario of figure 6.6.

```
R1.1: ∀ x [on_hook(x)  Λ  EVENT(goes_off_hook(x))
           ⊃  BEGINS(dial-tone(x))]
R2.1: ∀ x,y [dial-tone(x)  Λ  EVENT(dials(x,y))
           ⊃  BEGINS(calling(x,y))]
R2.2: ∀ x [∃y[dial-tone(x) Λ  EVENT(dials(x,y))]
           ⊃  BEGINS(ringback(x))]
R2.3: ∀x,y [dial-tone(x)  Λ  EVENT(dials(x,y))
           ⊃  BEGINS(ringing(y))]
R3.1: ∀x,y [calling(y,x)  Λ  ringback(y)  Λ  ringing(x)
                Λ EVENT(goes_off_hook(x))
           ⊃  BEGINS(connected(y,x))]
R4.1: ∀x,y [(connected(x,y)  Λ  EVENT(goes_on_hook(x))
           ⊃  ENDS(connected(x,y))
```

Figure 6.7 Rules for the telephone service episode

The Consistency and Completeness Analyzer identifies the anomalies present in the scenario. Some of these anomalies are:

a. antecedents may be omitted from episodes, resulting in over-generalized rules. An example of an over-general rule is R2.3. It is over-general because it does not consider what would happen if the callee, Bob, was already on the phone,

b. consequents may be missing, resulting in missing rules (remember that one rule is generated per consequent), for instance, there is no rule to state that Bob's phone stops ringing when going off-hook, and

c. irrelevant antecedents may be included resulting in under-generalized rules.

The types of corrections brought by Consistency and Completeness Analyzer are the following:

a. Repairing inconsistent rules. Rules are contradictory if different consequents can be deduced from compatible antecedents and stimuli.

b. Finishing incomplete episodes by adding new rules. In our scenario, we can see that Joe's dial-tone never ends. There should be an extra rule.

c. Eliminating unreachable "states". If the domain axioms express some states that are not obtained in the scenario when all the assertions are true, WATSON will ask for another scenario (e.g. R2.2 could also lead to a busy signal).

d. Ensuring that all stimuli are handled in all states. The domain axioms can sometimes help to determine the consequences of applying a stimulus in a situation not specifically covered by the scenario (e.g. if Joe hangs up during dial-tone). However, in more complex situation, WATSON might need the assistance of the user to select an alternative.

An example is presented where we will see WATSON repairing the inconsistency between R1.1 and R3.1. The problem is that R1.1 is too general. Rule R3.1 is a special case of R1.1 and its consequent is different from that of R1.1. We need to make R1.1 more specific to exclude all the cases covered by R3.1. The first step is to negate all the antecedents of R3.1 not in R1.1. The rule obtained is:

```
R1.1a: ∀ x [on_hook(x) ∧ [~ringing(x)
                    ∨  [∀y [~calling(y,x)
                         ∨ ~ringback(y)]]]
                  ∧  EVENT(goes_off_hook(x))]
            ⊃  BEGINS(dial-tone(x))]
```

That rule could be replaced by a much simpler one and WATSON identifies the following candidates:

```
R1.1b: ∀  x [on_hook(x)  ∧ ∀y [~ringback(y)]
                  ∧  EVENT(goes_off_hook(x))]
            ⊃  BEGINS(dial-tone(x))]

R1.1c: ∀  x [on_hook(x)  ∧ ∀y [~calling(y,x)]
                  ∧  EVENT(goes_off_hook(x))]
            ⊃  BEGINS(dial-tone(x))]

R1.1d: ∀  x [on_hook(x)  ∧ [~ringing(x)]
                  ∧  EVENT(goes_off_hook(x))]
```

```
⊃  BEGINS(dial-tone(x))]
```

Among the candidates, R1.1d is selected as the simplest because it introduces no new variables. The only danger is to select a rule which is not as general as R1.1a. That criteria is verified by establishing:

```
CWR :  ∀ x [ringing(x)
       ⊃  ∃y [calling(y,x)  Λ ringback(y)]]
```

where CWR is an expanded set of "closed word rules" established from the scenario used for reasoning. After the successful safety verification, WATSON asks the user to approve the replacement of R1.1 by R1.1d.

6.2.2.1 Discussion

The performance of theorem provers is typically slow. To improve that, the developers of WATSON integrated faster but approximate model-based reasoning. The models in that case are minimal automata implementing each type of agents. When WATSON needs to verify if some properties hold in a model, querying the model is 50 percent faster than using a theorem prover. The developers of WATSON have hard-coded the meta-reasoning needed to determine when some properties should be validated with the theorem prover or with the models. They are planning to automated the meta-reasoning for more flexibility based on the model.

In WATSON, the generate and test approach is used with no integration between the generation portion and the test portion. The training example scenario is transformed into a set of rules directly and a theorem prover is applied to the rules in order to verify its correctness and completeness. In LISE, the test is integrated into the generation process. By using the current (correct) domain theory to produce a partial explanation and to replace the unproven antecedents by analogous training example facts, WATSON ensures the correctness and completeness of the new rules according to the domain theory. Although LISE's domain theory as a whole is incomplete, the specifications of the operations it contains are individually correct and complete. Thus, each new specification deduced from the domain theory's specification is also correct and complete.

In contrast to LISE, WATSON does not have the equivalent of the operationality criterion which ensures that the resulting specification produced by the system is produced in a language useful for the implementation. However, WATSON produces an executable prototype of the specification. To change the language of the specification, without using the concept of the operationality criterion, must be difficult.

The scenarios presented to WATSON are presented in a restricted form of English. In LISE, unfortunately, the scenarios have to be presented in predicate form. It is a future goal to implement such an interface.

6.3 LISE vs Programming by Examples

In this section, we will compare LISE with the system Query-by-Example (QBE), a programming language designed to accept examples of queries provided by naive users [Zloof 1981].

A program in QBE is created by manipulating two-dimensional tables. These tables contain fields corresponding to the data elements of a database. A naive user can retrieve, modify, define and control the database with only basic training with QBE. QBE is part of a bigger system, called Office procedure By Example (OBE) which provides software support for functions such as word processing, electronic mail, report writing, and others, using the same principle of manipulating tables.

6.3.1 Example of QBE

Figure 6.8 illustrates a program created by a user to query a database. The query is: what is the amount of travel expenses of each employee working under manager LEE. To obtain that information, the user needs to extract information from two tables: EMP for the employee information and TRAVEL for the travelling information.

The user first enters the name of the field he needs to use in his query. The fields inserted in the table EMP are NAME and MGR and the fields inserted in the table TRAVEL are NAME, AMOUNT and DATE. Next, the user inserts in each column one of the following: a command if the user wants the information to be printed or displayed (e.g. P.), a constant if the user wishes to select only particular entries of the table (e.g. LEE), or example elements which act as variables (e.g. Smith).

In the example, the user entered LEE as a constant in the column MGR of the EMP table. The user also entered Smith in the NAME column. The underlining indicates that Smith is an example element. This element acts as a variable. The NAME column of the TRAVEL table receives P.Smith which is a cross-reference between the two tables. The P. in the three remaining columns of the TRAVEL table indicates that the user wants the information printed.

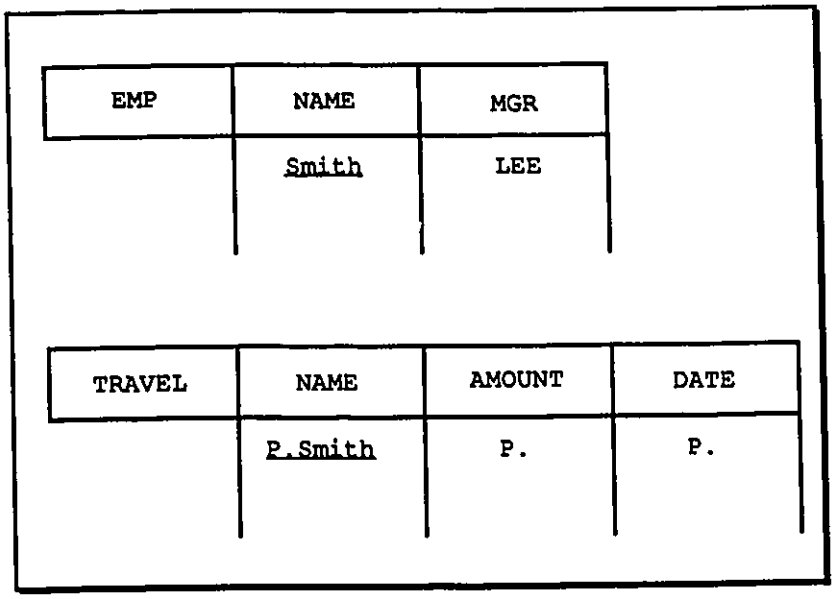

Fig 6.8 A program written in QBE

6.3.2 Discussion

QBE was not designed to produce custom applications. It was designed in order to give database access to non-programmers. The idea of user-programmable systems is very attractive for organizations needing a quick and direct access to information to remain competitive. However, in most organizations (e.g,. a bank), it is preferable to implement rigid controls within the software in order to eliminate the risk of people tampering with sensitive data. LISE was conceived for the design of that type of software.

The motivation for comparing LISE to QBE comes from the usage of examples in both systems. In QBE, the term example is mainly a metaphor for variable. The term example minimizes the confusion that the term variable may bring to a naive user.

As a final note, the interface aspect of querying by example is not the main contribution of QBE. The main contribution is the sophisticated database manipulation operations that the system infers from the simple queries.

Chapter 7

Future Work

We believe there are many ways to enhance our system. First, the overall learning approach could be improved. Second, the system's user interface could benefit from a better design. Third, the quality of the specification produced could also be enhanced.

7.1 Enhancing the learning approach

The learning approach used by LISE could be improved to be less dependent on the initial specifications contained in the domain theory. This could be achieved by building a better goal structure. Right now, there is no structure for the goals in the domain theory. The goals are only attached to the operations enforcing them.

A better goal structure would still attach the goals to the corresponding operations but would also relate the goals together in a hierarchical organization. Low-level goals would be specialization of high-level goals. To replace an unproven antecedent in a partial explanation, LARS could use the goal hierarchy to determine which fact of the training example is more likely to be analogous to the unproven antecedent when both their goals are different. Such an organization would enable goal inference and generalization of goals.

