Modular Design and Verification of Logical Knowledge Bases

Grigoris Antoniou
University of Osnabrueck
Dept. of Computer Science (FB 6)
Albrechtstrasse 28
D-4500 Osnabrueck
ga@informatik.Uni-Osnabrueck.DE

Abstract

In this paper we describe a framework for the design of modular knowledge based systems which is motivated by work in algebraic specification and software engineering. The main characteristic of the framework is that verification work can be done in a local setting. We present two concrete module concepts within this framework, and give formal semantics and correctness notions for them. Finally, we show a method for proving correctness of modules using an assertional proof system for logic programs.

1. Introduction

As artificial intelligence is coming to age and is moving towards practical application, it becomes apparent that it has to learn some lessons that conventional computer science did some time ago. Concepts like *structuring*, *specification*, *validation*, and *module* evolved and are nowadays central in programming systems. We expect the same to happen in AI. Especially formal specification notions seem to be close to concepts of the knowledge level approach to problem solving, as pointed out in [8].

In this paper we shall put emphasis on *modularity* and *verification*. Usually, a complex problem is attacked by decomposing it into simpler subproblems. In the context of knowledge engineering, modularization means that the knowledge about a problem is organized as the interaction of several well-defined, semantically related parts of knowledge (called modules). Benefits of such an approach include comprehensibility and maintainability, an important task in any large system where several developers and users are involved in the assimilation of increasingly many knowledge items; further reusability which is a crucial issue for future AI systems: As knowledge is very complex, we cannot afford wasting whatever knowledge we succeed to acquire or validate [7].

We present a general framework of modular design for logical knowledge bases. We have oriented ourselves at work in the field of algebraic specification [5] and work on modules with imperative implementation part [1], [9]. There are many similarities with the work in [8], but also some important new aspects: We give some concretizations of our model, i.e. special representation formalisms and special kinds of interfaces. For these concrete module concepts, we give full formal semantics and correctness notions. Therefore, we are also able to discuss *formal verification* of such modules. In particular, we give a method for showing correctness of modules in the context of logic programming.

Some words concerning the relationship of our work with the V&V area seem to be necessary. First, by verification we mean proving correctness w.r.t. a declarative, formal specification; important aspects like redundancy, validation against the 'real world expectations' etc. are not treated. It is our belief that 'hard', mathematically precise concepts and methods coming from the field of logic, and in particular of logic programming, should be used in the verification of knowledge-based systems. This appears especially important in critical applications (like supervision of nuclear factories), where we want to be sure that some erroneous situations cannot occur.

2. Basic notions of logic programming

We use here standard notions and notation for logic programming as can be found in [6] or [9]. In particular *Horn formulas* are formulas of one of the following forms:

$\forall((B_1 \wedge \ldots \wedge B_m) \rightarrow A)$ (or $A \leftarrow B_1,\ldots,B_m$)

$\forall(A)$, also written as A, and

$\forall(\neg B_1 \vee \ldots \vee \neg B_m)$ (or $\leftarrow B_1,\ldots,B_m$)

with atomic formulas A, $B_1,\ldots,B_m$. Formulas of the first two forms are called *definite program formulas*, formulas of the third form *definite goals*. Program formulas of the first kind are called *rules* and of the second kind *facts*. In a definite program formula, A is called its *head* and $B_1,\ldots,B_m$ (if existent) its *body*.

A *definite logic program* (in this paper often called *logic program*; please distinguish this notion from general logic programs in the sense of [6]) P is a (maybe infinite) set of definite program formulas. We say that P *defines* a predicate p if p occurs in the head of a formula in P, while P *uses* a predicate p if p occurs in the body of some formula in P.

The *least Herbrand model* of logic program P is denoted by M_P and is defined as the set of all ground atoms in a given signature Σ that logically follow from P. There is a well known approximation of M_P making use of an increasing union of Herbrand models $I_n(P)$ for n=0,1,2,... (see [6]).

