PUC

ISSN 0103-9741

Monografias em Ciência da Computação
41/00

Aiding the Construction of Libraries of Typical Plans

Antonio L. Furtado
Angelo E. M. Ciarlini

Departamento de Informática

PONTIFÍCIA UNIVERSIDADE CATÓLICA DO RIO DE JANEIRO
RUA MARQUÊS DE SÃO VICENTE, 225 - CEP 22453-900
RIO DE JANEIRO - BRASIL

PUC RIO - DEPARTAMENTO DE INFORMÁTICA ISSN 0103-9741

Monografias em Ciência da Computação, N° 41/00
Editor: Carlos J. P. Lucena November, 2000

Aiding the Construction of Libraries of Typical Plans *

Antonio L. Furtado
Angelo E. M. Ciarlini

* This work has been sponsored by the Ministério de Ciência e Tecnologia da Presidência da República Federativa do Brasil.

In charge of publications:
Rosane Teles Lins Castilho
Assessoria de Biblioteca, Documentação e Informação
PUC-Rio - Departamento de Informática
Rua Marquês de São Vicente, 225 - Gávea
22453-900 Rio de Janeiro RJ Brazil
Tel. +55 21 529-9386 Telex + 55 21 31048 Fax +55 21 511-5645
E-mail bib-di@inf.puc-rio.br

AIDING THE CONSTRUCTION OF LIBRARIES OF TYPICAL PLANS

Antonio L. Furtado
e-mail: furtado@inf.puc-rio.br

Angelo E. M. Ciarlini
e-mail: angelo@inf.puc-rio.br

Puc-RioInf.MCC41/00 November, 2000

Abstract: Databases able to represent, not only facts, but also *events* in the mini-world of the underlying information system can be seen as repositories of narratives about the agents and objects involved. The events treated in our approach are those attributed to executions of predefined application-oriented operations. This work addresses the identification of *typical plans* adopted by agents, by analysing a *Log* registering the occurrence of events, as represented by executions of such operations. The analysis is done by applying a previously formulated set of *goal-inference rules* to sequences of interrelated events, called *plots*, taken from the Log. The obtained **Library of Typical Plans**, together with the goal-inference rules, constitute the behavioural level of our proposed three-level conceptual schemas for the specification of information systems. A prototype Prolog implementation of the method for extracting typical plans is operational. A simple example is used to illustrate the discussion.

Keywords: Information Systems Design, Agents, Planning, Logic Programming

Resumo: Bancos de dados capazes de representar, não apenas fatos, mas também *eventos* no mini-mundo do sistema de informações subjacente podem ser vistos como repositórios de narrativas sobre os agentes e objetos envolvidos. Os eventos tratados em nossa abordagem são os que podem ser atribuídos a execuções de operações predefinidas orientadas para a aplicação. Este trabalho visa à identificação de *planos típicos* adotados pelos agentes, através da análise de um *Log* registrando a ocorrência de eventos, representados como execuções de tais operações. A análise é feita aplicando um conjunto previamente formulado de *regras de inferência de objetivos* a seqüências de eventos interrelacionados, denominadas *enredos*, tomadas do Log. A **Biblioteca de Planos Típicos** obtida, juntamente com as regras de inferência de objetivos, constituem o nível comportamental dos esquemas conceituais em três níveis que propomos para a especificação de sistemas de informação. Uma implementação experimental em Prolog do método para extrair planos típicos encontra-se em funcionamento. Um exemplo simples é utilizado para ilustrar a discussão.

Palavras-chave: Projeto de Sistemas de Informação, Agentes, Planejamento, Programação em Lógica.

1. Introduction

The initial emphasis of the database approach to the conceptual specification of information systems was mostly on the static description of objects and their properties. At a later stage, however, attention was also given to functional characteristics [BCN]. Theoretical and practical work at these two complementary levels has led to entity-relationship schemes, object-oriented classes, and workflows, among other important contributions. More recently, there has been a growing realization that the specification of an information system must also consider the *agents* [MC, MCY, KS] which will eventually put it to use. What agents do is not fortuitous; they organize *plans* in an attempt to reach specific *goals*. In turn, goals arise when certain *situations* occur. Besides adding to the definition of objects a characterization of their functional aspects, a third stage of specification is therefore needed, where agents and their expected interactions are modelled. Informally speaking, agents cause the occurrence of *events* affecting the existence and various properties of entities in the mini-world of a given information system. And what they make happen in this mini-world, which as a consequence of their actions traverses a series of intermediate states, can be viewed, borrowing from traditional literary terminology, as *narratives*.

Accordingly, we specify schemas at three successive levels. The first is the *static level*, where the types of facts about entities to be stored in the database are declared, according to the Entity-Relationship model extended with is-a hierarchies for entity types. Secondly, *application-oriented operations* are defined, in a STRIPS-like formalism [FN], to provide the *dynamic level*. A third level, the *behavioural level*, is added, in order to model the reason why operations are executed and the way they are typically combined and performed. The behavioural level is composed of *goal-inference rules* and a **Library of Typical Plans.** Each goal-inference rule declares, for an agent or class of agents, that the occurrence of a certain situation tends to motivate the agent to pursue a given goal. Typical plans, consisting of partially ordered sets of one or more executions of the application-oriented operations, represent the expected patterns of database usage by the various agents, towards the achievement of their goals.

The availability of a **Library of Typical Plans** enormously increases the understanding of strategies and policies habitually adopted by agents, and even enables the use of plan-recognition algorithms to detect, by matching against the **Library** observed actions of an agent, whether such actions fit in some known plan or plans. Also, when a typical plan is detected, its associated goal can be recognized, which helps predicting possible future events. In [CF], we describe a tool in which plan-recognition and plan-generation are combined in order to simulate both typical and non-typical interactions of database agents.

This three-level specification approach is especially useful if the database implemented provides the two following features:

- updates can only be performed through the execution of the application-oriented operations introduced at the dynamic level;

- each execution of an operation triggers the insertion in a **Log** of a record containing the name and arguments of the operation executed, together with a time-stamp indicating the moment of execution.

A time-stamp-ordered sequence of records of executions of interrelated operations, extracted from the Log, clearly corresponds to a sequence of events, which justifies calling one such sequence a ***plot*** [CF,Sg]. We mentioned above the notion of narratives happening in the context of an information system. Plots can then be interpreted as summaries of such narratives. As demonstrated in a companion paper [FC], if a *text-generator* is available, plots can be interpreted to produce natural language answers to queries like: "What happened to Mary between time instants t1 and t2?".

But the thrust of the present paper is how to use plots to help formulating the behavioural level of conceptual specification. As a first step towards this objective, we assume that the goal-inference rules have been introduced at a preliminary stage. How to discover the rules is, of course, a difficult knowledge-discovery task [MCP, Ko], which we are investigating separately. Once the static and dynamic levels have been specified, and the goal-inference rules are available, we proceed to a trial phase, where the prospective agents are called to operate on a prototype implementation of the information system, thereby allowing the Log to grow to a size estimated large enough to constitute a sample. Then our tool, called **BLIB**, analyses a series of plots taken from the Log, on the basis of the goal-inference rules, in order to identify typical plans toward such goals and, with the designer's participation, to build a **Library of Typical Plans**.

The paper is organized as follows. Section 2 presents the three-level modelling concepts, emphasizing the visualization of plots as the result of plans of the various agents. Section 3 describes the method for obtaining typical plans from plots taken from the Log. Section 4 contains concluding remarks, pointing out aspects where certain tentative decisions adopted in the implemented prototype may be revised, by considering different alternatives. A small example is used throughout the paper to illustrate how the process works. For a more formal treatment of our modelling approach see [CVF]; other related formalisms can be found in [CL, MHL, MS]. The appendices show, in the notation of the Prolog tool, the schemas and input plots for running an extended version of the example (Appendix 1), and the Library of Typical Plans as obtained initially (Appendix 2) and after restructuring (Appendix 3); the implemented algorithm for computing the most specific generalization of plans is also included (Appendix 4).

2. Three-level specifications

The concepts used at each level will be introduced with the help of the very simple example of a Company Alpha's database. Schemas are specified, at each level, in a notation compatible with logic programming.

2.1. The static level

At the static level, *facts* are classified according to the Entity-Relationship model. Thus, a fact may refer either to the existence of an entity instance, or to the values of its attributes, or to its relationships with other entity instances. Entity classes may form an *is-a* hierarchy. Entities must have one privileged attribute, which identifies each instance at all levels of the is-a hierarchy. Moreover, we shall restrict ourselves to single-valued attributes and binary relationships without attributes. All kinds of facts are denoted by *predicates*.

The example static schema — given in Figure 1 — includes, among the entity classes, person, company, and course; in addition, class employee is a specialization of person, and client a specialization of company. The identifying attributes are name (for person, and consequently also for employee), denomination (company and client) and title (course). For the attribute level (of employee) there are only two possible values: 1 and 2. Account_status is an attribute of client, referring to the status of the client's account, whose only value that will concern us here, because of its criticality, is "inactive". Relationships serving and dissatisfied_with are defined between employees and clients; employees and courses are related by taking.