LISE could be enhanced to provide a means to handle partial explanations which can not be completed using abduction, analogical reasoning and case-based reasoning. Such partial explanations may be produced for a training example introducing facts which are not related to any known goals. One enhancement would be to prompt the expert for the goals of the training example facts. LISE could be able to identify which fact are relevant and which facts are analogous to unproven antecedents.

Another enhancement to handle the same problem would be to turn off learning and provide an empty specification template. The expert would fill the template with the specification for the new training example. Unfortunately, the expert could introduce incorrectness in the domain theory. LISE would have to be adapted to deal with incorrect domain theories.

7.2 Enhancing the user interface

The user interface in our current implementation is very limited. The user issues commands at the Prolog interpreter prompt. A basic help facility ensures that the user is reminded of the commands at any time. To enhance our implementation with a basic windowing capability, where the expert would have a window on the domain theory, a window on the goal structure, a window for the training example entry and a command window, would require that we use a different Prolog interpreter. That interpreter should include the infrastructure required for a window-based interface.

The interface could also be improved to allow the user to enter the training example in a restricted English-like syntax. That improvement would make our system comparable to WATSON described in section 6.2.2.

7.3 Enhancing the specification quality

The specification produced by LISE is not executable because the primitive operations are not implemented. It may be interesting to define the operationality criterion according to a database manipulation language (DML) or to a set of packages in Ada. The specification produced by LISE could then be executed. LISE would then become a programming-by-example system.

Conclusion

The various contributions of the research described in this thesis are important. A novel approach to deal with an incomplete theory based on abduction, analogical reasoning and case-based reasoning was successfully defined. This approach makes explanation-based learning more competitive with other learning systems by making it less dependent on the quality of its domain theory.

The area of application of our learning system is one that is in great need of automation: software design. Not only do we propose a tool which can automate the recording of the specification from examples, but can also ensure that the specification is complete and correct.

Our complexity analysis indicated that the approach is at most polynomial. This is better than the undesirable NP-complete problem of exploring the space of partial explanations and of matching training examples to cases. That performance was achieved by using special heuristic to limit the search space in both the domain theory and the case-base.

We demonstrated that our approach was well-founded with an implementation in the language Prolog. Significant portions of the specification of a banking system and a fleet management system were successfully produced. The implementation runs on the IBM PC and was programmed in Arity Prolog.

References

Berdagano, F. and Giordana, A. (1988) "A Knowledge Intensive Approach to Concept Induction, " Procs. of *5th International Conference on Machine Learning.*, 1988.

Brodie, M.L. and Ridjanovic, D. (1984). "On The Design of Database Transactions," in *On Conceptual Modelling (Perspectives from Artificial Intelligence and Programming Languages)* , M. L. Brodie, J. Mylopoulos and J. W. Schmidt, ed. , Springer-Verlag, New York, pp. 277-312, 1984.

Carbonell, J. (1986). "Derivational Analogy: A Theory of Reconstructive Problem Solving and Expertise Acquisition," in *Machine Learning: An Artificial Intelligence Approach* , vol. 2, R. S. Michalski, J. Carbonell and T. Mitchell, ed. , Morgan-Kaufmann, pp. 1986.

Chen, Peter P.S., (1983). *Entity-Relationship Approach to Information Analysis and Modelling* , Elsevier, 1983.

DeJong, G.F. and Mooney, R. (1986). "Explanation-Based Learning: An Alternative View, " *Machine Learning*, vol. 1, pp. 145-176, 1986.

Dietterich, T. (1986). "Learning at the Knowledge Level, " *Machine Learning*, vol. 1, no. 3, 1986.

Dietterich, T.G. and Flann, N.S. (1988) "An Inductive Approach to the Imperfect Theory Problem, " Procs. of *AAAI Spring Symposium on Explained-Based Learning*, pp. 42-46, 1988.

Ellman, T. (1989). "Explanation-Based Learning: A survey of Programs and Perspective., *ACM Computing Surveys*, vol. 21, no. 2, pp. 163-221, 1989.

Fawcett, T. (1989) "Learning from Plausible Explanations, " Procs. of *9th International Conference on Machine Learning*, Ithaca, pp. 37-39, 1989.

Gane, C. and Sarson, T. (1979). *Structured Systems Analysis* , Prentice Hall, 1979.

Genest, J. and Matwin, S. (1990a) "Building Systems Specification using Explanation-Based Learning with an Incomplete Theory, " Procs. of *Canadian Society of Computational Studies of Intelligence*, Ottawa, 1990a.

Genest, J., Matwin, S. and Plante, B. (1990b) "Explanation-Based Learning with Incomplete Theories: A three-step approach, " Procs. of *7th International Conference on Machine Learning*, M. Kaufmann, ed., Austin, 1990b.

Hall, R.J. (1988). "Learning By Failing to Explain: Using Partial Explanations to Learn in Incomplete or Intractable Domains, " *Machine Learning*, vol. 3, no. 1, pp. 45-77, 1988.

Hill. (1988) "Machine Learning for Software Reuse, " Procs. of *10th International Joint Conference on Artificial Intelligence*, pp. 338-344, 1988.

Holte, R.C., Acker, L.E. and Porter, B. (1989) "Concept Learning and the Problem of Small Disjuncts., " Procs. of *11th International Joint Conference on Artificial Intelligence*, Detroit, MI, pp. 813-818, 1989.

Kelly, V.E. and Nonnenmann, U. (1988) "Inferring Formal Software Specification from Episodic Descriptions, " Procs. of 1988.

Kodratoff, Y. (1988). *Introduction to Machine Learning* , Morgan-Kaufmann, 1988.

Liskov, B. and Guttag, S. (1986). *Abstraction and Specification in Program Development* , MIT Press, Cambridge, 1986.

Matwin, S. and Plante, B. (1990) "Learning of Flexible Concepts with Theory Revision", Research Report, University Of Ottawa, TR-90-02, 1990

Michalski, R.S., Carbonell, J. and Mitchell, T. ed. (1983). *Machine Learning: An Artificial Intelligence Approach*, **1**, Morgan-Kaufmann, 1983.

Mitchell, T., Keller, R.M. and Kedar-Cabelli, S.T. (1986). "Explanation-Based Generalization: A Unifying View, " *Machine Learning*, vol. 1, pp. 47-80, 1986.

Mooney, R. and Ourston, D. (1989) "Induction over the Unexplained: Integrated Learning of Concepts with Both Explainable and Conventional Aspects, " Procs. of *6th International Conference on Machine Learning*, Alberto Segre, ed., Ithaca, NY, pp. 5-8, 1989.

Muggleton, S. and Buntine, W. (1988) "Machine Invention of First-Order Predicates by Inverting Resolution, " Procs. of *Fifth Machine Learning Conference*, pp. 339-352, 1988.

Pople, H.E., Jr. (1973) "On the Mechanization of Abductive Logic, " Procs. of *3rd IJCAI*, 1973.

Riesbeck, C. and Schank, R.C. (1990). *Inside Case-based Reasoning* , Lawrence Erlbaum Associates, Hillsdale, NJ, 1990.

Shlear, S. and Mellor S., J. (1988). *Object-Oriented Systems Analysis, Modelling the World in Data* , Yourdon Press, 1988.

Winston, P.H. (1984). *Artificial Intelligence* , Addison-Wesley, 1984.

Yau, S.S. and Tsai, J.J.-P. (1986). "A Survey of Software Design Techniques, " *IEEE Transactions On Software Engineering*, vol. 12, no. 6, pp. 713-721, 1986.

Zloof, M. (1981). "QBE/OBE: A language for Office Automation, " *IEEE Computer.*, May 1981.

Appendix A

Listing of LISE

The code of LISE is located in five files. The content of the files are:

lise.ari:	This file contains the EBL procedure. It also implements the module ELLI (EBL for LISE).
prevexp.ari:	This file contains the procedures to store and retrieve previous explanations.
extrrule.ari:	This file contains the procedures to extract the rules from the frames.
adapt.ari:	This file contains the procedures to adapt previous explanations to new problems. It also contains the procedure for our case-based approach. This file implements the modules LARS (LISE's Analogical Reasoning) and CARL (Cases-based reasoning for LISE).
wrtexpl.ari:	This file contains all the procedures for I/O
banking.ari:	This file contains a domain theory for a banking system used as an example in this thesis. It also contains cases for the case-base module.
trucking.ari:	This file contains a domain theory for a fleet management system.