When trying to describe some domain with logic programs it is often necessary to use *auxiliary predicates*, even though they are not of interest. In these cases it is usually desirable that the definition of the new predicates does not change the meaning of the old predicates. This idea leads to the concept of conservative extensions.

Let Q be a definite logic program and M a set of predicates. We say that Q is *conservative with respect to* M if Q does not define predicates from M. Predicates of M may thus be only used in the body of rules in Q.

Let P and Q be definite logic programs over the same set of function symbols. We say that $P \cup Q$ is a *conservative extension* of P if Q is conservative w.r.t. the set of predicates occurring in P. The following lemma expresses the main property of conservative extensions.

Lemma Let P and Q be definite logic programs over the same set of function symbols such that $P \cup Q$ is a conservative extension of P. Then

$$M_{P \cup Q} \cap H(P) = M_P$$

where H(P) denotes the set of ground atoms $p(t_1,\ldots,t_k)$ with p occurring in P.

The lemma states thus that conservatively extending of P to $P \cup Q$ does not affect the semantics of predicates in P.

3. A general framework for modular knowledge bases

3.1 Static modular systems

A module includes an *import interface* describing the knowledge required in order to define the module's functionality, and an *export interface* describing the knowledge the module is offering to the public. Both parts are parametrized by a common *parameter part*. This parametrization allows development of *generic modules* that can be adapted to many specific applications by suitably instantiating their parameter part. The module's *body* is the part defining the functionality of the module, i.e. its knowledge.

Modules are independent entities communicating with their environment via their interfaces. The ideal case would be that the export interface gives a full description of the knowledge exported (thus playing a role similar to that of abstract data types in conventional computer science). Unfortunately, such a complete specification of AI systems is usually impossible. In such cases, the export interface describes the signature of exported knowledge (like interfaces in imperative programming languages) and some *integrity (consistency) conditions* the exported knowledge should fulfill.

A modular system is built by putting modules together that are combined by gluing together the module interfaces of several modules; the knowledge required by one module is imported from other modules. In general, the approach consists in first solving the problem on a high level of abstraction and then realizing the needed information in form of a *hierarchy of modules*.

The simplest situation is that the information needed by module M1 is obtained by the export interface of a module M2. In this case there must be some kind of mapping of the import interface of M1 to the export interface of M2. In case of first order interfaces, this mapping is a *specification morphism*, a well-known concept from the theory of algebraic specification [5]. For interfaces in Horn logic, a variant called logic program morphism (see [2] for details) is more appropriate.

Following the tradition of algebraic modules (e.g. [5]), modules are combined by so-called *module operations* describing natural ways of putting modules together. The main module operations are:

- *Composition* (or hierarchical combination): The knowledge imported by module M1 is exported by module M2. Therefore, there must be a map-

ping (for example specification morphism) from the import interface of M1 to the export interface of M2.

- *Union*: It is not reasonable to suppose that all operations imported by one module are exported by *one* other module, as the composition operation requires. It is rather usual that parts of the import side are provided by several modules. Therefore we need a possibility to build the union of these modules.
- *Actualization*: It instantiates the parameter part of a generic module by some concrete elements and thus serves the purpose of adapting parameter parts to specific applications.

Formal descriptions of these operations for some concrete module concepts (like those in section 4) can be found in [2]. Results concerning these operations are:

- *compositionality of semantics*: The semantics of the resulting knowledge base must be naturally expressible in terms of the semantics of the constituent modules.
- *modular correctness*: Correctness of the resulting system follows directly from the correctness of the modules used as arguments of the module operation. Therefore, verification and validation work can be done in a local setting: If the entire knowledge base is built using correctness preserving module operations, it suffices to show correctness of each single module involved.

Let us see a simple informal example of a knowledge base obtained by composition and union operations: The example in Figure 1 (the figures can be found at the end of the paper) shows a very raw sketch of the knowledge someone uses when selecting a conference for submitting a paper. It requires knowledge about whether the person's research interests are included among the conference topics, whether it is financially possible to attend the conference, whether the conference location is interesting, etc. Each unit is a module with interfaces. The module on personal reasons, for instance, could include in its export interface predicates as *like(Conference)*, in its import interface predicates as *conference_location* or *like_country*, and its body rules as

```
like(Conf) :- conference_location(Conf,Loc),
              is_located(Loc,Country),
              like_country(Country).
```

The general modularity framework sketched above leaves many options open in the development and validation of a modular knowledge base.