With respect to onomastic criteria, notice that nouns are used to name entity classes (e.g. person) and attributes (e.g. level). For relationships, we favour past or present participles (e.g. serving). Examples of predicate instances representing facts are: (a) entity instance: "person('Mary')"; (b) attribute of entity: "level('Mary',1)"; (c) relationship: "serving('Mary','Beta')".

The set of all predicate instances of all types holding at a given instant constitutes a *state*. In temporal database environments [Oz], one can ask whether or not some fact F holds at a state S associated with a time instant t.

```
dbowner('Company Alpha').

% Facts

entity(person, name).

entity(employee).
is_a(employee, person).
attribute(employee, level).

entity(company, denomination).

entity(client).
is_a(client,company).
attribute(client, account_status).

entity(course, title).

relationship(serving, [employee, client]).

relationship(dissatisfied_with, [client, employee]).

relationship(taking, [employee, course]).
```

Fig. 1: static sub-schema

2.2. The dynamic level

The dynamic level covers the *events* happening in the mini-world of interest. A real world event is perceived in a temporal database environment as a *transition* between database states. Our dynamic level schemas — figure 2, for the current example — specify a fixed repertoire of *operations*, whose execution provides the only kind of admissible events, hence the only way to cause state transitions [FuN]. Accordingly, from now on we shall equate the notion of event with the execution of one of these operations.

As in the STRIPS formalism [FN], each operation is defined through its signature, pre-conditions, and post-conditions or effects. Both pre-conditions and effects are expressed in terms of facts, thus establishing a connection with the static level. Pre-conditions are conjunctions of positive (or negated) facts, which should hold (or not hold) before execution, whereas effects consist of facts added and/or deleted by the operation.

When defining the signature of an operation, we declare the type of each parameter (which implicitly imposes a preliminary pre-condition to the execution of the operation) and its semantic role, borrowing from Fillmore's case grammars [Fi], a major contribution from the field of Linguistics. From the cases proposed by Fillmore, we retained *agent* (denoted by the letter "a") and *object* ("o"); we found convenient to denote the other cases (e.g. *beneficiary*, *instrument*, etc.) by some preposition able to suggest the role when used as prefix. The agent is, of course, whoever is in charge of executing the operation. In our example, operation complain is the only one whose definition indicates the agent explicitly. If none of the parameters is indicated as playing the role of agent, the database owner is assumed by default to have the initiative. Thus the clause:

```
oper(replace(E1,E2,C),
  [employee/o, employee/by, client/for]).
```

allows us to interpret the event "replace(Mary,Leonard,Beta)" unambiguously as:

- Company Alpha replaces employee Mary by employee Leonard for client Beta.

The other clauses defining the operation (cf. figure 2) give its preconditions and effects. As a consequence of these clauses, as the reader can verify, this particular replace event will indeed produce the state transition below, whose net effect is that, in state **sj**, Leonard, instead of Mary, is serving Beta:

si		**si**
employee('Mary').		employee('Mary')
employee('Leonard')	⟶	employee('Leonard')
client('Beta').		client('Beta').
serving('Mary','Beta').		serving('Leonard','Beta').

The other operations make it possible for company Alpha to sign a contract with a company (so as to make it one of its clients), to hire a person as employee with initial level 1, to assign an employee to the service of a client, to enroll an employee in a training course, to promote an employee by raising the level to 2, and to fire an

employee. To clients it is allowed to formally complain about the service rendered by the assigned employee, with the contractual effect of suspending all business transactions (account_status = "inactive").

Pre-conditions and effects are usually tuned in a combined fashion, aiming at the enforcement of integrity constraints. It can be shown that the integrity constraints below, among others, will be preserved if, in consonance with the abstract data type discipline, the initial database is consistent and these pre-defined operations are the only way to cause database transitions:

- an employee can serve at most one client and a client can be served by at most one employee(i.e. serving is a 1-1 relationship);
- an employee can only be fired if currently not serving any client;
- to have a level raise, an employee must be serving a client whose account is not inactive.

Verbs are employed to name the operations, possibly with trailing prepositions or other words or particles, separated by underscore.

```
% Operations

oper(sign_contract(C), [company/ with]).
added(sign_contract(C), client(C)).

oper(hire(E), [person/ o]).
added(hire(E), (employee(E), level(E, 1))).

oper(assign(E,C), [employee/ o, client/ to]).
added(assign(E,C), serving(E,C)).
precond(assign(E,C), ((not serving(E,C1)), (not serving(E1, C)))).

oper(enroll(E,T), [employee/ o, course/ in]).
added(enroll(E,T), taking(E,T)).
deleted(enroll(E,T), (dissatisfied_with(C,E),
  account_status(C,inactive))).
precond(enroll(E,T), (serving(E,C), not taking(E,T1))).

oper(promote(E), [employee/ o]).
added(promote(E), level(E,2)).
deleted(promote(E), level(E,1)).
precond(promote(E),
  (serving(E,C), not dissatisfied_with(C,E),level(E,1))).

oper(replace(E1,E2,C),
  [employee/ o, employee/ by, client/ for]).
added(replace(E1,E2,C), serving(E2,C)).
deleted(replace(E1,E2,C), serving(E1,C)).
precond(replace(E1,E2,C), (serving(E1,C), not serving(E2,C1))).

oper(fire(E), [employee/ o]).
deleted(fire(E), (employee(E), level(E,N),
  dissatisfied_with(C,E), account_status(C,inactive))).
precond(fire(E), (not serving(E,C))).

oper(complain(C,E), [client/ a, employee/ about]).
added(complain(C,E), (dissatisfied_with(C,E),
  account_status(C,inactive))).
precond(complain(C,E), serving(E,C)).
```

Fig. 2: dynamic sub-schema

2.3. The behavioural level

Carefully designed application-oriented operations enable the various agents to handle the database in a consistent way. The question remains of whether they will coexist well with a system supporting such operations, and, if so, what actual usage patterns will emerge. Ideally, the designers of an information system should try to predict how agents will behave within the scope of the system, so as to ensure that the specification at the two preceding levels is adequate from a *pragmatic* viewpoint. The ability to make predictions about behaviour is also crucial for decision-making based on simulations of future events.

To model the reactions of prospective agents, our behavioural sub-schema for the Company Alpha example — given in figure 3 — contains a few illustrative *goal-inference rules*, plus some *typical plans* (represented as *complex operations*).

A goal-inference rule has, as antecedent, some *situation* which, if observed at a database state, will arouse in a given agent the impulse to act in order to reach some *goal*. Two rules refer to Company Alpha, the database owner. The first one indicates that, if employee E is not currently serving any client, Alpha will want that E cease to be an employee. The goal in the second rule is that Alpha will do an effort to placate any client C who, being dissatisfied with the employee assigned to its service, has assumed an inactive status. (Notice, incidentally, that "keeping a client happy" is, in the terminology of [MC], a *soft goal*, i.e. an imprecisely defined objective; in our example, it assumes a more firm aspect through its dependence on the concrete consideration of the account_status attribute). A goal is indicated for employees: if E1 has merely level 1, whilst some other employee E2 has been raised to level 2, then, presumably moved by emulation, E1 will want to reach this higher level.

The specification of behaviour is complemented by a **Library of Typical Plans**. A *typical plan* is a description of how an agent (or class of agents) usually proceeds towards some goal. It consists of either a set of partially ordered operations or plans, or of a set of specialized alternative plans able to achieve the goal. Plans of both kinds are expressed in the **Library** as complex operations. Let us call the operations introduced in the previous section *basic operations*. Then, a *complex operation* can be defined from the repertoire of basic operations (or from other complex operations, recursively) by either composition (part-of hierarchy), giving origin to ***composite*** operations, or by generalization (is-a hierarchy), yielding ***generic*** operations. In case of composition, the definition must specify the component operations and the ordering requirements, if any (noting that we allow plans to be *partially-ordered*). In case of generalization, the specialized operations must be specified.

In our example, complex operation renovate_assistance is composed of basic operations hire, replace and fire. In turn, complex operation improve_service generalizes basic operation enroll and complex operation renovate_assistance. (A minor technical detail: the fact "serving(E,C)", introduced by ":" in the first is_a clause is needed to identify E, which is not in the parameter list of improve_service). Notice that the two (specialized) forms of improve_service have, among others, the effect of removing the undesired de-activation

of a client's account. Both can be regarded as reflecting customary strategies (typical plans) of Company Alpha to placate a complaining client: it either trains the faulty employee or "renovates" the manpower offered to the client. And therefore both are adequate to achieve the goal expressed in the second rule of figure 3.

Complex operation advance_the_career has an apparent peculiarity, in that it deviates from the usual norm of plan-generation algorithms, whereby operations are chained together exclusively as needed for the satisfaction of pre-conditions. Here, however, the component operation enroll is not required for satisfying a pre-condition for promote (except in the special case where training is the chosen way to remove the effects of a pending complaint). Our notion of typical plans, similarly to scripts [SA], allows however a looser interpretation. A plan is typical if it reflects the usages and policies, imposed or not by rational reasons, that are observed (or anticipated) in the real-world environment. Thus, we may imagine that the employer, company Alpha, is sensed to be more favourable to promoting employees who, even in the absence of complaints against their service, seek the training program.