This appendix presents the five files in the order mentioned above.

```
%-------------------------------------------------------------------------
-
% LISE (Learning In Software Engineering)
%
% (c) Copyright 1990 by Jean Genest, All Rights Reserved
%-------------------------------------------------------------------------
-
%
% LISE is a learning system designed to transform non-operational user
% requirements into operational specifications
%
% For a complete explanation of this system, please consult:
%
%    Genest, J. and Matwin, S., Building System Specifications using
%     Explanation-Based Learning with an Incomplete Theory, Proceedings
%     of CSCSI-90, Ottawa, 1990.
%
%     Genest, J., Matwin, S., Plante, B., Explanation-Based Learning
%     with Incomplete Theories: A Three-Step Approach, Proceedings of
%     the Seventh International Conference on Machine Learning, Austin,
%     Texas, 1990.
%
%     Jear Genest, Building System Specifications using
%     Explanation-Based Learning with an Incomplete Theories, Master
%     Degree Thesis, University of Ottawa, 1990.
%
% Required files:
%  lise.ari: This file. Contains the EBL procedure. Implements the
% model ELLI (EBL for LISE)
%  prevexp.ari:  contains procedures for storing and retreiving previous
%               explanations
%  extrrule.ari: contains procedures to extract the rules from the
frames
%  adapt.ari:   contains procedures to adapt previous explanations to new
%               problems. Also contains the procedure to apply cases
%               from the case-base. Implements the modules LARS (LISE's
%               Analogical Reasoning) and CARL (Cases-based reasoning for
%               LISE)
%  wrtexpl.ari:  contains all the procedures for I/O
%  banking.ari:  contains a demo domain theory for the banking domain.
Also
%               contains cases for the case-base module.
%  trucking.ari: contains a demo domain theory for the domain trucking.
%
%
%  Note: LISE prompts you for the name of the domain theory. To use
%        the demo, enter banking. To use your own domain theory, make
%        your own file and enter its name. All of the above reference
%        explain the format of the domain theory and banking is a good
%         example.

%-------------------------------------------------------------------------
---

:- op(800,xfx,<-).

:- nl,write('*** LISE WELCOMES YOU ***'),nl,
      write('Please type ''cons'' to load other files'),nl.
```

```
cons :- [prevexp], [wrtexpl], [adapt], [cases], [extrrule].

el :- edit(lise).

%    operator for the clauses
%    ------------------------

%    the EBG program
%    ---------------
%     The following EBG program is inspired from Kodratoff's Introduction
%     to Machine Learning. It was modified in order to produce all
partial
%     explanations when the domain theory is incomplete

ebg((A, B),(GenA, GenB), Result, Unsolved, Tree):-
     !,
     ebg(A, GenA, ResultA, UnsolvedA, TreeA),
     ebg(B, GenB, ResultB, UnsolvedB, TreeB),
     append(ResultA, ResultB, Result),
     append(UnsolvedA, UnsolvedB, Unsolved),
     append(TreeA, TreeB, Tree).

% ebg(A, GenA, [GenA], Unsolved, Tree):-
%     nonop(A),
%     clause(A, true), !.

ebg(A, GenA, Result, Unsolved, [subtree(GenA,Tree)]):-
     nonop(A), !,
     clause(GenA, GenB),
     copy_term((GenA:-GenB),(A:-B)),
     res(B, GenB, Result, Unsolved, Tree).

ebg(A, GenA, [GenA], _, [GenA]) :-
     call(A),
     assert(feature_used(A)).

ebg(A, GenA, _, [GenA], [unproven(GenA)]).

res(B, GenB, Result, Unsolved, Tree) :-
     ebg(B, GenB, Result, Unsolved, Tree),
     !.

%    procedure solve_the_unsolved
%    ----------------------------
%     This procedure solves the unsolved portion of a partial explanation

solve_the_unsolved(Old_result,Old_result,Unsolved,[],_) :-
     var(Unsolved).

solve_the_unsolved(Old_result,Old_result,[Unsolved],_,_) :-
      functor(Unsolved,Pred_name,Arity),
      training_example(Trg_example),
      functor(Trg_example,Pred_name,Arity).

solve_the_unsolved(Old_result,New_result,Unsolved,Remainder,Message) :-

solve_by_abduction(Old_result,New_result,Unsolved,Remainder,Message).

solve_the_unsolved(Old_result,Old_result,Unsolved,Unsolved,_).
```

```
%    procedure solve_by_abduction
%    ----------------------------
%     This is the procedure used to complete the partial explanation
using
%     abduction

solve_by_abduction(
            Old_result,
            New_result,
            Unsolved,
            Remainder,
            Message)  :-
     locate_abductive_clause(Unsolved,Solved,RemainderA,Head,Body),
     \+ Solved = [],
     append(Solved, Old_result, Old_resultA),
     solve_by_abduction(
      Old_resultA,
      New_result,
      RemainderA,
      Remainder,
      MessageA),
     append([message_on_abduction(Head,Body)],MessageA,Message).

solve_by_abduction(New_result,New_result,Remainder,Remainder,[]).

%    procedure locate_abductive_clause
%    ---------------------------------
%     This procedure search throught the database for a rule which
%     could be used to abduce the unproven antecedent of the partial
%     explanation

locate_abductive_clause([],_,_,_,_) :- !, fail.

locate_abductive_clause(Unsolved,Solved,[],Head,Body) :-
     [Head] <- Body,
     call([Head]),
     del_body_term_from_unsolved(Body,Unsolved,Solved,[]),
     !.

locate_abductive_clause(Unsolved,Solved,[Remainder],Head,Body) :-
     [Head] <- Body,
     call([Head]),
     del_body_term_from_unsolved(Body,Unsolved,Solved,[Remainder]),
     !.

locate_abductive_clause(Unsolved,Solved,Remainder,Head,Body) :-
     [Head] <- Body,
     call([Head]),
     del_body_term_from_unsolved(Body,Unsolved,Solved,Remainder).

%     procedure message_on_abduction, inform_abduction and
mention_other_hyp
%     ------------------------------------------------------------------
---
%    These predicates are used to inform the user that abduction took
place.
%     It also mentions the other hypotheses that could have been used to
%     deduce the consequent of the rule used in abduction
```

```
message_on_abduction(Head,Body) :-
     inform_abduction(Head,Body),
     mention_other_hyp(Head,Body).

inform_abduction(Head,Body) :-
     tab(3),write('Abduction was used with hypothesis on rule:'),nl,
     tab(3),write([Head <- Body]),nl.

mention_other_hyp(Head,BodyA) :-   % A is the head
     locate_clause(Head,BodyB),
     BodyA \= BodyB,
     BodyB \= true,
     tab(3),write('Other possibilities are: '),nl,
     tab(3),write([Head <- BodyB]),nl,nl.

mention_other_hyp(_,_) :-
      tab(3),write('No other hypothesis known.'),nl,nl.

locate_clause(Head,Body) :-
     [Head] <- Body.

%     procedure prepare_rules_for_ebg, prepare_rules and prepare_facts
%     ----------------------------------------------------------------
%    The following procedure transform the rules from the format
%     [head] <- [boby] to the standard format head :- body.

prepare_rules_for_ebg :-
     nl,
     write('The domain theory'),nl,
     write('-----------------'),nl,
     prepare_rules,
     listing(<-),nl,nl,
     write('The training example'),nl,
     write('--------------------'),nl,
      prepare_facts,nl,nl.

prepare_rules :-
      [A] <- Body,
      [!convert_list_to_conjunction(Body,Conjunction),
      assert((A :- Conjunction))!],
      fail.

prepare_rules.

prepare_facts :-
      [A],
      [!write([A]),nl,
      assert(A)!],
      fail.

prepare_facts.

%     procedure convert_list_to_conjunction
%     -------------------------------------
%     This procedure converts a list in the form [A,B,...N] into a
%     conjunction in the form (A,B,...N) ready to use as the body of a
%     clause
```

```
convert_list_to_conjunction([A|Tail],(A, B)) :-
      convert_list_to_conjunction(Tail, B).

convert_list_to_conjunction([A], A).

%     procedure copy_term
%     -------------------
%      Copy_term is used to make a copy of a rule with new variables

copy_term(Term,Copy) :-
      recorda(Copy,copy(Term),DBref),
      instance(DBref,copy(Temp)),
      erase(DBref),
      Copy = Temp.

%     procedure append
%     ----------------
%
append([],L,L).

append(L,[],L).

append([X|L1],L2,[X|L3]) :-
      append(L1,L2,L3).

%     procedure member
%      ----------------
%
member(X,[X|Tail]).

member(X,[Head|Tail]) :-
      member(X,Tail).

%     procedure member
%      ----------------
%      This procedure check if an unstantiated variable is the member of a
%      list of variables

member_as_var(X,[Y|Tail]) :-
      X == Y.

member_as_var(X,[Head|Tail]) :-
      member_as_var(X,Tail).

%      procedure del_body_term_from_unsolved
%      -------------------------------------
%     This procedure deletes the terms that were solved from the list
%     of the unsolved terms. It is used when more than one rule is
%      necessary to solve the unproven antecedent using abduction

del_body_term_from_unsolved([X|Tail],Unsolved,Solved,Remainder) :-
      del_body_term(X,Unsolved,SolvedA,RemainderA),
      del_body_term_from_unsolved(Tail,RemainderA,SolvedB,Remainder),
      append(SolvedA,SolvedB,Solved).

del_body_term_from_unsolved([],List,[],List).

%      procedure del_body_term
%      -----------------------
```

```
%    deleting a term from a list without
%    instantiating the variables

del_body_term(X,[Y|Tail1],[Y|Tail2],Tail3) :-
     X =.. [X1|L1],
     Y =.. [Y1|L2],
     X1 == Y1,
     compatible_arguments(L1,L2),
     del_body_term(X,Tail1,Tail2,Tail3).

del_body_term(X,[Y|Tail1],Tail2,[Y|Tail3]) :-
     del_body_term(X,Tail1,Tail2,Tail3).

del_body_term(X,[],[],[]).

%     procedure compatible_arguments
%     ------------------------------
%    this predicate makes sure that a variable in the theory match
%     with a variable or a constant in the generalized explanation,
%     a constant in the theory can only be paired with a similar
%     constant in the example.

compatible_arguments([X|L1],[Y|L2]) :-
     nonvar(X),
     nonvar(Y), !,
     X == Y,
     compatible_arguments(L1,L2).

compatible_arguments([X|L1],[Y|L2]) :-
     compatible_arguments(L1,L2).

compatible_arguments([],[]).  % this ensures that they are the same size

%    procedure run
%    -------------
%     This procedure starts the learning procedure by first reading in
the
%     domain theory and the initial training example

run :-
     write('Enter the name of the file containing the domain theory:
'),nl,
     read_line(0,File1),
     consult(File1),
     write('Enter the name of the file containing the trg example:
'),nl,
     read_line(0,File2),
     consult(File2),
     extract_rules_from_frames,
     prepare_rules_for_ebg,
     process_training_example,
      keep_successful_explanation,
      reset_expl.

%     procedure pte
%     -------------
%     this procedure is an abreviate command to process the current
%     training example using the current domain theory. Process