1. Knowledge in the module body is hidden from the outside, if it does not appear in the interfaces. Furthermore, the module interfaces define the formal requirements to the semantics of a module. So, formal verification of the module's knowledge against its interfaces is possible. Exactly this is the view we shall take in the rest of the paper.
2. It should also be noted that various representation formalisms may be used in the same knowledge base (for example Horn logic with equality, many- sorted and ordered-sorted logic etc.). In order that such modules can work together, the specification language used at the module interfaces should be general and powerful enough to give a semantical description of various representations.

3.2 Dynamic module interconnection

It is not always possible to split knowledge in a static, hierarchical manner as in the model of the previous subsection. Often, the constituent modules may vary depending on the concrete constellation. Or, modules with mutually inconsistent knowledge may be maintained in one knowledge base but may not be used at the same time. Here, we outline a possibility of dealing with such situations.

As it would be beyond the scope of this paper to give more details of this model, we shall only consider an example that will hopefully illustrate the underlying ideas. Figure 2 shows a structured knowledge base with (a part of) the knowledge I need when being downtown. In some connections, the "lower" modules contain knowledge on different aspects (for example the top connection), whereas in others they contain competing knowledge items that exclude each other (for example, module *going to work* could contain in-hurry, while *spending my time* could include ¬in-hurry).

The first sort of packet connection is called *AND-connection* and states that the packets from which the top one can import do not contain competing knowledge, but rather information on different topics of the modeled domain (Remark: Whereas it is often intuitively clear what competing knowledge means, it is difficult to give a general, formal definition; it is up to the knowledge engineer to decide). This means that the knowledge of all these packets (or of some of them) may be used at the same time.

The second connection possibility is *OR-connection* which indicates that the packets (on the lower level) involved contain competing knowledge. In this

case, only one of these packets may be visible at a given time, similar to the model described in the beginning of this subsection.

The meaning of a structured, modular knowledge base is defined with respect to a *current focus*. This focus defines a current view on the knowledge base and must be such that competing parts of knowledge are not visible at the same time. In the example of Figure 2, a focus could consist of the modules *by car*, *going to work*, and *last night*. Then, these modules and all modules above them are visible at the moment, i.e. their knowledge may be used. Note that for each OR-connection, at most one module can be included in a current focus. For each AND-connection, none, one, some, or all involved modules (of the lower level) may be included. In our example here, we have not included *my health* to the focus. It could be the case that I have slept bad tonight, so even my good health cannot prevent my mood from being bad.

4. Formal module concepts: two concrete variants

4.1 Modules with Horn logic interfaces

Horn logic is a well-understood representation formalism with many applications in practice and quite efficient implementation. Therefore, we shall discuss Horn logic module bodies in this and the next subsection.

Here we also consider *interfaces given in Horn logic*. Though it could seem quite strange, there are many situations where a more abstract (and therefore possibly inefficient or unrunnable) logic program can be used to *describe in a clear way* the semantics of an efficient but more complicated logic program. Consider, for example, the implementation of cycle-free path searching in a directed graph in figure 3. Naturally, the graph is used as a parameter of the module. The usual logic programming implementation is included in the module body, while the imported and exported predicates occur in the import and export interface respectively. The meaning of these predicates is described *in form of logic programs*. Note that the export interface is also a logic program for path searching, a readable but not runnable one. Our idea is that effective and efficient implementations are used in the body, while they are described by more abstract logic programs in the interfaces (with no emphasis on efficiency).