Through an analysis of the component or alternative operations of a given complex operation, it is possible to determine the pre-conditions for its execution. It is also possible to identify, among the facts that necessarily hold (or do not hold) after execution, a goal to be achieved by the operation. Given, as input, observations concerning the execution of a few operations, the **Library of Typical Plans** can be used by plan-recognition algorithms to detect which possible plans the agents may be trying to perform. The recognized plan (or plans) can then be used in simulations of future events. Also, plan detection implies the detection of the respective goals and pre-conditions. Once the pre-conditions are obtained, they can be analysed to check whether the plan can be completed. In turn, the detected goals can serve as input to plan-generation algorithms to produce still other plans able to achieve them, which may also be worthy to be tried in simulation runs.

```
% Goal-inference rules and typical plans

gi_rule('company Alpha', (employee(E), not serving(E,C)), not employee(E)).
gi_rule('company Alpha', (serving(E,C), account_status(C,inactive)),
  not account_status(C,inactive)).

gi_rule(employee(E1), (level(E1,1),level(E2,2)),
  level(E1,2)).

op_complex(renovate_assistance(C,E2,E1),
  [client/ to, person/ with, employee/ 'in the position of']).
components(renovate_assistance(C,E2,E1),
  [f1: hire(E2), f2: replace(E1,E2,C), f3: fire(E1)],
  [f1-f2, f2-f3]).

op_complex(advance_the_career(E), [employee/ of]).
components(advance_the_career(E),
  [f1: enroll(E,C), f2: promote(E)],
  [f1-f2]).

op_complex(improve_service(C), [client/ for]).
is_a(enroll(E,T), improve_service(C): serving(E,C)).
is_a(renovate_assistance(C,E2,E1), improve_service(C)).
```

Fig. 3: behavioural sub-schema

In a previous work [CF] we have used the three-level schemata for *simulation* purposes, with the help of a *plan-recognition / plan-generation* method, combining algorithms introduced in [Ka] and [YTW], and supported by a Prolog prototype. In that context, a simulated process is enacted, whereby, at each state reached, the goal-inference rules are applied to propose goals by detecting situations affecting each agent. For attempting to fulfil such possibly collaborating or conflicting goals, *plans* are taken from a **Library of Typical Plans** or built by the plan-generator component. In turn, the execution of such plans leads to other states, where the goal-inference rules are again applied, and again plans are obtained and executed, so that the multistage process will continue until it reaches a state where no more goals arise, or until it is arbitrarily terminated.

3. Using goal-inference rules to extract plans from plots

More often than not it is difficult for the designers of real-life information systems to anticipate the usage patterns that will emerge after the system is delivered to operation. Hence, it may be necessary to postpone some design steps and interpose a trial phase, wherein agents are given access to a prototypical version of the system, with a Log of executed operations being recorded for later analysis. The missing design steps can then be undertaken with the benefit of the sample experimental evidence extracted from the Log.

In section 2.3. we were considering the formulation of:

(a) goal-inference rules, and
(b) typical plans (consisting of basic or complex operations).

Now suppose instead that, assuming a more realistic scenario, it was possible, by interviewing the prospective agents of an information system being designed, to achieve step (a) to a reasonable extent, whereas (b) could not be completed, since the persons consulted felt unable to predict beforehand how they would use the proposed basic operations. So, as suggested above, we resort to the trial phase strategy, allowing the agents to interact with a prototypical version until such time as the operation Log is sensed to contain enough data for a comprehensive analysis.

We proceed by extracting from the Log a series of plots. Each plot is a sequence of events, ordered by their time-stamps, where the first event occurred at a time instant t1 and the last event at t2. In other words, if the entire Log is regarded as a sequence of events, then a plot PL is the subsequence of the Log circumscribed to a given time interval t1..t2. To avoid excessively long plots, involving many disparate events, it is possible (and often useful), as shown in [FC], besides restricting plots to time intervals, to filter them so as to only retain events directly or indirectly related to certain specified objects. In the sequel, we shall assume that all plots to be processed have passed, whenever convenient, through this preliminary filtering step.

3.1. Interpreting plots to detect plans

Having isolated a number of plots to be used as input, we analyse them by applying the various goal-inference rules supposed to have been determined at a preliminary phase. Our method relies on the assumption that plots generally reflect the interaction of diverse plans — not always totally executed and successful with respect to the intended goals — undertaken by the various agents.

When considering the plots, differently from the context of our previous work (cf. end of section 2.3), we are not looking at simulation runs, but rather at observed actions, which may not be entirely rational. Hence, our use of goal-inference rules falls into an *abductive* mode of reasoning, as explained in the sequel. Assume that a rule R indicates that agent A, confronted with situation S, will have the desire to achieve a goal G. Now suppose that in the plot being examined an operation (or sequence of operations) O is present, with the effect of achieving goal G for A, and suppose further that, in the state before the execution of O, the motivating situation S prevailed.

We then formulate the *hypothesis* that the event can be explained by this rule R, i.e. that agent A executed (or was able to induce an authorized agent A' to execute) operation O because A previously observed the occurrence of S, being thereby motivated to achieve G. This kind of reasoning is no more than hypothetical, because there may exist other reasons (possibly expressed in other goal-inference rules) that may better explain why O was executed. So, each goal-inference rule helps us to suggest one *interpretation* for the events in a narrative.

Our Prolog prototype tool, **BLIB**, builds a **Library of Typical Plans** by examining a succession of plots taken from the Log. It begins by trying to extract from each such plot PL one or more plans, P, that can be **associated** with a goal-inference rule R = gi_rule(A,S,G), establishing that situation S motivates agent A to pursue goal G. A subsequence of events P' from PL is said to be associated with R if, prior to the execution of the first event in P', the situation S holds and, after the execution of the last event in P', a state is reached where G finally holds. Plan P is obtained from P' by a second more refined filtering process, which only keeps the events whose post-conditions contribute to G, plus, proceeding backwards, recursively, those that achieve pre-conditions of events already included in P. It should be stressed, in view of the preceding considerations, that a plurality of plans can be extracted from the same plot, and that the same plan can be associated with more than one goal-inference rule. Accordingly, all interpretations warranted by the existing rules are taken into account by the tool.

3.2. Overview of the library-construction process

The algorithm accumulates its output in a data structure, called the **ASG-Index** (from now on simply referred to as Index), which is organized as a table, whose entries correspond to each goal-inference rule gi_rule(A,S,G), defined on agent A, situation S, and goal G. Each entry of the table is consequently indexed by [A,S,G], and stores an (initially empty) list of operations, which can be either basic or complex. Each of these operations incorporates a specific plan associated with the

rule. At any instant, the Index contains, in its essential elements, a representation of the current stage of the **Library of Typical Plans** being constructed.

Consider, for instance, the plot:

```
PL = [s0, complain('Beta','Mary'), hire('Leonard'), hire('John'),
replace('Mary','Leonard','Beta'), fire('Mary')]
```

where s0 denotes the preceding database state. Now, consider the rule below, to be tentatively applied to PL:

```
gi_rule('company Alpha', account_status(C,inactive),
  not account_status(C,inactive)).
```

which says that, for agent A = Alpha, the situation S where the account of a client has become inactive induces the goal G of bringing it back to activity

When analysing PL to check whether or not the rule is applicable, the chaining of pre-conditions and effects in PL is verified by a conventional holds meta-predicate, which, incidentally, is the basis for simple plan-generators following STRIPS formalisms. A fact F holds after an operation O is executed at a state s_Q reached by executing a previous sequence of operations Q if either:

1. O is the pseudo-operation s0 and F belongs to the corresponding database state;
2. F is among the facts declared to be added by O, and the pre-conditions of O hold at s_Q;
3. F already held at s_Q and is not among the facts declared to be deleted by O.

Clearly, in case (1) the tool must have access to facts concerning the objects involved, holding at the temporal database state corresponding to s0, either directly retrieved by the tool itself or prefetched (as happens with the present version). To check facts at states reached along the execution of operations in PL, the holds meta-predicate simply resorts to the definitions of the operations. Notice that (2) and (3) make the process recursive (fixing the pre-conditions as sub-goals, or looking for F in the effects of the operations in Q), and that (3) is a standard solution for the frame problem (facts not affected by O continue to hold). By using holds we not only check coherence but also provide for the instantiation of some of the variables in the pre-conditions and effects that do not correspond to the parameters.

In this case, the rule is found to be applicable to PL, allowing the extraction of a sequence, which is readily refined to a plan P (by eliminating the irrelevant hire('John')):

```
P = hire('Leonard'),replace('Mary','Leonard','Beta'),fire('Mary')
```

Notice that S occurs as a consequence of complain('Beta','Mary') and that G holds immediately after fire('Mary').