%     training example first tries to apply a previous explanation to
```

```
%     the training example and will apply EBG if that does not work.

pte :-
      process_training_example,
      reset_expl.

process_training_example :-
      solve_using_previous_explanation.

process_training_example :-
      ebg,
      keep_successful_explanation.

%     procedure ebg
%     ------------
%     This is the initial call to ebg when we first attempt to explain a
%     training example

ebg :-
     training_example(Goal_with_constant),
     functor(Goal_with_constant,Pred_name,Arity),
     functor(Goal_with_var,Pred_name,Arity),
     ebg_process(Goal_with_constant,Goal_with_var),
     check_for_successful_explanations,
     display_successful_explanation.

ebg :-
     ebg_process(transaction(_),transaction(_)).

ebg :-
      [!display_partial_explanations,nl,nl,
      write(' *** MESSAGE from LISE *** '),nl,nl,
      write('Do you want to build a new transaction'), nl,
      write('from the partial explanations? (y/n) ')!],
     read_line(0,$y$),nl,nl,
     build_frames,
     assert_new_frames.

ebg.

ebg_process(Goal_with_constant,Goal_with_var) :-
      multiple_expl_ebg(Goal_with_constant,Goal_with_var).

multiple_expl_ebg(Goal_with_cons,Goal_with_var) :-
     ebg(Goal_with_cons,Goal_with_var,Old_expl,Unsolved,Tree),

[!solve_the_unsolved(Old_expl,New_expl,Unsolved,Remainder,Message)!],
     categorize_result(Goal_with_var,New_expl,Remainder,Tree,Message).

% categorize_ result accepts each result and classifies it as either an
% explanation or a partial explanation. Note that we can accept many
% explanation (incomplete theory problem introduce multiple
explanations).

categorize_result(Goal,Result,[],Tree,Message) :-
      training_example(Trg_ex),
      functor(Trg_ex,Trg_ex_name,Arity),
      functor(Goal,Trg_ex_name,Arity),
      nonvar(Result),
```

```
      assert(explanation([Goal,Result,[],Tree,Message])),
      reset_used_features.

categorize_result(Goal,Result,Remainder,Tree,Message) :-
      training_example(Trg_ex),
      functor(Trg_ex,Trg_ex_name,Arity),
      functor(Goal,Trg_ex_name,Arity),
      reset_used_features,
      !,
      fail.

categorize_result(Goal,Result,Remainder,Tree,Message) :-
      nonvar(Result),
      get_features_not_used(Features_not_used),
      assert(
            partial_explanation(

      [Goal,Result,Remainder,Tree,Message,Features_not_used])),
      reset_used_features,
      !,
      fail.

% the following procedure checks for the presence of an explanation
%
check_for_successful_explanations :-
      findall(A,explanation(A),List),
      \+ List == [].

% the following procedure checks displays the explanation
%
display_successful_explanation :-
      findall(A,explanation(A),List),nl,
      write('Successful explanation'),nl,
      write('----------------------'),nl,
      write_successful_expl(List).


% the system does not know about the transaction presented
% as a training example as such,
% but it can built complete explanations using some other
% transactions.
% the following procedure presents the explanations
% which explain the training example using another
%
display_partial_explanations :-
      findall(A,partial_explanation(A),List),nl,
      \+ List == [],
      write('NO successful explanation; Partial explanation(s)
follow:'),nl,
      write('-----------------------------------------------------------
'),nl,
      write_partial_expl(List).

display_partial_explanation :-
      nl,
      write('No partial explanations found'),nl,
      write('---------------------------'),nl.

% if the goal concept doesn't produce any partial explanations
```

```
%  it is because the goal concept is not the consequent of
%  any rules in the domain theory. In that case, we expect
%  to be able to find a similar concept from which we can
%  re-use as much as we can. The best similar concept will be
%  obtained from the previous explanations.

% Then, We will make the
%  concept to learn a subconcept of the parent concept of
%  the most similar concept and generate partial explanations
%  based on the other subconcepts of the parent. The best partial
%  explanation will be the one that we can get the most re-use from.
%  That one will be selected from the set of subconcept using a
%  hieuristic applied to the set of partial explanations.

% For now, the parent concept of the concept to be learned is the
%  one identifid by the relation ISA given part of the training
%  example.


% the following code is used to switch the training example
% without restarting the main program

get_new_te(Trg_Exp) :-
      retract(training_example(A)),
      retract_all_facts,
      reset_expl,
      [Trg_Exp],
      prepare_facts.

retract_all_facts :-
      findall(A,[A],List_of_facts),
      retract_list_of_facts(List_of_facts).

retract_list_of_facts([Fact|Tail]) :-
      retract([Fact]),
      retract(Fact),
      retract_list_of_facts(Tail).

retract_list_of_facts([]).

reset_expl :-
      findall(A,explanation(A),List_of_expl),
      retract_list_of_expl(explanation,List_of_expl),
      findall(B,partial_explanation(B),List_of_part_expl),
      retract_list_of_expl(partial_explanation,List_of_part_expl),
      findall(A,transaction_feature_found(A),List_of_t_f),
      retract_list_of_expl(transaction_feature_found,List_of_t_f).

retract_list_of_expl(Type,[Element|Tail]) :-
      Pred =.. [Type,Element],
      retract(Pred),
      retract_list_of_expl(Type,Tail).

retract_list_of_expl(_,[]).

view_te :-
      nl,write('The training example is: '),
      training_example(TE),
      write(TE),nl,
```

```
      write('The facts are:'),nl,
      findall(A,[A],List_of_facts),
      write_list(List_of_facts).

write_list([Fact|Tail]) :-
      tab(3),write(Fact),nl,
      write_list(Tail).

write_list([]).

find_transaction_name(
      [subtree(transaction(_),[subtree(Transaction,Subtree)])|Tail],
      Transaction).

find_transaction_name(
      [subtree(Transaction,Subtree)|Tail],Transaction).

find_transaction_feature_name(
      [subtree(transaction_feature(_),
       [subtree(Transaction_feature,Subtree)])|Tail],
       Transaction_feature).

get_features_not_used(Feature_not_used) :-
      findall(A,unused_feature(A),Feature_not_used),
      reset_used_features.

unused_feature(A) :-
      [A],
      \+ feature_used(A).

reset_used_features :-
      findall(A,feature_used(A),List),
      retract_used_feature_list(List).

retract_used_feature_list([Pred|Tail]) :-
      Pred_to_ret =.. [feature_used,Pred],
      retract(Pred_to_ret),
      retract_used_feature_list(Tail).

retract_used_feature_list([]).

rpe :-
      findall(A,past_explanation(A),List_of_past_expl),
      retract_list_of_expl(past_explanation,List_of_past_expl).
```

```
%-------------------------------------------------------------------
-
% File adapt.ari from LISE (Learning In Software Engineering)
%
% (c) Copyright 1990 by Jean Genest, All Rights Reserved
%-------------------------------------------------------------------
-

%  adapt.ari contains procedures used to combine one or many
%  partial explanations into a single plausible explanation.

% the partial explanations match on some of the training example
% features. We can verify if these features have a certain goal
% with respect to the background knowledge. If they do, we accept
% the feature as relevant and we look for other features related to
% same goal.

:- op(800,xfx,<-).

ea :- edit(adapt).
% -----------------------------------------------------------------
--
% this is the main procedure used in the construction of a new frame
using
% a set of partial explanations

build_frames :-
      findall(Part_expl,partial_explanation(Part_expl),Part_expl_list),
      length(Part_expl_list,L_part_expl_list),
      write('There was '),write(L_part_expl_list),
      write(' partial explanation(s) found'), nl,nl,
      write('To build a plausible explanation, LISE will use '),nl,
      write('one or many partial explanations'),nl, nl,
      write('The partial explanation will be ranked from the most '), nl,
      write('plausible to the less plausible according to a scoring '),
nl,
      write('function'), nl, nl,
      write('Assigning scores...'),nl, nl,
      assign_score(Part_expl_list,Scored_part_expl_list),
      write('Ranking partial explanations according to score...'),nl, nl,
      quicksort_expl_list(Scored_part_expl_list,Sorted_part_expl_list),
      write('The explanations are, from the best to the worst:'),nl,
      write_expl_with_score(Sorted_part_expl_list),nl,
      flush,
      write('Hit any key to continue...'),
      get0_noecho(X),nl,nl,
      build_new_transaction(
            Sorted_part_expl_list,
            New_t_name,
            New_t_parents,
            New_t_properties),
      assert(
            nf(name(New_t_name),isa(New_t_parents),New_t_properties)).

% -----------------------------------------------------------------
--
%
```

```
rename_action_and_assert_if_necessary(New_action_name) :-
      nf(name(Old_action_name),isa([action]),Tran_properties),
      Old_action_name =.. [new_action|Tail],
      write('The frame named: '),write(Old_action_name),
      write(' has to be renamed'),nl,
      write('Please enter new name, without the parameters: '),
      read_line(0,New_name_strg),nl,nl,
      atom_string(New_name,New_name_strg),
      New_action_name =.. [New_name|Tail],
      assert(frame(name(New_action_name),isa([action]),Tran_properties)),
      write('The new action was renamed, the new frame is:'),nl,nl,
      write('Name: '),write(New_action_name),nl,
      write(' Parents: '),write([action]),nl,
      write_properties(Tran_properties),nl,nl,
      retract(nf(name(Old_action_name),isa([action]),Tran_properties)).

rename_action_and_assert_if_necessary(_).

% ----------------------------------------------------------------------
--
% The next procedure is used to assign a score to the partial
explanation.
% The higher the score, the more plausible it is.

assign_score(
      [[Goal,Result,Remainder,Tree,Message,Features_not_used]|Tail1],
      [[Score,Goal,Result,Remainder,Tree,Message,Features_not_used]|Tail2
]) :-
      length(Result,L_result),
      length(Remainder,L_remainder),
      Subscore1 is L_result - L_remainder,
      length(Features_not_used,L_f_not_used),
      Subscore2 is Subscore1 - L_f_not_used,
      length(Message,L_message),
      L_message_score is L_message / 2,
      Score is Subscore2 - L_message_score,
      assign_score(Tail1,Tail2).

assign_score([],[]).

% ----------------------------------------------------------------------
--
% Once the best partial explanation is selected, it is passed to
% the next procedure which use it as a starting point to build a
% new frame

build_new_transaction(

      [[_,_,Common_features,Remainder,Tree,_,Features_not_used]|TailE],
            New_frame_name,