A **variant 1 module** M consists of a set Ω of function symbols, a parameter part PAR, an export interface EXP, an import interface IMP and an implementation part BODY. Their formal definition is:

- PAR consists of a set of predicates Σ_{par} and a first order axiom set *Par* over $\Sigma_{par} \cup \Omega$, the *constraints* of M.
- EXP consists of a set of predicates Σ_{exp} disjoint from Σ_{par}, and a definite logic program *Exp* over $\Sigma_{par} \cup \Sigma_{exp} \cup \Omega$. Exp must be conservative w.r.t. Σ_{par}. Certain predicates from Σ_{exp} are distinguished as *exported* predicates (these are made available to module users), while the remaining ones are called *auxiliary*.
- IMP consists of a set of predicates Σ_{imp} disjoint from $\Sigma_{par} \cup \Sigma_{exp}$, and a definite logic program *Imp* over $\Sigma_{par} \cup \Sigma_{imp} \cup \Omega$. Imp must be conservative w.r.t. Σ_{par}. Certain predicates from Σ_{imp} are distinguished as *imported* predicates (these are made visible to the outside), while the remaining ones are called *auxiliary*.
- BODY consists of a set of auxiliary predicates Σ_{body} disjoint from $\Sigma_{par} \cup \Sigma_{exp} \cup \Sigma_{imp}$, and a definite logic program *Implem* over $\Sigma_{par} \cup \Sigma_{body} \cup \Omega \cup \{p \in \Sigma_{exp} \mid p \text{ exported}\} \cup \{p \in \Sigma_{imp} \mid p \text{ imported}\}$. It is required that this program is conservative w.r.t. $\Sigma_{par} \cup \Sigma_{imp}$.

The required conservativity condition of Implem w.r.t. $\Sigma_{par} \cup \Sigma_{imp}$ is essential. It states that imported knowledge is protected in the module body. Since the idea is that the module's functionality is defined using the imported knowledge, the protection of the latter is natural.

Let us now define the formal semantics and correctness of such modules. Looking at a module as described above, there are two points of view concerning its meaning:

- What knowledge is the module supposed to offer to the public?
- What knowledge does the module indeed offer to the public?

The first view is described by the export interface and the parameter part of the module. The second view is obtained by interpreting the logic programs in the module body defining the module's functionality. Obviously, a module is *correct* if these two aspects of semantics coincide, i.e. if the module exports knowledge satisfying the conditions of its export interface. Here the precise definitions for these concepts:

Definition Given a logic module M, a *parameter model* P of M is a (perhaps infinite) set of program formulas in the signature $\Sigma_{par} \cup \Omega$ extended by an arbitrary set Const(P) of new constants such that M_P (M_P is the least Herbrand model of logic program P) is a model (in the sense of predicate logic) of Par. The set of new constant symbols may depend on P.

Definition Let a logic module Mod and a parameter model P of Mod be given. The following logical notions must be read w.r.t. the set of functions $\Omega \cup \text{Const}(P)$. We define

$$\text{export}_{\text{Mod}}(P) = M_{P \cup \text{Exp}}$$
$$\text{import}_{\text{Mod}}(P) = M_{P \cup \text{Imp}}$$
$$\text{implement}_{\text{Mod}}(P) = M_{\text{import}_{\text{Mod}}(P) \cup \text{Implem}}$$

Hence $\text{export}_{\text{Mod}}(P)$ describes the requirements to module semantics, $\text{implement}_{\text{Mod}}(P)$ the actual semantics of the module as it is implemented. The module is correct if these two views are equivalent. More formally: Mod is called **correct** iff for every parameter model P of Mod, every exported predicate q and all ground terms $t_1,\ldots,t_n$

$$q(t_1,\ldots,t_n) \in \text{export}_{\text{Mod}}(P) \Leftrightarrow$$
$$q(t_1,\ldots,t_n) \in \text{implement}_{\text{Mod}}(P)$$

Example

Let P be a parameter model of the path-searching example. It contains facts

edge(a,b)
diff(a,b)

where a,b are new constants (thought of as members of the domain of a usual model of the parameter part; these are the members of Const(P); the benefit of representing parameter models this way is that in the following, logic programming techniques can be applied to them). The facts with predicate *edge* define a graph. The following notions are w.r.t. this graph.