Having extracted a plan, **BLIB** must decide about its possible impact on the Index. The plan may be simple, involving a single event, or compound. If the plan is compound, it is first put in a standard representation consisting of:

1) a set of tagged events;
2) a set of order dependencies, expressed as tag-pairs.

where the order dependencies are determined exclusively on the basis of the satisfaction of post-conditions by pre-conditions, and where dependencies deducible by transitivity are omitted. In standard representation, the compound plan P exemplified above becomes:

- set of tagged events:
```
f1: hire('Leonard'), f2: replace('Mary','Leonard','Beta'), f3:fire('Mary')]
```

- set of dependencies:
[f1-f2, f2-f3]

In the Index, all operations involving a compound plan are kept in this format, which is convenient for testing if a candidate plan Pi brings a novel contribution. The inclusion or not of Pi at an entry [A,S,G] depends on a comparison with the operations already in that entry. One possibility is that an operation Oj defined on a (simple or compound) plan Pj identical to Pi is already there — in which case nothing is done. Another case is that of similar plans. We say that two plans Pi and Pj are *similar* if, even with different parameter values and executed in a different order, they involve the same:

- number and type of events
- order dependencies
- co-designation/ non-co-designation schemes

Co-designation (or, respectively, non-co-designation) allows (forbids) the occurrence of the same value (constant or variable) in different parameter positions. Notice, for instance, that in the example above 'Mary' occurs in the first position of replace and in fire; a plan with 'John' in both places (or, say, X in both places) would meet the same co-designation requirement. To verify whether the order dependencies are the same, one looks for a renaming of the tags of one of the plans that can render the sets of order dependencies in the two plans identical.

Now, if we are considering a candidate plan Pi and there already exists an operation Oj with a similar plan Pj in Index, Pj will be replaced by the *most specific generalization* P* of Pi and Pj, whenever P* is more general (contains a larger number of variables) than Pj; otherwise Pi is discarded. Reliance on most specific generalization [Fu] is a fundamental feature of our approach; we strive to stay as close as possible to the evidence supplied by the plots, and thus, in particular, keep constants that tend to repeat (e.g. one certain course taken by all employees who later succeeded to be promoted).

The third case is that of a Pi with no similar plan in the [A,S,G] entry. If Pi is simple, it is immediately added to the entry. If compound, **BLIB** asks the designer's help (noting that he can decline, in which case Pi is discarded) to indicate the signature and parameter roles of the new *composite* complex operation Oi. The generated clauses for the example, to be stored in Index at the [A,S,G] entry are shown below:

```
op_complex(renovate_assistance('Beta','Leonard','Mary'),
  [client/ to, employee/ with, employee/ 'in the position of']).
components(renovate_assistance('Beta','Leonard','Mary'),
  [f1: hire('Leonard'), f2: replace('Mary','Leonard','Beta'),
   f3:fire('Mary')],
  [f1-f2, f2-f3]).
```

If the added basic or composite Oi is the first in the [A,S,G] entry, nothing else is done. Otherwise, **BLIB** regards the existence of more than one plan associated with the rule as an opportunity to create a *generic* complex operation. Accordingly, it asks the designer (who, again, can decline) to supply the signature of the complex operation. Suppose that, before introducing the new `renovate_assistance` operation, the respective entry already contained the basic operation `enroll`. With the designer's help, generic operation improve_service can be further added, with the following clauses being recorded in the entry (notice that all parameters are filled with variables, so that no propagated change will be needed in view of future detections of similar plans):

```
op_complex(improve_service(C), [client/for]).
is_a(enroll(E,T), improve_service(C): serving(E,C)).
is_a(renovate_assistance(C,P,E), improve_service(C)).
```

The current version of BLIB supports only one generic operation Og per entry. So, if Oi is introduced when there already exist two or more operations in the entry, and Og has already been introduced, then the definition of the existing Og is merely expanded through an additional is_a clause, relating Oi with Og.

Figure 4 below shows a fragment of the generated **Library of Typical Plans**. Following the conventions in Kautz's plan-recognition project [Ka], single arrows denote part-of links (composite operations) and double arrows are for is-a links (generic operations).

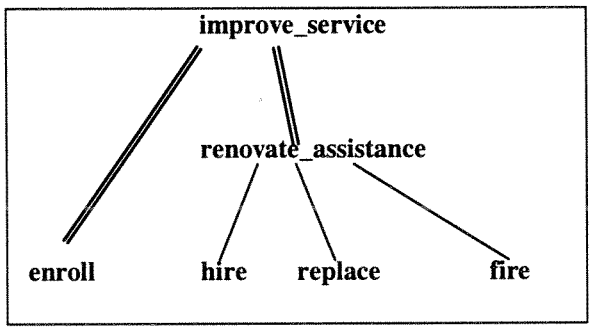

Fig. 4: fragment of the Library of Typical Plans

3.3. Reorganizing the Index

One software engineering requirement to be met by the organization of the **Library of Typical Plans** is that it should be conducive to a *modular architecture* for the later implementation of the operations. Multi-level composition naturally reflects in operational modules calling other modules on a shared basis, and multi-level generalization leads to the conditional choice of modules appropriate to each different case.

Yet, in its current version, the *first phase* of **BLIB** creates no more than one-level composition and generalization hierarchies, the only slightly more involved possibility being the presence of composite operations as alternatives of a generic operation, as exemplified in figure 4. Due to our concern to avoid complicated and time-consuming cases of propagation, we decided to keep this simple structure as long as new plots continue to be submitted as input. However, at any time after a batch of plots has been processed, **BLIB** can be called to execute a reorganization *second phase* over the Index (and, hence, over the represented structure of the **Library**), so as to produce certain improved multi-level hierarchies. If the acquisition of plans is resumed later, then the original restricted structure must be first reinstated (which is made possible by keeping a back-up copy prior to second phase runs).

The second phase of **BLIB** allows three kinds of restructuring, which are attempted in the indicated order:

1) multi-level generalizations;
2) generic operations as components of composite operations;
3) multi-level compositions.

Reflecting what often happens in practice, we expanded our example application, which resulted in new larger versions of the three schemas (not reproduced here, for space considerations), and processed additional plots. The second phase transformations were then executed. One instance of each kind is displayed below in diagrammatic form.

1) Suppose that entry [A1,S1,G1] has a generic operation Og1, and entry [A2,S2,G2] has a generic operation Og2, and that the entire set of alternatives of Og2 is a proper subset {O1, O2, ..., Ok} of the alternatives of Og1. Then, transformation (1) can be applied to replace, at the definition of Og1, all the k alternatives by a single occurrence of Og2.

Figure 5 illustrates the transformation. In the extended application, besides hiring regular employees (operation hire, renamed to hire_r), company Alpha was allowed to hire trainees (hire_t) and consultants (hire_c); notice that both regular employees and trainees are specializations of employee, whereas consultants are a separate category of manpower. At the first phase, operation hire_mp (hire manpower) was introduced as a generalization of hire_r, hire_t and hire_c, and operation hire_e (hire employee) as a generalization of hire_r and hire_t. After the transformation, hire_mp becomes a generalization of hire_e and hire_c.

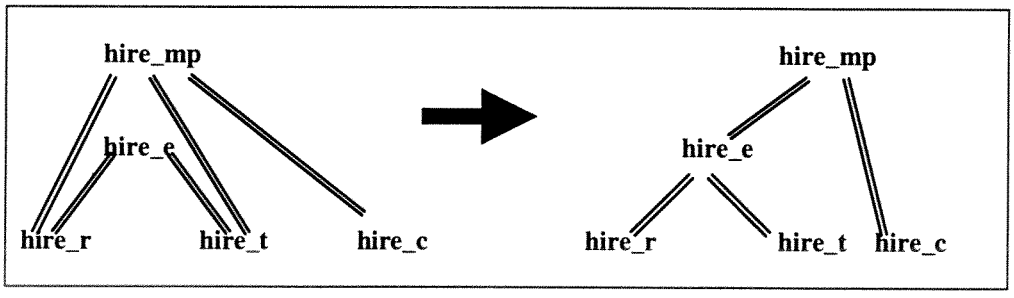

Fig. 5: multi-level generalization

2) The second transformation replaces a generic operation, whose alternatives are composite operations differing by a single component, by one composite operation with a generic component. Assume that entry [A1,S1,G1] has a generic operation Og1, and that all the alternatives O1, O2, ... , On generalized by Og1 are composite operations with the same number m of components, of which m-1 involve the same operations, and that the order dependencies are analogous. So, each Oj alternative essentially differs from the others by only one component Oji. Let the set {O1i, O2i, ... , Oni} of the dissimilar components of the n alternatives correspond exactly to the set of alternatives under another generic operation Og2, contained in a separate entry [A2,S2,G2] of the Index. Then, transformation (2) can be applied to replace the entire contents of entry [A1,S1,G1] by one composite operation, keeping for convenience the same name as Og1. The components of the new Og1 will result from the most specific generalization of the components of O1, O2, ..., On, to be computed after replacing by Og2 the dissimilar component Oji of each Oj.

Since employees (both regulars and trainees) cannot be fired as long as they stay associated with some client, they strive not simply to be hired but, as soon as possible, to become fully engaged through an assignment. The complex operation engage emerged at the first phase of the process with two specializations: engage_r and engage_t, for each kind of employee. As shown in figure 6, a considerably simpler structure results from adopting hire_e as a (generic) component of engage.