            Parents,
            [precondition(Prec),procedure(Proc)]) :-
      check_for_coverage_of_TE_features(Common_features),
      find_transaction_name(Tree,Source_transaction_name),
      frame(
            name(Source_transaction_name),
            isa(ParentsA),
            Source_tran_properties),
      write('Building a new frame using: '),
```

```
      write(Source_transaction_name),nl,nl,
      Source_transaction_name =.. [Tran_pred_name|Tail],
      training_example(Name),
      Name =.. [New_frame_pred_name|Tail2],
      New_frame_name =.. [New_frame_pred_name|Tail],
      adapt_transaction_prec_and_proc(
            partial_explanation,
            Common_features,
            Features_not_used,
            Common_features_used,
            Source_tran_properties,
            Transaction_propertiesA),
      flush,
      write('Hit any key to continue...'),
      get0_noecho(X),nl,nl,
      remove_common_features_and_score(Common_features_used,TailE,Pruned_
tail),
      assign_score(Pruned_tail,Rescored_tail),
      quicksort_expl_list(Rescored_tail,Sorted_part_expl_list),
      build_new_transaction(
            Sorted_part_expl_list,
            New_frame_name,
            ParentsB,
            Transaction_propertiesB),
      member(precondition(PrecA),Transaction_propertiesA),
      member(procedure(ProcA),Transaction_propertiesA),
      member(precondition(PrecB),Transaction_propertiesB),
      member(procedure(ProcB),Transaction_propertiesB),
      merge_list(ParentsA,ParentsB,Parents),
      append(PrecA,PrecB,Prec),
      append(ProcA,ProcB,Proc).

build_new_transaction(

      [[_,_,Common_features,Remainder,Tree,_,Features_not_used]|Tail],
            New_frame_name,
            Parents,
            Transaction_prop) :-
      build_new_transaction(Tail,New_frame_name,Parents,Transaction_prop)
.

build_new_transaction(
            [],
            New_frame_name,
            [],
            [precondition([]),procedure([])]).

% ------------------------------------------------------------------------
--
% this procedure verifies if a partial explanation covers at least
% one element of the training example that has not been used yet
% to build the new transaction

check_for_coverage_of_TE_features([Common_feature|Tail]) :-
      [Training_example_feature],
      Training_example_feature =.. [Feature_pred_name|TailX],
      Common_feature =.. [Feature_pred_name|TailY].

check_for_coverage_of_TE_features([Common_feature|Tail]) :-
```

```
      check_for_coverage_of_TE_features(Tail).

check_for_coverage_of_TE_features([]) :- fail.

% ------------------------------------------------------------------------
--
% the list of partial explanations need to be resorted after
% some of the features have been used to build a transaction

remove_common_features_and_score(
      Common_features_used,
      [[Score,B,Common_features,C,D,E,F]|Tail1],
      [[B,Cleaned_common_features,C,D,E,F]|Tail2]) :-
      del_feature_list(
            Common_features_used,
            Common_features,
            Cleaned_common_features),
      remove_common_features_and_score(Common_features_used,Tail1,Tail2).

remove_common_features_and_score(Common_features_used,[],[]).

del_feature_list([Feature|Tail],List1,List2) :-
      del_feature(Feature,List1,List3),
      del_feature_list(Tail,List3,List2).

del_feature_list([],List,List).

del_feature_list(List,[],[]).

del_feature(Feature,[Feature|Tail], Tail).

del_feature(Feature,[],[]).

del_feature(Feature,[Other_feature|Tail1],[Other_feature|Tail2]) :-
      del_feature(Feature,Tail1,Tail2).

% ------------------------------------------------------------------------
--
% this procedure adapt the transaction properties according to
% the current partial explanations

adapt_transaction_prec_and_proc(
            Type,
            Common_features,
            Features_not_used,
            Common_features_used,
            Tran_properties,
            [precondition(New_frame_prec),
             procedure(Converted_frame_proc)]) :-
      member(precondition(Tran_prec),Tran_properties),
      convert_feature_list(
            Type,
            Common_features,
            Features_not_used,
            Tran_prec,
            New_frame_prec,
            Common_features_usedA),
      member(procedure(Tran_proc),Tran_properties),
      convert_feature_list(
```

```
            Type,
            Common_features,
            Features_not_used,
            Tran_proc,
            New_frame_proc,
            Common_features_usedB),
      append(
            Common_features_usedA,
            Common_features_usedB,
            Common_features_used),
      convert_parameters(
            New_frame_proc,
            New_frame_prec,
            New_frame_proc,
            Converted_frame_proc).


% ----------------------------------------------------------------------
--
% Convert feature list adapts relevant features from the training example
by
% integrating them in the new frame.
% A training example feature is relevant if :
%     a. it appears in the partial explanation, or,
%     b. it has the same purpose as a feature of the partial explanation
%       which is not found directly in the training example,
% Some feature from the partial explanation are inserted in the frame
even
%  if they are not training example feature when they are constraint
%  expressed using a relational operator ('=','<','>',etc.)

convert_feature_list(
            Source_type,
            Common_features,
            Features_not_used,
            [Tran_prec|Tail1],
            New_frame_features,
            Common_features_used) :-
      member(Tran_prec,Common_features),
      write('Feature of TE deemed relevant because it was found in the
'),
      write(Source_type),write(' :'),nl,
      write(Tran_prec),nl,nl,
      convert_feature_list(
            Source_type,
            Common_features,
            Features_not_used,
            Tail1,
            New_frame_featuresA,
            Common_features_usedA),
      append([Tran_prec],New_frame_featuresA,New_frame_features),
      append([Tran_prec],Common_features_usedA,Common_features_used).

convert_feature_list(
            Source_type,
            Common_features,
            Features_not_used,
            [Tran_prec|Tail1],
            New_frame_features,
```

```
          Common_features_used) :-
      purpose(Tran_prec,constraint),
      write('Constraint from '),
      write(Source_type),
      write(' kept :'),nl,
      write(Tran_prec),nl,nl,
      convert_feature_list(
            Source_type,
            Common_features,
            Features_not_used,
            Tail1,
            New_frame_featuresA,
            Common_features_used),
      append([Tran_prec],New_frame_featuresA,New_frame_features).

convert_feature_list(
            Source_type,
            Common_features,
            Features_not_used,
            [Tran_prec|Tail1],
            New_frame_features,
            Common_features_used) :-
      purpose(Tran_prec,Purpose),
      member(Feature,Features_not_used),
      purpose(Feature,Purpose),
      Feature =.. [Feature_name|Tail3],
      Tran_prec =.. [Tran_prec_name|Tail4],
      New_feature =.. [Feature_name|Tail4],
      write('Feature of training example found relevant because it has'),
      write(' the same purpose as a'),nl,
      write(Source_type),
      write(' feature.'),nl,
      write('Training example feature :'),write(Feature),nl,
      write(Source_type),
      write(' feature :'),write(Tran_prec),nl,
      write('Purpose shared:'),write(Purpose),nl,nl,
      convert_feature_list(
            Source_type,
            Common_features,
            Features_not_used,
            Tail1,
            New_frame_featuresA,
            Common_features_usedA),
      append([New_feature],New_frame_featuresA,New_frame_features),
      append([New_feature],Common_features_usedA,Common_features_used).

convert_feature_list(  % this is the case where the action was found
            Source_type,
            Common_features,
            Features_not_used,
            [Tran_proc|Tail1],
            New_frame_features,
            Common_features_used) :-
      frame(name(Tran_proc),isa(Parents),Tran_properties),
      copy_term((Tran_proc :- Body),(Tran_proc2 :- Body2)),
      Tran_proc2,
      [Tran_proc] <- Body3,
      length(Body3,L_body),
      write(L_body),
```

```
      write(' feature(s) of the training example found relevant
because'),nl,
      write('they are the features necessary for an action in the'),nl,
      write(Source_type),nl,
      write('Training example feature :'),write(Body3),nl,
      write('Action of the partial explanation
:'),write(Tran_proc),nl,nl,
      convert_feature_list(
            Source_type,
            Common_features,
            Features_not_used,
            Tail1,
            New_frame_featuresA,
            Common_features_usedA),
      append([Tran_proc],New_frame_featuresA,New_frame_features),
      append(Body3,Common_features_usedA,Common_features_used).

convert_feature_list(
            Source_type,
            Common_features,
            Features_not_used,
            [Tran_proc|Tail1],
            New_frame_features,
            Common_features_used) :-
      [!frame(name(Tran_proc),isa(Parents),Tran_properties),
      adapt_transaction_prec_and_proc(
            Source_type,
            Common_features,
            Features_not_used,
            Common_features_usedA,
            Tran_properties,
            New_tran_properties),
      member(procedure(Proc),New_tran_properties)!],
      \+ member(not_founded(_),Proc),
      write('A new action was found, its properties are:'),nl,
      write(New_tran_properties),nl,
      write('Please enter a name for the new action without parameters:
'),
      read_line(0,New_name_strg),nl,nl,
      atom_string(New_name,New_name_strg),
      Tran_proc =.. [Tran_proc_pred_name|Tail3],
      New_action =.. [New_name|Tail3],
      assert(naf(name(New_action),isa(Parents),New_tran_properties)),
      convert_feature_list(
            Source_type,
            Common_features,
            Features_not_used,
            Tail1,
            New_frame_featuresA,
            Common_features_usedB),
      append(
            Common_features_usedA,
            Common_features_usedB,
            Common_features_used),
      append([New_action],New_frame_featuresA,New_frame_features).

convert_feature_list(
            Source_type,
            Common_features,
```

```
            Features_not_used,
            [Tran_proc|Tail1],
            New_frame_features,
            Common_features_used) :-
      frame(name(Tran_proc),isa(Parents),Tran_properties),
      Tran_proc =.. [Tran_proc_pred_name|Tail3],
      convert_feature_list(
            Source_type,
            Common_features,
            Features_not_used,
            Tail1,