$M_{P \cup \text{Exp}} = P \cup \{\text{path}(a,b) \mid \text{there is a path from a to b}\}$

$M_{P \cup \text{Imp}} = P \cup \{\text{new}(a,[b1,\ldots,bn]) \mid a \notin \{b1,\ldots,bn\}\}$

$M_{M_{P \cup \text{Imp}} \cup \text{Implem}} =$
$P \cup \{\text{new}(a,[b1,\ldots,bn]) \mid a \notin \{b1,\ldots,bn\}\} \cup$
{restrictedPath(a,c,[b1,…,bn]) | there is a path from a to c with interior nodes not in {b1,…,bn}} ∪
{path(a,b) | there is a path from a to b}.

Hence we have that the same ground facts occur in $M_{P \cup \text{Exp}}$ as in $M_{M_{P \cup \text{Imp}} \cup \text{Implem}}$. Since path is the only exported predicate of Mod, we have that Mod is correct.

Though simple, the above correctness proof required reasoning about Herbrand models, i.e. infinite objects. In section 5 we shall be concerned with presenting a *finitary proof strategy* for modules.

4.2 Modules with first order interfaces

Though theoretically elegant, usage of Horn logic interfaces has its limit in practice, since often the expressive power of predicate logic is needed in order to give a simple description of a logic program, or if only an incomplete specification of a module (in form of some crucial conditions) is appropriate or viable.

Definition A *variant 2 module* is defined like a variant 1 module except for Exp and Imp: Exp (Imp) is now a first order specification over $\Sigma_{par} \cup \Sigma_{exp} \cup \Omega$ ($\Sigma_{par} \cup \Sigma_{imp} \cup \Omega$) such that Exp∪Par (Imp∪Par) is a conservative extension of Par.

Definition Let P be a parameter model of Mod. $P \cup A$ is an *import model* of Mod if A is a set of program formulas over signature $\Sigma_{imp} \cup \Omega \cup \text{Const}(P)$ such that $M_{P \cup A}$ is a model of Imp. Export models are defined in an analogous way.

Definition Given an import model $P \cup A$ of Mod, define its semantics as follows:

$$\text{Sem}_{\text{Mod}}(P \cup A) := M_{P \cup A \cup \text{Implem}} \cap$$
$$H(\Sigma_{par} \cup \Sigma_{exp} \cup \Omega \cup \text{Const}(P)).$$

M is *correct* if for each $P \cup A$, $\text{Sem}_M(P \cup A)$ is a model of Exp.

Example

Consider a module containing knowledge about persons, their relationships, and their age. Its body contains thus program clauses like

mother(jean, peter),
father(peter,tom),
age(peter,37),
grandmother(X,Y) :- mother(X,Z), mother(Z,Y)
etc.

The export interface contains formulas like

$\forall X \exists Y age(X,Y)$

$\forall X \forall Y (father(X,Y) \rightarrow older(X,Y))$

older is an auxiliary predicate used in the export interface. Its definition is:

$\forall X \forall Y (older(X,Y) \leftrightarrow$

$\exists U \exists V (age(X,U) \wedge age(Y,V) \wedge U > V))$

The import interface of this module can be left empty (it does not require any external knowledge). The module is correct if the conditions in the export interface are valid in the least Herbrand model of the body program.

Remark

Usage of Horn logic as representation language can be easily extended in several directions, for example by introducing negation (use stratified logic programs replacing the least Herbrand model by the canonical model) or sorts and equality [2], thus enabling knowledge representation using also *functions*.

5. Verification of logical knowledge bases

Now we turn our attention to giving a method for proving module correctness as it was defined in previous sections. The following method resembles the method of *intermediate assertions* in *Horn logic* because of the decomposition of verification tasks to simpler ones in a structured manner.

Module correctness is reduced to showing some relations of the form $M_P \models \varphi$ for logic program P and formula φ (φ holds in M_P). These are then resolved using completion and induction (see [9] for details).