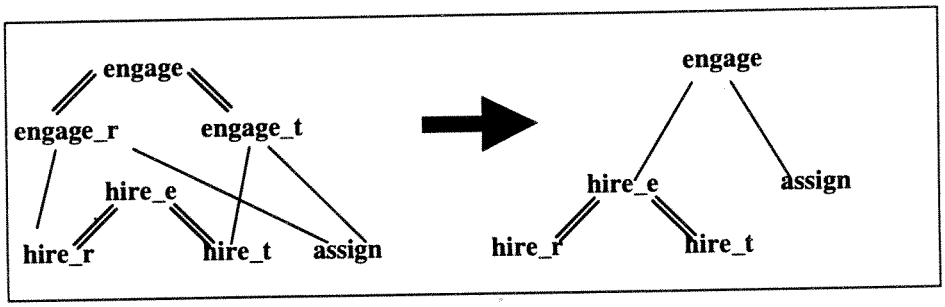

Fig. 6: generic operation as component of composite operation

3) Finally, assume that entry [A1,S1,G1] has a composite operation Oc1, and entry [A2,S2,G2] has another composite operation Oc2, such that the entire set of components $\Sigma 2 = \{O1, O2, \ldots, Ok\}$ of Oc2 is compatible, without imposing restrictions, with a proper subset $\sigma 1$ of the set $\Sigma 1$ of components of Oc1 (i.e. $\Sigma 2$ can be unified with $\sigma 1$, and the number of variables in $\sigma 1$ remains the same after unification). Suppose further that the order dependencies between the components of Oc2 are exactly those holding for those of subset $\sigma 1$ in the Oc1 definition. Transformation (3) then tries to replace, at [A1,S1,G1], all components of Oc1 in the $\sigma 1$ subset by a single component Oc2, and adjusts each order dependency [Oi-Oj] as follows, letting $\sigma 1^-$ be the complement subset $\Sigma 1 - \sigma 1$:

(a) if both Oi and Oj are in $\sigma 1^-$, keep [Oi-Oj];
(b) if Oi is in $\sigma 1$ and Oj in $\sigma 1^-$, replace [Oi-Oj] by [Oc2-Oj];
(c) if Oi is in $\sigma 1^-$ and Oj in $\sigma 1$, replace [Oi-Oj] by [Oi-Oc2];
(d) if both Oi and Oj are in $\sigma 1$, simply drop [Oi-Oj].

Cases (b) and (c) may obviously yield duplicates, which should be eliminated. A situation that is treated as an inconsistency, and causes the transformation to fail, is the simultaneous presence of [Oc2-Op] and [Op-Oc2] among the resulting order dependencies, which will arise whenever there existed dependencies [Oq-Op] and [Op-Or], with both Oq and Or in $\sigma 1$.

For company Alpha, to obtain a new client involves signing a contract and then providing assistance through the appointment of an employee. The chosen employee could be anyone previously hired or freed (by replace) from a previous assignment. But one possibly typical plan is to perform a new hiring specifically for immediate assignment to the new client, as indicated in composite operation obtain. On the other hand, the combination of hiring and assigning can be compactly expressed by composite operation engage. Figure 7 shows a restructuring over the part-of hierarchy, which parallels what is done with the is-a hierarchy in figure 5.

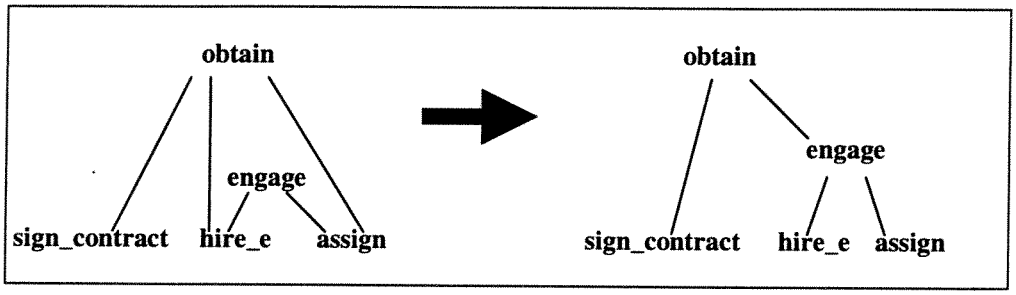

Fig. 7: multi-level composition

Note:

The order of application of the transformations is not irrelevant. As will be exemplified in the sequel, different final configurations may result from the adoption of different orders.

Suppose that, at the end of the first phase, composite operations obtain_t and engage_t have been derived, with the following components, where X is a variable:

obtain_t:
[f1: sign_contract('Beta'), f2: hire_t(X), f3: assign(X,'Beta')]

engage_t:
[f1: hire_t('John'), f2: assign('John','Beta')]

The the set s1 = {hire_t('John'), assign('John','Beta')}of components of engage_t unifies with the subset s2 = {hire_t(X), assign(X,'Beta')}, taken from the components of obtain_t. However, an attempt to apply transformation (3) will fail, because s1 (which has no variable) is less general than s2. We claim that it is reasonable that, based on the limited evidence of a single person appearing in the plot(s) leading to engage_t, we should not feel justified to use engage_t to generate a new version of obtain_t that could retain the same generality of the original version, allowing therefore the involvement of any person:

obtain_t:
[f1: sign_contract('Beta'), f2: engage_t(X,'Beta')]

But now suppose that obtain_r and engage_r had also been derived, as well as the generic operations obtain (combining the alternatives obtain_t and obtain_r) and engage (with alternatives engage_t and engage_r). Suppose further that the components of obtain_r and engage_r were as follows:

obtain_r:
[f1: sign_contract('Beta'), f2: hire_r(X), f3: assign(X,'Beta')]

engage_r:
[f1: hire_r(Y), f2: assign(Y,'Beta')]

Clearly, transformation (3) would work to produce a new obtain_r with components [f1: sign_contract('Beta'), f2: engage_r(X,'Beta')], but we can go farther than that if we begin with transformation (2), eliminating obtain_t and obtain_r and making obtain a **composite** instead of generic operation; likewise, transformation (2) will replace engage_t and engage_r by one composite operation engage. As said before, transformation (2) uses most specific generalization to combine the components of previously existing alternatives to form the components of the new composite operation. Since, in particular, as most specific generalization is applied to combine the components of engage_t and engage_r, the parameter value resulting from combining 'John' and Y is a variable, and recalling that transformation (2) replaces hire_t and hire_r by the generic hire_e (cf. figure 6), we have:

obtain:
[f1: sign_contract('Beta'), f2: hire_e(X), f3: assign(X,'Beta')]

engage:
[f1: hire_e(Y), f2: assign(Y,'Beta')]

and we can now apply transformation (3) with a much stronger effect, to end-up with:

obtain:
[f1: sign_contract('Beta'), f2: engage(X,'Beta')]

Thus, in this case at least, it is more convenient to apply transformation (2) before transformation (3).

4. Concluding Remarks

The present work is part of a larger research project, centred on the use of three-level schema specifications for a variety of purposes. For simulation of possible futures in the mini-world of information systems — as well as for the interactive generation of plots of narratives, belonging to real-life or literary genres —, we developed a framework, formally described in [CVF]. In conformity with the framework, a prototype tool was implemented [CF], called *Interactive Plot Generator* (**IPG**), which utilizes logic programming and constraint programming features to conduct simulation experiments, supporting goal-inference rules of greater generality than those exemplified here. The architecture of **IPG** is displayed in figure 8. Rectangular boxes represent modules, all of which, except the Rule Formulation module, have been implemented. The **BLIB** tool introduced in this paper corresponds to the main constituent of the (shaded) Library Construction module.

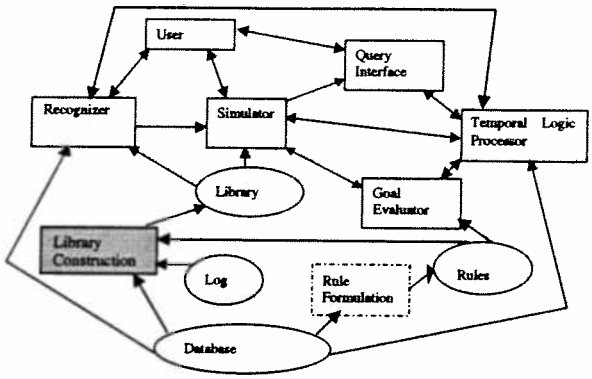

Fig. 8. General architecture of **IPG**

Library construction terminates by extracting, from the Index built by **BLIB**, the clauses defining the complex operations and, after adding complementary information (especially the derived pre-conditions and the associated goals, which constitute the main effects of the operations), composing the **Library** in the exact format required for access by the other **IPG** modules.

Even though **BLIB** is fully operational, some of our decisions concerning its implementation were only tentative, and may be revised as we experiment with an ampler variety of examples, ideally adapting the tool to work directly with databases of a realistic size. The following points deserve special mention.