            New_frame_featuresA,
            Common_features_used),
      append([not_founded(Tran_proc)],New_frame_featuresA,New_frame_featu
res).

convert_feature_list(
            Source_type,
            Common_features,
            Features_not_used,
            [Tran_prec|Tail1],
            New_frame_features,
            Common_features_used) :-
      write('Feature in the '),
      write(Source_type),
      write(' for which no '),
      write('equivalence was found'),nl,
      write('Feature of the '),
      write(Source_type),
      write(' : '),
      write(Tran_prec),nl,nl,
      convert_feature_list(
            Source_type,
            Common_features,
            Features_not_used,
            Tail1,
            New_frame_featuresA,
            Common_features_used),
      append(
            [not_founded(Tran_prec)],
            New_frame_featuresA,
            New_frame_features).

convert_feature_list(_,_,_,[],[],[]).

%-------------------------------------------------------------------------
% the parameters of the action generated
% need to be  adapted to the context of the
% transaction calling the action

convert_parameters(
            [New_action|Tail1],
            Frame_prec,
            Frame_proc,
            [New_action3|Tail2]) :-
      New_action =.. [new_action|Tail3],
      nf(name(New_action),isa(Act_par),Act_prop),
      extract_rule(action,New_action,Act_par,Act_prop),
      prepare_rule(New_action),
```

```
      copy_term((New_action :- Body1),(New_action2 :- Body2)),
      New_action2,
      append(Frame_prec,Frame_proc,Frame_predicates),
      constants_to_vars(New_action2,Frame_predicates,New_action3),
      convert_parameters(Tail1,Frame_prec,Frame_proc,Tail2),
      retract((New_action :- Body3)),
      retract([New_action] <- Body4).

convert_parameters(
            [Action|Tail1],
            Frame_prec,
            Frame_proc,
            [Action|Tail2]) :-
      convert_parameters(Tail1,Frame_prec,Frame_proc,Tail2).

convert_parameters([],_,_,[]).

constants_to_vars(
            New_action2,
            Frame_predicates,
            New_action3) :-
      vars_n_consts_in_ord_list(Frame_predicates,List_of_Const,List_of_va
rs),
      New_action2 =.. [Pred_name|Constants],
      constants_to_vars(Constants,List_of_Const,List_of_vars,Vars),
      New_action3 =.. [Pred_name|Vars].

vars_n_consts_in_ord_list(
            [Pred|Tail],
            List_of_Const,
            List_of_vars) :-
      Pred =.. [new_action|Vars],
      vars_n_consts_in_ord_list(Tail,List_of_Const,List_of_vars).

vars_n_consts_in_ord_list(
            [Pred|Tail],
            List_of_Const,
            List_of_vars) :-
      Pred =.. [Pred_name|Vars],
      [TE_feature],
      TE_feature =.. [Pred_name|Consts],
      vars_n_consts_in_ord_list(Tail,List_of_ConstA,List_of_varsA),
      append(Consts,List_of_ConstA,List_of_Const),
      append(Vars,List_of_varsA,List_of_vars).

vars_n_consts_in_ord_list(
            [Pred|Tail],
            List_of_Const,
            List_of_vars) :-
      vars_n_consts_in_ord_list(Tail,List_of_Const,List_of_vars).

vars_n_consts_in_ord_list([],[],[]).

constants_to_vars(
            [Cons|Tail1],
            List_of_Const,
            List_of_vars,
            [Var|Tail2]) :-
      constant_to_var(Cons,List_of_Const,List_of_vars,Var),
```

```
      constants_to_vars(Tail1,List_of_Const,List_of_vars,Tail2).

constants_to_vars([],_,_,[]).

constant_to_var(Cons,[Cons|Tail],[Var|Tail2],Var).

constant_to_var(Cons,[ConsA|Tail1],[VarA|Tail2],Var):-
      constant_to_var(Cons,Tail1,Tail2,Var).

constant_to_var(_,[],[],_).

prepare_rule(New_action) :-
      [New_action] <- Body,
      convert_list_to_conjunction(Body,Conjunction),
      assert((New_action :- Conjunction)).

convert_list_to_conjunction([A|Tail],(A, B)) :-
      convert_list_to_conjunction(Tail, B).

convert_list_to_conjunction([A], A).

% ------------------------------------------------------------------------
--
% Assert_transaction is used after learning is done
% the first assert_transaction ensure that the name
% of a new action is changed

assert_new_frames :-
      assert_transaction1,
      assert_transaction2,
      write_and_assert_new_actions,
      nf2(name(Transaction),isa(Parents),Tran_properties),
      write('The new transaction frame is:'),nl,nl,
      write('Name: '),write(Transaction),nl,
      write(' isa: '),write(Parents),nl,
      write_properties(Tran_properties),nl,nl,
      assert(frame(name(Transaction),isa(Parents),Tran_properties)),
      retract(nf2(name(Transaction),isa(Parents),Tran_properties)).

write_and_assert_new_actions :-
      write_and_assert_new_action.

write_and_assert_new_actions.

write_and_assert_new_action :-
      naf(name(Action),isa(Parents),Ac_prop),
      [!write('A new action frame was created:'),nl,nl,
      write('Name: '),write(Action),nl,
      write(' isa: '),write(Parents),nl,
      write_properties(Ac_prop),nl,nl,
      flush,
      write('Hit any key to continue...'),
      get0_noecho(X),nl,nl,
      assert(frame(name(Action),isa(Parents),Ac_prop)),
      retract(naf(name(Action),isa(Parents),Ac_prop))!],
      fail.

% ------------------------------------------------------------------------
--
```

```
% this assert_transaction  tries to apply a rewrite rule if
% there is not_founded facts

assert_transaction1 :-
      [!nf(name(Transaction),isa(Parents),Tran_properties),
      member(precondition(Prec),Tran_properties),
      member(procedure(Proc),Tran_properties)!],
      member(not_founded(A),Proc),
      check_for_rewrite_rule(Proc,New_proc),
      assert(nf1(
            name(Transaction),
            isa(Parents),
            [precondition(Prec),procedure(New_proc)])),
      retract(nf(name(Transaction),isa(Parents),Tran_properties)).

assert_transaction1 :-
      nf(name(Transaction),isa(Parents),Tran_properties),
      assert(nf1(name(Transaction),isa(Parents),Tran_properties)),
      retract(nf(name(Transaction),isa(Parents),Tran_properties)).

assert_transaction2 :-
      [!nf1(name(Transaction),isa(Parents),Tran_properties),
      member(precondition(Prec),Tran_properties),
      member(procedure(Proc),Tran_properties)!],
      [!member(not_founded(A),Proc)!],
      write('Do you want to search for previous case? (y/n) '),
      read_line(0,$y$), nl,nl,
      write('Searching for previous case...'),nl,nl,
      findall(Case,applicable_prev_case(Case),Case_list),
      \+ Case_list = [],
      write('Previous case found...'),nl,nl,
      length(Case_list,L_part_expl_list),
      write('There was '),write(L_part_expl_list),
      write(' case(s) found'), nl,nl,
      write('To build the new transaction, LISE will use '),nl,
      write('one or many cases'),nl, nl,
      write('The cases will be ranked from the most applicable '), nl,
      write('to the less applicable according to a scoring function'),
nl,nl,
      write('Assigning scores...'),nl, nl,
      assign_score(Case_list,Scored_case_list),
      write('Ranking cases according to score...'),nl, nl,
      quicksort_expl_list(Scored_case_list,Sorted_case_list),
      write('The cases are, from the best to the worst:'),nl,
      write_cases_with_score(Sorted_case_list),nl,
      flush,
      write('Hit any key to continue...'),
      get0_noecho(X),nl,nl,
      build_new_trans_from_cases(
            Sorted_case_list,
            New_transaction,
            New_Parents,
            New_tran_properties),
      assert(nf2(name(New_transaction),isa(Parents),New_tran_properties))
,
      retract(nf1(name(Transaction),isa(Parents),Tran_properties)).

assert_transaction2 :-
      nf1(name(Transaction),isa(Parents),Tran_properties),
```

```
      assert(nf2(name(Transaction),isa(Parents),Tran_properties)),
      retract(nf1(name(Transaction),isa(Parents),Tran_properties)).

% ------------------------------------------------------------------------
--
% The procedure handling the rewrite rules are used to replace
not_founded
% facts by other facts based on their purpose

check_for_rewrite_rule(Proc,New_Proc) :-
      rewrite_rule(List_of_purp,New_facts),
      write('Trying to replace not founded facts using rewrite rule...'),
      nl,nl,
      apply_rewrite_rule(List_of_purp,Proc,New_ProcA),
      write('Success applying rule replacing: '),
      write(List_of_purp),nl,
      write(' by: '),write(New_facts),nl,nl,
      append(New_ProcA,New_facts,New_Proc).

check_for_rewrite_rule(Proc,Proc) :-
      write('Unable to find adequate rewrite rule...'),nl,nl,
      flush,
      write('Hit any key to continue...'),
      get0_noecho(X),nl,nl.

apply_rewrite_rule([Purpose|Tail],Proc,New_proc) :-
      member(not_founded(Feature),Proc),
      purpose(Feature,Purpose),
      del_feature(not_founded(Feature),Proc,New_procA),
      apply_rewrite_rule(Tail,New_procA,New_proc).

apply_rewrite_rule([],List,List).

% ------------------------------------------------------------------------
--
% this is the procedures used to find previous cases and put them in
% a list from which a plausible transaction can be built
% previous case
fc(L) :- findall(C,applicable_prev_case(C),L).

applicable_prev_case(

      [Case_name,Common_feature_list,Remainder,_,_,Unused_feature_list])
:-
      case(name(Case_name),isa(Case_parents),Case_properties),
      [!member(precondition(Case_prec),Case_properties),
      member(procedure(Case_proc),Case_properties),
      append(Case_prec,Case_proc,Case_features),
      check_feature_list(Case_features,Common_feature_list,Remainder),
      findall(A,unused_feature(A),Unused_feature_list),
      reset_used_features!].

check_feature_list([C_F|Tail1],[C_F|Tail2],Remainder) :-
      copy_term((C_F :- true),(C_F_A :- true)),
      [C_F_A],
      assert(feature_used(C_F_A)),
      check_feature_list(Tail1,Tail2,Remainder).

check_feature_list([C_F|Tail1],Common_feature_list,[C_F|Tail2]) :-
```

```
      check_feature_list(Tail1,Common_feature_list,Tail2).

check_feature_list([],[],[]).

build_new_trans_from_cases(

      [[_,Case_name,Common_features,Remainder,_,_,Features_not_used]|Tail
E],
            New_frame_name,
            Parents,
            [precondition(Prec),procedure(Proc)]) :-
      check_for_coverage_of_TE_features(Common_features),
      case(name(Case_name),
            isa(ParentsA),
            Case_tran_properties),
      write('Building a new frame using previous case: '),
      write(Case_name),nl,nl,
      Case_name =.. [Tran_pred_name|Tail],
      training_example(Name),
      Name =.. [New_frame_pred_name|Tail2],
      New_frame_name =.. [New_frame_pred_name|Tail],
      adapt_transaction_prec_and_proc(
            case,
            Common_features,
            Features_not_used,
            Common_features_used,
            Case_tran_properties,
            Transaction_propertiesA),
      flush,
      write('Hit any key to continue...'),