All extracted plans formed by more than one operation are discarded if the user does not choose to give to it the status of a named composite operation. One may, instead, prefer to keep record of such non-used plans for future analysis.

At the restructuring phase we limited ourselves to a few transformations which we found relatively safe. Other transformations that are both safe and useful may be identified in the future, leading, for instance, to complex operations that are generic and, at the same time, contain components (as found in [Ka], and even in our full version of **IPG**). Also, the preferred order of application of the transformations was chosen in an attempt to carry the process as far as possible, as demonstrated in the specific case discussed at the end of the previous section, but a more systematic study of their mutual interactions is needed (and should be redone if new types of transformations are added).

The present approach exhibits an *extensional*, rather than an *intensional* bent. For instance, for the first transformation we check whether the set of specialized operations of an Index entry [A1,S1,G1] currently contains the set of specialized operations of another entry [A2,S2,G2]. But, in principle, this may follow from an implication R2 $\rightarrow$ R1 relating the corresponding goal-inference rules, with the consequence that R1 = gi_rule(A1,S1,G1) would be more widely applicable than R2 = gi_rule(A2,S2,G2). This optional approach would involve a logical comparison of the two situation-goal pairs.

Turning to a broader issue, recall that we assumed the goal-inference rules available before the typical plans. One may find more convenient to adopt a different strategy, possibly trying to identify the typical plans on the basis of criteria not depending on the rules. Moreover, being "typical" carries a notion of frequent occurrence, which indicates that statistic measures should be incorporated, at least as a confirmation criterion. With the continuation of the project we intend, regardless of the preferred strategy for detecting typical plans, to focus our attention on goal analysis and on methods and tools for the discovery of goal-inference rules.

References

[BCN] C. Batini, S. Ceri, S.B. Navathe. *Conceptual Database Design: an Entity-Relationship Approach*, Benjamin-Cummings, 1992.

[CF] A. E. M. Ciarlini and A. L. Furtado. "Simulating the interaction of database agents". In Proc. *DEXA'99 Database and Expert Systems Applications Conference*, Florence, Italy, 1999.

[CL] P. R. Cohen and H. J. Levesque. "Intention is choice with commitment". *Artificial Intelligence*, 42: 213-261, 1990.

[CVF] A.E.M. Ciarlini, P.A.S. Veloso and A.L. Furtado. "A Formal Framework for Modelling at the Behavioural Level". *The Tenth European-Japanese Conference on Information Modelling and Knowledge Bases*, Saariselkä, 2000.

[FC] A.L. Furtado and A. E. M. Ciarlini. "Generating narratives from plots using schema information". In Proc. *NLDB'00 Applications of Natural Language to Information Systems*, Versailles, 2000.

[Fi] C. Fillmore. "The case for case". In *Universals in Linguistic Theory*. E. Bach and R. Harms (eds.). Holt, Rinehart and Winston, 1968.

[FN] R. E. Fikes and N. J. Nilsson. "STRIPS: A new approach to the application of theorem proving to problem solving". *Artificial Intelligence* , 2(3-4), 1971.

[Fu] A. L. Furtado. "Analogy by generalization and the quest of the grail". *ACM/SIGPLAN Notices*, 27, 1, 1992.

[FuN] A. L. Furtado and E. J. Neuhold. *Formal Techniques for Data Base Design*. Berlin: Springer-Verlag, 1986.

[Ka] H. A. Kautz. "A formal theory of plan recognition and its implementation", in *Reasoning about Plans*. J. F. Allen et al (eds.). San Mateo: Morgan Kaufmann, 1991.

[Ko] J. L. Kolodner. *Case-Based Reasoning*. San Mateo: Morgan Kaufmann, 1993.

[KS] R. Kowalski and F. Sadri. "From logic programming towards multi-agent systems". *Annals of Mathematics and Artificial Intelligence*, 25, 1-2: 391-419, 1999.

[MC] J. Mylopoulos and J. Castro. "Tropos: a framework for requirements-driven software development". in *Information Systems Engineering*. S. Brinkkemper, E. Lindencrona and A. Solvberg (eds.). London: Springer-Verlag, 2000.

[MCP] C.J. Matheus, P.K. Chan. and G. Piatesky-Shapiro. "Systems for knowledge discovery in databases". *IEEE Transactions on Knowledge and Data Engineering*, 5, 6, 1993.

[MCY] J. Mylopoulos, L. Chung and E. Yu. "From object-oriented to goal-oriented requirements analysis". Communications of the ACM, 42, 1: 31-37, 1999.

[MHL] J. J. Meyer, W. Hoek and B. Linder. "A logical approach to the dynamics of commitments". *Artificial Intelligence*, 113 (1-2): 1-41, 1999.

[MS] R. Miller and M. Shanahan. "Narratives in the situation calculus". *Journal of Logic & Computation*, 4, 5, 1994.

[Oz] G. Ozsoyoglu. and R.T. Snodgrass. "Temporal and real-time databases: a survey". *IEEE Transaction on Knowledge and Data Engineering*, 7, 4, 1995.

[SA] R.C. Schank and R.P. Abelson. *Scripts, Plans, Goals and Understanding*. Lawrence Erlbaum Associates, Hillsdale, NJ, 1977.

[Sg] N. M. Sgouros. "Dynamic generation, management and resolution of interactive plots". *Artificial Intelligence*, 107: 29-62, 1999.

[YTW] Q. Yang, J. Tenenberg and S. Woods. "On the Implementation and Evaluation of Abtweak". *Computational Intelligence Journal*, 1996.

Appendix 1

Schema declarations and input plots for the extended example, as in the **BLIB** Prolog tool:

```
% COMPANY ALPHA EXAMPLE

dbowner('company Alpha').


% Facts

entity(person, name).
attribute(person, in_payroll).

entity(employee).
is_a(employee, person).
attribute(employee, level).

entity(consultant).
is_a(consultant, person).

entity(company, denomination).

entity(client).
is_a(client,company).
attribute(client, account_status).

entity(course, title).

relationship(serving, [employee, client]).

relationship(dissatisfied_with, [client,employee]).

relationship(taking, [employee, course]).


% Operations

oper(sign_contract(C), [company/ with]).
added(sign_contract(C), client(C)).

oper(hire_r(E), [person/ o]).
added(hire_r(E), (employee(E), level(E, 1), in_payroll(E, yes))).

oper(hire_t(E), [person/ o]).
added(hire_t(E), (employee(E), level(E,1), in_payroll(E, yes))).

oper(hire_c(P), [person/ o]).
added(hire_c(P), (consultant(P), in_payroll(P, yes))).

oper(assign(E,C), [employee/ o, client/ to]).
added(assign(E,C), serving(E,C)).
precond(assign(E,C),
   ((not serving(E,C1)), (not serving(E1, C)))).
```

```
oper(enroll(E,T), [employee/ o, course/ in]).
added(enroll(E,T), taking(E,T)).
deleted(enroll(E,T), (dissatisfied_with(C,E),
        account_status(C,inactive))).
precond(enroll(E,T), (serving(E,C), not taking(E,T1))).

oper(promote(E), [employee/ o]).
added(promote(E), level(E,2)).
deleted(promote(E), level(E,1)).
precond(promote(E),
  (serving(E,C), not dissatisfied_with(C,E),
   level(E,1))).

oper(replace(E1,E2,C),
  [employee/ o, employee/ by, client/ for]).
added(replace(E1,E2,C), serving(E2,C)).
deleted(replace(E1,E2,C), serving(E1,C)).
precond(replace(E1,E2,C),
  (serving(E1,C), not serving(E2,C1))).

oper(fire(E), [employee/ o]).
deleted(fire(E),
  (employee(E), level(E,N), dissatisfied_with(C,E),
   account_status(C,inactive))).
precond(fire(E), (not serving(E,C))).

oper(complain(C,E), [client/ a, employee/ about]).
added(complain(C,E), (account_status(C,inactive),
      dissatisfied_with(C,E))).
precond(complain(C,E), serving(E,C)).


% Goal-inference rules

gi_rule('company_Alpha', (employee(E), not serving(E,C)),
  not employee(E)).
gi_rule('company_Alpha', (serving(E,C), account_status(C,inactive)),
  not account_status(C,inactive)).

gi_rule('company_Alpha',
  (company(C), not client(C), person(E), not employee(E)),
  (client(C), employee(E), serving(E,C))).

gi_rule('company_Alpha', (person(E), not in_payroll(E,yes),
   not employee(E)),
  (in_payroll(E,yes), employee(E))).

gi_rule('company_Alpha', (person(E), not in_payroll(E,yes)),
  (in_payroll(E,yes))).


gi_rule(employee(E1), (level(E1,1),level(E2,2)),
  level(E1,2)).

gi_rule(person(E), (person(E), not employee(E), client(C)),
  (employee(E), serving(E,C))).
```

```
% EXAMPLE RUNS

idb(1,(person('Leonard'), person('Mary'), person('John'),
     company('Beta'), company('Omega'), course(c135))).

idb(2,(person('Leonard'), person('Mary'), person('John'),
     company('Beta'), company('Omega'), course(c135),
     employee('Mary'),client('Beta'),serving('Mary','Beta'),
     level('Mary',1))).

idb(3,X) :- idb(2,X).

idb(4,(person('Leonard'), person('Mary'), person('John'),
     company('Beta'), company('Omega'), course(c135),
     employee('Mary'),client('Beta'),serving('Mary','Beta'),
     level('Mary',1),client('Omega'), employee('John'),
     level('John',1), serving('John','Omega'))).

idb(5,(person('Leonard'), company('Beta'), client('Beta'))).

idb(6,X) :- idb(1,X).

idb(7,X) :- idb(1,X).

idb(8,X) :- idb(1,X).

idb(9,X) :- idb(1,X).

idb(10,X) :- idb(1,X).

idb(11,X) :- idb(1,X).

plot(1,[s0,sign_contract('Beta'),hire_r('Mary'),
    assign('Mary','Beta')]).
plot(2,[s0,complain('Beta','Mary'),enroll('Mary',c135)]).
plot(3,[s0,complain('Beta','Mary'),hire_r('Leonard'),
    replace('Mary','Leonard','Beta'),fire('Mary')]).
plot(4,[s0,promote('John'),enroll('Mary',c135),promote('Mary')]).
plot(5,[s0,hire_r('Leonard'),assign('Leonard','Beta')]).
plot(6,[s0,hire_r('John'),fire('John')]).
plot(7,[s0,sign_contract('Beta'),hire_t('Mary'),
    assign('Mary','Beta')]).
plot(8,[s0,hire_t('John'),sign_contract('Beta'),
    assign('John','Beta')]).
plot(9,[s0,hire_c('Leonard')]).
plot(10,[s0,hire_t('Mary')]).
plot(11,[s0,hire_r('Mary')]).
```

Appendix 2

Contents of the Index after an example **Construction** phase:

```
% Index entry:
[[company_Alpha,
(employee(A),not serving(A,B)),
(not employee(A))],

[[fire(A)]]]


% Index entry:
[[company_Alpha,
(serving(A,B),account_status(B,inactive)),
(not account_status(B,inactive))],

[[improve_service(A),
op_complex(improve_service(A),[client/for]),
 [is_a(renovate_assistance(A,B,C),improve_service(A)),
  is_a(enroll(C,D),improve_service(A): serving(C,A))]],

[renovate_assistance(Beta,Leonard,Mary),
op_complex(renovate_assistance(Beta,Leonard,Mary),
 [client/for,person/with,employee/instead]),
components(renovate_assistance(Beta,Leonard,Mary),
  [f1: hire_r(Leonard),f2: replace(Mary,Leonard,Beta),
   f3: fire(Mary)],
  [f1-f2,f2-f3])],

[enroll(Mary,c135)]]]


% Index entry:
[[company_Alpha,
(company(A),not client(A),person(B),not employee(B)),
(client(A),employee(B),serving(B,A))],

[[obtain(A,B),
op_complex(obtain(A,B),[company/o,person/with]),
 [is_a(obtain_t(A,B),obtain(A,B)),
  is_a(obtain_r(A,B),obtain(A,B))]],

[obtain_t(Beta,A),
op_complex(obtain_t(Beta,A),[company/o,person/with]),
components(obtain_t(Beta,A),
  [f1: sign_contract(Beta),f2: hire_t(A),f3: assign(A,Beta)],
  [f1-f3,f2-f3])],

[obtain_r(Beta,Mary),
op_complex(obtain_r(Beta,Mary),[company/o,person/with]),
components(obtain_r(Beta,Mary),
  [f1: sign_contract(Beta),f2: hire_r(Mary),f3: assign(Mary,Beta)],
  [f1-f3,f2-f3])]]]
```

```
% Index entry:
[[company_Alpha,
(person(A),not in_payroll(A,yes),not employee(A)),
(in_payroll(A,yes),employee(A))],

[[hire_e(A),
op_complex(hire_e(A),[person/o]),
 [is_a(hire_t(A),hire_e(A)),
  is_a(hire_r(A),hire_e(A))]],

[hire_t(A)],

[hire_r(A)]]]


% Index entry:
[[company_Alpha,
(person(A),not in_payroll(A,yes)),
(in_payroll(A,yes))],

[[hire_c(Leonard)],

[hire_mp(A),
op_complex(hire_mp(A),[person/o]),
 [is_a(hire_c(A),hire_mp(A)),
  is_a(hire_t(A),hire_mp(A)),
  is_a(hire_r(A),hire_mp(A))]],

[hire_t(A)],

[hire_r(A)]]]


% Index entry:
[[employee(A),
(level(A,1),level(B,2)),
(level(A,2))],

[[advance(Mary),
op_complex(advance(Mary),[employee/o]),
components(advance(Mary),
  [f1: enroll(Mary,c135),f2: promote(Mary)],
  [f1-f2])]]]


% Index entry:
[[person(A),
(person(A),not employee(A),client(B)),
(employee(A),serving(A,B))],

[[engage(A,B),
op_complex(engage(A,B),[person/o,client/for]),
 [is_a(engage_t(A,B),engage(A,B)),
  is_a(engage_r(A,B),engage(A,B))]],
```

```
[engage_t(Mary,Beta),
op_complex(engage_t(Mary,Beta),[person/o,client/for]),
components(engage_t(Mary,Beta),
  [f1: hire_t(Mary),f2: assign(Mary,Beta)],
  [f1-f2])],

[engage_r(A,Beta),
op_complex(engage_r(A,Beta),[person/o,client/for]),
components(engage_r(A,Beta),
  [f1: hire_r(A),f2: assign(A,Beta)],
  [f1-f2])]]]
```

Appendix 3

Contents of the Index after the **Restructuring** phase:

```
% Index entry:
[[company_Alpha,
(employee(A),not serving(A,B)),
(not employee(A))],

[[fire(A)]]]


% Index entry:
[[company_Alpha,
(serving(A,B),account_status(B,inactive)),
(not account_status(B,inactive))],

[[improve_service(A),
op_complex(improve_service(A),[client/for]),
 [is_a(renovate_assistance(A,B,C),improve_service(A)),
  is_a(enroll(C,D),improve_service(A): serving(C,A))]],

[renovate_assistance(Beta,Leonard,Mary),
op_complex(renovate_assistance(Beta,Leonard,Mary),
 [client/for,person/with,employee/instead]),
components(renovate_assistance(Beta,Leonard,Mary),
  [f1: hire_r(Leonard),f2: replace(Mary,Leonard,Beta),
   f3: fire(Mary)],
  [f1-f2,f2-f3])],

[enroll(Mary,c135)]]]


% Index entry:
[[company_Alpha,
(company(A),not client(A),person(B),not employee(B)),
(client(A),employee(B),serving(B,A))],

[[obtain(Beta,A),
op_complex(obtain(Beta,A),[company/o,person/with]),
components(obtain(Beta,A),
  [f1: sign_contract(Beta),f2: engage(A,Beta)],
  [f1-f2])]]]


% Index entry:
[[company_Alpha,
(person(A),not in_payroll(A,yes),not employee(A)),
(in_payroll(A,yes),employee(A))],

[[hire_e(A),
op_complex(hire_e(A),[person/o]),
 [is_a(hire_t(A),hire_e(A)),
  is_a(hire_r(A),hire_e(A))]],
```

```
[hire_t(A)],

[hire_r(A)]]]


% Index entry:
[[company_Alpha,
(person(A),not in_payroll(A,yes)),
(in_payroll(A,yes))],

[[hire_c(Leonard)],

[hire_mp(A),
op_complex(hire_mp(A),[person/o]),
 [is_a(hire_e(A),hire_mp(A)),
  is_a(hire_c(A),hire_mp(A))]],

[hire_t(A)],

[hire_r(A)]]]


% Index entry:
[[employee(A),
(level(A,1),level(B,2)),
(level(A,2))],

[[advance(Mary),
op_complex(advance(Mary),[employee/o]),
components(advance(Mary),
  [f1: enroll(Mary,c135),f2: promote(Mary)],
  [f1-f2])]]]


% Index entry:
[[person(A),
(person(A),not employee(A),client(B)),
(employee(A),serving(A,B))],

[[engage(A,Beta),
op_complex(engage(A,Beta),[person/o,client/for]),
components(engage(A,Beta),
  [f1: hire_e(A),f2: assign(A,Beta)],
  [f1-f2])]]]
```