      get0_noecho(X),nl,nl,
      remove_common_features_and_score(Common_features_used,TailE,Pruned_
tail),
      assign_score(Pruned_tail,Rescored_tail),
      quicksort_expl_list(Rescored_tail,Sorted_case_list),
      build_new_trans_from_cases(
            Sorted_case_list,
            New_frame_name,
            ParentsB,
            Transaction_propertiesB),
      member(precondition(PrecA),Transaction_propertiesA),
      member(procedure(ProcA),Transaction_propertiesA),
      member(precondition(PrecB),Transaction_propertiesB),
      member(procedure(ProcB),Transaction_propertiesB),
      merge_list(ParentsA,ParentsB,Parents),
      append(PrecA,PrecB,Prec),
      append(ProcA,ProcB,Proc).

build_new_trans_from_cases(

      [[_,Case_name,Common_features,Remainder,_,_,Features_not_used]|Tail
E],
            Frame_name,
            Parents,
            Properties) :-
      build_new_trans_from_cases(TailE,Frame_name,Parents,Properties).

build_new_trans_from_cases(
            [],
```

```
            Frame_name,
            [],
            [precondition([]),procedure([])]).
%-------------------------------------------------------------------
% the find_transaction_name procedure is used to find the name
% of a transaction that was used to generate a partial explanation

find_transaction_name(
      [subtree(transaction(_),[subtree(Transaction,Subtree)])|Tail],
      Transaction).

find_transaction_name(
      [subtree(Transaction,Subtree)|Tail],Transaction).


%     procedure append
%     ----------------
%
append([],L,L).

append(L,[],L).

append([X|L1],L2,[X|L3]) :-
     append(L1,L2,L3).


%     membership predicate
%
%
member(X,[]) :- !, fail.

member(X,[X|Tail]).

member(X,[Head|Tail]) :-
     member(X,Tail).


%-------------------------------------------------------------------
% this is quicksort to sort the list of partial_explanations

quicksort_expl_list([],[]).

quicksort_expl_list([[Score,A,B,C,D,E,F]|Tail],Sorted) :-
      split([Score,A,B,C,D,E,F],Tail,Small,Big),
      quicksort_expl_list(Small, Sorted_small),
      quicksort_expl_list(Big, Sorted_big),
      conc(Sorted_small,[[Score,A,B,C,D,E,F]|Sorted_big],Sorted).

split([Score,A,B,C,D,E,F],[],[],[]).

split([ScoreX,X1,X2,X3,X4,X5,X6],
      [[ScoreY,Y1,Y2,Y3,Y4,Y5,Y6]|Tail],[[ScoreY,Y1,Y2,Y3,Y4,Y5,Y6]|Small
],Big) :-
      ScoreX < ScoreY, !,
      split([ScoreX,X1,X2,X3,X4,X5,X6],Tail,Small,Big).

split([ScoreX,X1,X2,X3,X4,X5,X6],[[ScoreY,Y1,Y2,Y3,Y4,Y5,Y6]|Tail],Small,
      [[ScoreY,Y1,Y2,Y3,Y4,Y5,Y6]|Big]) :-
      split([ScoreX,X1,X2,X3,X4,X5,X6],Tail,Small,Big).
```

```
conc([],L,L).

conc([X|L1],L2,[X|L3]) :-
      conc(L1,L2,L3).

%-------------------------------------------------------------------
% this procedure is used to write the partial explanations after
% they have receive a score and they are sorted

write_expl_with_score([[Score,_,_,_,Tree,_,_]|Tail]) :-
      find_transaction_name(Tree,Transaction_name),
      tab(3), write('Score: '),write(Score),tab(3),
      write('Transaction name: '),write(Transaction_name),nl,
      write_expl_with_score(Tail).

write_expl_with_score([]) :- nl.

%-------------------------------------------------------------------
% this procedure is used to write the cases after
% they have receive a score and they are sorted

write_cases_with_score([[Score,Name,_,_,_,_,_]|Tail]) :-
      tab(3), write('Score: '),write(Score),tab(3),
      write('Case name: '),write(Name),nl,
      write_cases_with_score(Tail).

write_cases_with_score([]) :- nl.

%-------------------------------------------------------------------
% this procedure writes the new frame

write_new_frames :-
      nf(name(Transaction),isa(Parents),Tran_properties),
      write('The new transaction frame is:'),nl,nl,
      write('Name: '),write(Transaction),nl,
      write(' Parents: '),write(Parents),nl,
      write_properties(Tran_properties),nl,nl.

%-------------------------------------------------------------------
% this procedure writes the properties of a frame using a more
% readable format

write_properties([Property|Tail]) :-
      Property =.. [Name|Rest],
      tab(3),write(Name),write(':'),nl,
      write_property_list(Rest),
      write_properties(Tail).

write_properties([]).

write_property_list([Head|Tail]) :-
      write_property_sublist(Head),
      write_property_sublist(Tail).

write_property_list([]).

write_property_sublist([Head|Tail]) :-
      tab(5),write(Head),nl,
```

```
      write_property_sublist(Tail).

write_property_sublist([]).

%-------------------------------------------------------------------------
% merge list takes two lists and creates a third one
% combining elements from both lists without duplication

merge_list([],L,L).

merge_list([X|L1],L2,[X|L3]) :-
      \+ member(X,L2),
      merge_list(L1,L2,L3).

merge_list([X|L1],L2,L3) :-
      merge_list(L1,L2,L3).
```

```
%-----------------------------------------------------------------------
-
% file: prevexp.ari
% Module of LISE (Learning In Software Engineering)
%
% (c) Copyright 1990 by Jean Genest, All Rights Reserved
%-----------------------------------------------------------------------
-

% This file contains the precedures used to store and retrieve a
% new rule resulting from an explanation

ec :- edit(prevexp).

% this procedure stores the successful explanation

keep_successful_explanation :-
      explanation([Goal,Result,[],Tree,Message]),
      assert(past_explanation([Goal,Result,Tree])).

keep_successful_explanation.

% this procedure is called before EBL is applied to a new training
% example. It search for a previous explanation which would explain
% the training example.

solve_using_previous_explanation :-
      past_explanation([Goal,Result,Tree]),
      copy_term((Goal :- Result),(GoalA :- ResultA)),
      training_example(Trg),
      functor(Trg,Trg_pred,Arity),
      functor(Goal,Trg_pred,Arity),
      apply_explanation(Result,Messages), nl,
      write('Solved using previous explanation'), nl,
      write('--------------------------------'), nl,
      assert(explanation([GoalA,ResultA,[],Tree,Messages])),
      findall(A,explanation(A),List_of_expl),
      write_successful_expl(List_of_expl).

% Only one type of chaining is allowed on the rule derived from
% the previous explanation. The type of chaining allowed is
% to go down the concept hierarchy. For example, if the learned
% rule requires that a person has an account, then the training
% exampl could contain a saving-account, which is a specialization
% of account
% This was programmed to address the utility problem

apply_explanation([A|B],Messages) :-
      solve_term(A,MessageA),
      apply_explanation(B,MessageB),
      append(MessageA,MessageB,Messages).

apply_explanation([A],MessageA) :-
      solve_term(A,MessageA).

apply_explanation([],_).

solve_term(A,[]) :-
      A.
```

```
solve_term(inst_of(Variable,Constant),
           [message_on_hierarchy(Subconcept,Constant)|Message]) :-
      frame(name(Subconcept),isa(Superconcepts),_),
      member(Constant,Superconcepts),
      solve_term(inst_of(Variable,Subconcept),Message).

message_on_hierarchy(Class_from_TE,Class_from_previous_exp) :-
      write('MESSAGE concerning the previous explanation:'), nl,
      tab(3),write('The training example refered to '),
      write(Class_from_TE),nl,
      tab(3),write('which is a subclass of '),
      write(Class_from_previous_exp),
      write(' in the previous explanation.'),nl.
```

```
%-------------------------------------------------------------------------
-
% File wrtexpl.ari from LISE (Learning In Software Engineering)
%
% (c) Copyright 1990 by Jean Genest, All Rights Reserved
%-------------------------------------------------------------------------
-
%
% This file contains the procedures necessary
% to interact with the user.

ew :- edit(wrtexpl).

write_successful_expl([[Goal,Result,Remainder,Tree,Message]|Tail]) :-
      write_message_list_if_non_empty(Message),
      write('Goal: '), nl,
      write(Goal), nl,
      write('Result: '), nl,
      write(Result), nl,
      write('Remainder: '), nl,
      write(Remainder), nl,
%     write('Tree: '), nl,
%     write(Tree), nl,
      write_successful_expl(Tail).

write_successful_expl([]).

write_partial_expl(
            [[Goal,Result,Remainder,Tree,Message,Unused_features]|Tail])
:-
      nl,nl,
      write_message_list_if_non_empty(Message),
      write('PARTIALLY explained using transaction: '),
      find_transaction_name(Tree,Transaction),
      write(Transaction), nl, nl,
      write('Common features: '), nl,
      write(Result), nl,nl,
      write('Unexplained features of '),
      write(Transaction),
      write(' :'), nl,
      write(Remainder), nl,nl,
      write('Features of the training example not used in explanation
:'),
      nl,
      write(Unused_features), nl,nl,
%      write('Tree: '), nl,
%      write(Tree), nl,nl,nl,
      write('Hit any key to continue...'),
      get0_noecho(X), nl,nl,
            write_partial_expl(Tail).

write_partial_expl([]).

write_message_list_if_non_empty(Message) :-
      \+ Message = [],
      write('MESSAGE from LISE concerning the next explanation'), nl,nl,
      write_message_list(Message).

write_message_list_if_non_empty([]).
```

```
write_message_list([Message|Tail]) :-
      write_message(Message),
      write_message_list(Tail).

write_message_list([]).

write_message(Message) :-
      Message.
```

```
%-----------------------------------------------------------------------
-
% File: extrrule.ari from LISE (Learning In Software Engineering)
%
% (c) Copyright 1990 by Jean Genest, All Rights Reserved
%-----------------------------------------------------------------------
-
%
% This file contains the code required to extract the rules for EBG
% from the frames contained in the domain theory selected by the user
%
:- op(800,xfx,<-).
:- op(800,xfx,:).
:- op(800,xfx,or).

ex :- extract_rules_from_frames.
%
% Procedure called to extract the rules from the frames
%
extract_rules_from_frames :-
      frame(name(Name),isa(Parents), Property_list),
      [!extract_rule_from_frame(Name,Parents,Property_list)!],
      fail.

extract_rules_from_frames :-
      assert(nonop(transaction(A))).

%     the following procedure sends the frame selected to
%     the proper conversion predicate based on its content
%
extract_rule_from_frame(Name,Parent,Property_list) :-
      member(necessary_condition(_),Property_list),
      extract_rule(condition,Name,Parent,Property_list).

extract_rule_from_frame(Name,Parent,Property_list) :-
      member(action,Parent),
      extract_rule(action,Name,Parent,Property_list).

extract_rule_from_frame(Name,Parents,Property_list) :-
      is_subclass_of([Name],transaction(Person)),
      extract_rule(transaction,Name,Parents,Property_list).

extract_rule_from_frame(_,_,_).

%     The following procedure is used to convert a frame