Appendix 4

Algorithm for the Most Specific Generalization of Plans

```
% most specific generalization of plans

msg_plan(P1,P1d,P2,P2d,M,L) :-
  length(P1,N1), length(P2,N1),
  length(P1d,N2), length(P2d,N2),
  numb_pos(P1,N3), numb_pos(P2,N3),
  findall([M1,L1],
    (msg_plan1(P1,P1d,P2,P2d,M1,Lsub),
     nsubst(Lsub,L1)),
    ML),
  minsubst(ML,M,Ns),
  listvar(M,Lvar),
  length(Lvar,L).

msg_plan1(P1,P1d,P2,P2d,M,Lsub) :-
  numb_pos(P2,Np),
  copy(P2,P2c),
  varnames(P2c),
  copy(P1,P1c),
  mk_v([P1c,P1d],[Xv,Xdv],[],L),
  copy(P1,P1cc),
  mk_v(P1cc,Xv1,[],L),
  mp(Xv,Xv1,P2,P2c),
  included(P2d,Xdv),
  numb_pos(Xv,Np),
  copy(P2,P2cc),
  copy(Xv,Xvc),
  forall(on(_:0,P2cc),on(_:0,Xvc)),
  pairl(P1,Xv,P2r),
  numb_pos(P2r,Np),
  msg(M,P1,P2r,[],Lsub).

nsubst([],0).
nsubst([[U,V,W]|R],N) :-
  U == V, !,
  nsubst(R,N).
nsubst([[U,V,W]|R],N) :-
  copy(U,U1),
  string_term(Us,U1),
  substring(Us,0,1,$f$),
  string_length(Us,L),
  L1 is L - 1,
  substring(Us,1,L1,I),
  string_term(I,I1),
  integer(I1), !,
  nsubst(R,N).
nsubst([T|R],N) :-
  nsubst(R,N1),
  N is N1 + 1 .

mp([],[],_,_).
mp([F:Av|R],[F:Av1|R1],B,B1) :-
  on(F:Av1,B1),
  on(F:Av,B),
  mp(R,R1,B,B1).
```



```
pairl([],[],[]) :- !.
pairl([F:_|R],[_:B|S],[F:B|T]) :-
  pairl(R,S,T).

minsubst([[M1,L1]|R],M,L) :-
  mins(R,M1,L1,M,L).

mins([],M,L,M,L).
mins([[M,L]|R],M1,L1,M2,L2) :-
  L @< L1, !,
  mins(R,M,L,M2,L2).
mins([[M,L]|R],M1,L1,M2,L2) :-
  mins(R,M1,L1,M2,L2).

numb_pos(X,N) :-
  copy(X,Y),
  varnames(Y),
  mk_v(Y,_,[],L),
  length(L,N1),
  length(X,N2),
  N is N1 - N2.


% most specific generalization of terms

msg(M,A,B) :-
  msg(M,A,B,[],L),
  chk_msg(L).

msg(T1,T1,T2,L,M) :-
  var(T1),
  T1 == T2, !,
  newv(T1,T2,T1,L,M).
msg(T3,T1,T2,L,M) :-
  (var(T1); var(T2)), !,
  newv(T1,T2,T3,L,M).
msg(T3,T1,T2,L,M) :-
  not var(T2), not var(T1),
  not type_similar(T1,T2), !,
  newv(T1,T2,T3,L,M).
msg(T1,T1,T2,L,L) :-
  atomic(T1),
  atomic(T2), !.
msg(T,T1,T2,L,M) :-
  type_similar(T1,T2),
  not atomic(T1),
  T1 =.. [F|T1args],
  T2 =.. [F|T2args],
  msg_list(Targs,T1args,T2args,L,M),
  T =.. [F|Targs].
msg_list([],[],[],L,L).
msg_list([T|Ts],[T1|T1s],[T2|T2s],L,M) :-
  msg(T,T1,T2,L,M1),
  msg_list(Ts,T1s,T2s,M1,M).

type_similar(A,A) :-
  atomic(A), !.
type_similar(T1,T2) :-
  T1 =.. [F|T1s], T2 =.. [F|T2s],
  length(T1s,N),
  length(T2s,N).
```

```
pairl([],[],[]) :- !.
pairl([F:_|R],[_:B|S],[F:B|T]) :-
  pairl(R,S,T).

minsubst([[M1,L1]|R],M,L) :-
  mins(R,M1,L1,M,L).

mins([],M,L,M,L).
mins([[M,L]|R],M1,L1,M2,L2) :-
  L @< L1, !,
  mins(R,M,L,M2,L2).
mins([[M,L]|R],M1,L1,M2,L2) :-
  mins(R,M1,L1,M2,L2).

numb_pos(X,N) :-
  copy(X,Y),
  varnames(Y),
  mk_v(Y,_,[],L),
  length(L,N1),
  length(X,N2),
  N is N1 - N2.


% most specific generalization of terms

msg(M,A,B) :-
  msg(M,A,B,[],L),
  chk_msg(L).

msg(T1,T1,T2,L,M) :-
  var(T1),
  T1 == T2, !,
  newv(T1,T2,T1,L,M).
msg(T3,T1,T2,L,M) :-
  (var(T1); var(T2)), !,
  newv(T1,T2,T3,L,M).
msg(T3,T1,T2,L,M) :-
  not var(T2), not var(T1),
  not type_similar(T1,T2), !,
  newv(T1,T2,T3,L,M).
msg(T1,T1,T2,L,L) :-
  atomic(T1),
  atomic(T2), !.
msg(T,T1,T2,L,M) :-
  type_similar(T1,T2),
  not atomic(T1),
  T1 =.. [F|T1args],
  T2 =.. [F|T2args],
  msg_list(Targs,T1args,T2args,L,M),
  T =.. [F|Targs].
msg_list([],[],[],L,L).
msg_list([T|Ts],[T1|T1s],[T2|T2s],L,M) :-
  msg(T,T1,T2,L,M1),
  msg_list(Ts,T1s,T2s,M1,M).

type_similar(A,A) :-
  atomic(A), !.
type_similar(T1,T2) :-
  T1 =.. [F|T1s], T2 =.. [F|T2s],
  length(T1s,N),
  length(T2s,N).
```

```
newv(A,B,V,L,M) :-
  (on([T1,T2,T3],L),
   A == T1, B == T2, !, V = T3, L = M;
   M = [[A,B,V]|L]).

chk_msg(L) :-
  chk_msg(L,L).

chk_msg([],_).
chk_msg([[A,B,V]|R],L) :-
  var(A), not var(B), !,
  (not (on([T1,T2,T3],L),
   (T1 == A, not (T2 == B);
    T2 == A)), !,
   V = A;
   true),
  chk_msg(R,L).
chk_msg([[A,B,V]|R],L) :-
  not var(A), var(B), !,
  (not (on([T1,T2,T3],L),
   (T2 == B, not (T1 == A);
    T1 == B)), !,
   V = B;
   true),
  chk_msg(R,L).
chk_msg([T|R],L) :-
  chk_msg(R,L).


% uninstantiation -- changes all constant arguments into variables

mk_v(T,T,L,L) :-
  var(T), !.
mk_v(T,T,L,L) :-
  T = [], !.
mk_v(T,V,L,L) :-
  atomic(T),
  on([T,V],L), !.
mk_v(T,V,L,L1) :-
  atomic(T), !,
  L1 = [[T,V]|L].
mk_v(T,V,L,L1) :-
  T = [A|R], !,
  mk_vl([A|R],V,L,L1).
mk_v(T,V,L,L1) :-
  T =.. [F|Ar],
  mk_vl(Ar,Ar1,L,L1),
  V =.. [F|Ar1].

mk_vl([],[],L,L).
mk_vl([A|R],[B|S],L,L1) :-
  mk_v(A,B,L,L2),
  mk_vl(R,S,L2,L1).


% general utilities

on(X, [X|_]).
on(X, [_|R]) :- on(X,R).
```

```
reverse(Xs,Ys) :-
  rev1(Xs,[],Ys).

rev1([X|Xs],A,Ys) :-
  rev1(Xs,[X|A],Ys).
rev1([],Ys,Ys).

no_dup([X|Xs],Ys) :-
  v_on(X,Xs), !,
  no_dup(Xs,Ys).
no_dup([X|Xs],[X|Ys]) :-
  not v_on(X,Xs),
  no_dup(Xs,Ys).
no_dup([],[]).

c_write(X) :-
  copy(X,Y),
  varnames(Y),
  write(Y).


% example runs

test :-
  p1(P1),d1(D1),
  p2(N, P2), d2(N, D2),
  write(N),nl,
  msg_plan(P1,D1,P2,D2,M,L),
  c_write(M),write($ number of variables: $),write(L),nl.

% plan to be compared with the others
p1([f1:g(a,b),f2:h(a,a),f3:i(b,c)]).
d1([f1-f2]).

% succeeds
p2(1, [f4:g(c,b),f2:i(b,d),f3:h(c,c)]).
d2(1, [f4-f3]).

% fails: different designations
p2(2, [f1:g(a,b),f2:h(c,c),f3:i(b,c)]).
d2(2, [f1-f2]).

% fails: different designations
p2(3, [f1:g(A,B),f2:h(C,C),f3:i(B,C)]).
d2(3, [f1-f2]).

% fails: different designations
p2(4, [f1:g(A,A),f2:h(A,A),f3:i(A,C)]).
d2(4, [f1-f2]).

% succeeds
p2(5, [f4:g(A,B),f3:h(A,A),f2:i(B,C)]).
d2(5, [f4-f3]).

% fails: different dependencies
p2(6, [f1:g(a,b),f2:h(a,a),f3:i(b,c)]).
d2(6, [f2-f1]).

% fails: different dependencies
p2(7, [f1:g(a,b),f2:h(a,a),f3:i(b,c)]).
d2(7, [f1-f3]).
```