%     into a rule
%
extract_rule(transaction,Name,Parents,Property_list) :-
      member(transaction(_),Parents),
      get_body(Property_list,Body),
      assert([Name] <- Body),
      assert(nonop(Name)),
      assert([transaction(_)] <- [Name]).

extract_rule(transaction,Name,Parents,Property_list) :-
      get_body(Property_list,Body1),
      append(Parents,Body1,Body),
      assert([Name] <- Body),
      assert(nonop(Name)),
```

```
      assert([transaction(_)] <- [Name]).

extract_rule(condition,_,_,Property_list):-
      member(necessary_condition(Clauses), Property_list),
      assert_clauses(Clauses).

extract_rule(action,Name,Parent,Property_list) :-
      get_body(Property_list,Body),
      assert([Name] <- Body),
      assert(nonop(Name)).

%     the case of disjunctions in the condition is handled with
%     the operator "or" in the frames
%
assert_clauses([[HeadA] <- BodyA] or Clauses) :-
      assert([HeadA] <- BodyA),
      assert_clauses(Clauses).

assert_clauses([[Head] <- Body]) :-
      assert([Head] <- Body),
      assert(nonop(Head)).

is_subclass_of([Name|Tail],Class) :-
      frame(name(Name),isa(Parents),_),
      member(Class,Parents).

is_subclass_of([Name|Tail],Class) :-
      frame(name(Name),isa(Parents),_),
      is_subclass_of(Parents,Class).

is_subclass_of([Name|Tail],Class) :-
      is_subclass_of(Tail,Class).

is_subclass_of([],_) :- fail.

% the following procedure extracts the body of the clause
% from a frame

get_body(Property_list,Body) :-
      get_property(precondition,Property_list,Precondition),
      get_property(procedure,Property_list,Procedure),
      append(Procedure,Postcondition,BodyA),
      append(BodyA,Precondition,Body).

get_property(Property_name,Property_list,Property) :-
      Property_predicate =.. [Property_name,Property],
      member(Property_predicate,Property_list).

get_property(Property_name,Property_list,[]).

append([],L,L).

append(L,[],L).

append([X|L1],L2,[X|L3]) :-
      append(L1,L2,L3).

member(X,[X|Tail]).
```

```
member(X,[Head|Tail]) :-
     member(X,Tail).
```

```
%-------------------------------------------------------------------------
-
% File banking.ari from LISE (Learning In Software Engineering)
%
% (c) Copyright 1990 by Jean Genest, All Rights Reserved
%-------------------------------------------------------------------------
-

% This is the K base containing the domain knowledge in format of
% frames for the application domain: Banking

eb :- edit(banking).

:- op(800,xfx,<-).
:- op(800,xfx,:).
:- op(800,xfx,or).

%     Each frame follows the following format:
%           the first slot is the name,
%           the second slot is the name of the parent, and
%           the third slot is a list of properties
%
%     the first part contains domain independant knowledge
%
%     object is the top concept
%

frame(name(object),
      isa([]),
      []).

%     the following part contain the transactions and the actions
%     define for the domain knowledge

frame(name(transaction(Person)),
      isa([object]),
      []).

frame(name(action),
      isa([object]),
      []).

frame(name(transaction_using_ATM(Person)),
      isa([transaction(Person)]),
      [precondition([bank_card(Person,Bank_card)]),
       procedure([insert_card(Bank_card,automated_teller)])]).

frame(name(deposit(Person,Amount)),
      isa([transaction(Person)]),
      [precondition([account(Person,Account)]),
       procedure([receive_money(Person,Amount),
                  credit(Account,Amount)])]).

frame(name(withdraw(Person,Amount)),
      isa([transaction(Person)]),
      [precondition([account(Person,Account)]),
       procedure([debit(Account,Amount),issue_money(Person,Amount)])]).

frame(name(withdraw_using_ATM(Person,Amount)),
```

```
      isa([withdraw(Person,Amount),transaction_using_ATM(Person)]),
      []).

frame(name(debit(Account, Amount)),
      isa([action]),
      [precondition([balance(Account,Balance1), Balance1 > Amount]),
       procedure([substract_from_balance(Account, Amount)])]).

frame(name(credit(Account,Amount)),
      isa([action]),
      [procedure([add_to_balance(Account,Amount)])]).

%     this part contains the entities of the conceptual model
%

frame(name(entity),
      isa([object]),
      []).

frame(name(person),
      isa([entity]),
      [name,
       address,
       phone_number]).

frame(name(client),
      isa([person]),
      [account,
       purpose(account,[identify_customer]),
       safety_box,
       credit_margin,
       purpose(credit_margin,[protect_BI_during_lending]),
       necessary_condition(
            [[client(Person)] <-
                 [account(Person, Account)]]
            or
            [[client(Person)] <-
                  [safety_box(Person, Safety_box)]])]).

frame(name(employee),
      isa([person]),
      [employee_number]).

frame(name(account),
      isa([entity]),
      [balance,
       purpose(balance,[protect_BI_during_withdrawal,
                        protect_CI_during_deposit])]).

frame(name(saving_account),
      isa([account]),
      []).

frame(name(checking_account),
      isa([account]),
      []).

% this is a "causal" net used to find analogies between the
% training example and the partial explanations generated to
```

```
% account for it

% actions which have the purpose: record_transaction

purpose(substract_from_balance(Balance,Amount),record_transaction).
purpose(add_to_balance(Balance,Amount),record_transaction).
purpose(record_loan(Client,Amount),record_transaction).

% actions which have the purpose: issue_money_to_client

purpose(issue_money(Person,Amount),issue_money_to_client).
purpose(issue_cheque(Person,Amount),issue_money_to_client).
purpose(issue_money_order(Person,Amount),issue_money_to_client).

% actions which have the purpose: receive_money_fm_client

purpose(receive_money(Person,Amount),receive_money_fm_client).
purpose(receive_cheque(Person,Amount),receive_money_fm_client).

% properties of entities which have the purpose: protect_bank_interest

purpose(balance(Account,Balance),protect_bank_interest ).
purpose(credit_margin(Client,Balance),protect_bank_interest).
purpose(assets(Client,Balance),protect_bank_interest).

% properties of entities which have the purpose: assets

purpose(gold_reserve(Client,MG),assets).
purpose(real_estate(Client,MG),assets).

% properties of entities which have the purpose: identify_client

purpose(account(Client,Account),identify_client).
purpose(identification(Client,valid_ID),identify_client).

% properties of entities which have the purpose: monthly_revenue

purpose(monthly_gnp(Client,MG),monthly_revenue).
purpose(monthly_income(Client,MG),monthly_revenue).

% some constrains on values are expressed using math. rel operator

purpose(_ < _, constraint).
purpose(_ > _, constraint).
purpose(_ = _, constraint).
purpose(_ is _ * _,constraint).
purpose(_ is _ + _,constraint).
purpose(_ is _ - _,constraint).


rewrite_rule([issue_money_to_client,receive_money_fm_client],[]).

% ------------------------------------------------
% previous cases
% ------------------------------------------------


case(name(cash_travelers_check(Person,Amount)),
     isa([transaction(Person)]),
```

```
      [precondition([identification(Person,valid_ID)]),
       procedure([issue_money(Person,Amount)])]).

case(name(cash_pay_check(Person,Amount)),
      isa([transaction(Person)]),
      [precondition([pay_check(Person,valid)]),
       procedure([issue_money(Person,Amount)])]).

case(name(loan_to_country(Country,Amount)),
      isa([transaction(Country)]),
      [precondition([
            account(Country,Account),
            gold_reserve(Country,Gold_reserve),
            value(Gold_reserve,Value_of_gold_reserve),
            Value_of_gold_reserve > Amount,
            monthly_gnp(Country,Monthly_GNP),
            installment(Amount,Interest_rate,Duration,Installment),
            Affordable_installment is Monthly_GNP * 0.3,
            Installment < Affordable_installment]),
      procedure([
            issue_money(Country,Amount)])]).

%     end of the "banking.ari"
file.
```

```
%-----------------------------------------------------------------------
-
% File trucking.ari from LISE (Learning In Software Engineering)
%
% (c) Copyright 1990 by Jean Genest, All Rights Reserved
%-----------------------------------------------------------------------
-

% This is another demo domain theory for LISE

:- op(800,xfx,<-).
:- op(800,xfx,:).
:- op(800,xfx,or).

frame(name(object),
      isa([]),
      []).

frame(name(transaction(Person)),
      isa([object]),
      []).

frame(name(action),
      isa([object]),
      []).

frame(name(hire_a_vehicle(Client,Date,Time_start,Time_end)),
      isa([transaction(Client)]),
      [precondition([
            authorize_request(Client)]),
       procedure([
            assign_a_vehicle(Date,Time_start,Time_end,Vehicle),
            vehicle_pickup(Client,Date,Time_start,Vehicle),
            vehicle_return(Client,Date,Time_end,Vehicle),
            prepare_vehicle_bill(Vehicle,Cost),
            send_bill(Client,Cost)])]).

frame(name(assign_a_vehicle(Date,Time_start,Time_end,Venicle)),
      isa([action]),
      [precondition([
            available_vehicle(Date,Time_start,Time_end,Vehicle)]),
       procedure([
            book_vehicle(Date,Time_start,Time_end,Vehicle)])]).

frame(name(prepare_vehicle_bill(Vehicle,Cost)),
      isa([action]),
      [precondition([
            get_meter_reading(Vehicle,Meter_reading)]),
       procedure([
            calculate_vehicle_cost(Vehicle,Meter_reading,Cost)])]).

purpose(assign_a_vehicle(Date,Time_start,Time_end,Driver),assign_resource
s).
purpose(assign_a_driver(Date,Time_start,Time_end,Driver),assign_resources
).
purpose(available_driver(Date,Time_start,Time_end,Driver),
                  ensure_resource_availability).
purpose(available_vehicle(Date,Time_start,Time_end,Vehicle),
                  ensure_resource_availability).
```

```
purpose(book_driver(Date,Time_start,Time_end,Driver),
                    record_resource_commitment).
purpose(book_vehicle(Date,Time_start,Time_end,Vehicle),
                    record_resource_commitment).
purpose(vehicle_pickup(Client,Date,Time_start,Vehicle),
                    resource_issue).
purpose(send_driver(Client,Date,Time_start,Driver),
                    resource_issue).
purpose(vehicle_return(Client,Date,Time_end,Vehicle),
                    resource_returned).
purpose(driver_return(Client,Date,Time_end,Driver),
                    resource_returned).
purpose(calculate_vehicle_cost(Vehicle,Meter_reading,Cost),
                    calculate_cost).
purpose(calculate_driver_cost(Driver,Time_sheet,Cost),
                    calculate_cost).
purpose(get_meter_reading(Vehicle,Meter_reading),record_usage).
purpose(get_time_sheet(Driver,Time_sheet),record_usage).


      end of the "trucking.ari" file.
```