In the following we shall need the notion of *annotated predicate*. It has the form $\{\varphi\}p(X_1,\ldots,X_n)\{\psi\}$ (where p is a predicate, and φ,ψ are formulas) and denotes the formula $((\varphi \wedge p(X_1,\ldots,X_n)) \rightarrow \psi)$ (φ should be thought of as the precondition, ψ as the postcondition for satisfaction of p). Figure 4 summarizes a method for showing validity of $\{\varphi\}p(X_1,\ldots,X_n)\{\psi\}$ in M_P. It is a modified and corrected version of [3]. We have shown that this method is sound w.r.t. the least Herbrand model semantics of logic programs.

Finally, we show how module correctness can be reduced to validity of annotated predicates. We consider only modules of variant 1; variant 2 can be treated in a similar way. Correctness of a version 1 module means:

(1) $p(t_1,\ldots,t_n) \in M_{P \cup Exp}$ iff
$p(t_1,\ldots,t_n) \in M_{P \cup Imp \cup Implem}$.

Rename the exported predicates $p \in \Sigma_{exp}$ into new ones p*, and let Exp* be the renamed version of Exp. Due to the conservativity conditions in the definition of modules, (1) is equivalent to

(2) $M_{P \cup Imp \cup Implem \cup Exp^*} \models$
$(p^*(t_1,\ldots,t_n) \leftrightarrow p(t_1,\ldots,t_n))$.

for all ground terms $t_1,\ldots,t_n$. Therefore we must show:

(3) $M_{P \cup Imp \cup Implem \cup Exp^*} \models$
$\forall X_1 \ldots \forall X_n (p^*(X_1,\ldots,X_n) \leftrightarrow p(X_1,\ldots,X_n))$.

This means in terms of annotated predicates:

(4) $M_{P \cup Imp \cup Implem \cup Exp^*} \models$
$\{true\}p^*(X_1,\ldots,X_n)\{p(X_1,\ldots,X_n)\}$

(5) $M_{P \cup Imp \cup Implem \cup Exp^*} \models$
$\{true\}p(X_1,\ldots,X_n)\{p^*(X_1,\ldots,X_n)\}$

(4) and (5) can be treated using the method described in Figure 4.

Example

As a very simple application of this method, consider the example from subsection 4.2, and suppose we want to show {true} grandmother(X,Y) {older(X,Y)} (such integrity constraints are usually used in data bases!). The only available rules for *grandmother* are

grandmother(X,Y) :- mother(X,Z), mother(Z,Y)
grandmother(X,Y) :- mother(X,Z), father(Z,Y).

We equip the first rule (the second is treated the same way) with intermediate assertions as follows:

grandmother (X,Y) :-mother(X,Z) {older(X,Z)},
mother(Z,Y) {older(X,Y)}.

Having shown the *subverification task* {true} mother(X,Y) {older(X,Y)} we are through using transitivity of *older* (follows from the definition of this predicate in the export interface and from the transitivity of <). Our final task

{true} mother(X,Y) {older(X,Y)}

is shown directly in the least Herbrand model of the knowledge base.

6. Conclusion

We introduced a modular framework for logical knowledge bases and two concrete module concepts. Major characteristics are formal interfaces and our insistence on correctness. Finally, we gave a verification strategy for our modules.

Usage of several representation languages within one system is supported, but it only works if we have sufficiently rich specifications in the interfaces. This appears to be especially difficult for nonmonotonic knowledge bases. We shall present some first ideas for specifying and validating such knowledge bases in a forthcoming paper.

Literature

[1] Antoniou, G.: *Modules and Verification*, Proc. IMA-Conference on the Unified Computation Laboratory, Stirling 1990, Oxford University Press 1992

[2] Antoniou, G. and Sperschneider, V.: *Modularity for Logic Programs*, Proc. Annual British Logic Programming Conference 1992, Springer

[3] Bossi,A. and Cocco,N.: *Verifying correctness of logic programs*, in Proc. Tapsoft'89, Springer

[4] Chandrasekaran, B.: *Generic Tasks in Knowledge-Based Reasoning: High-level Building Blocks for Expert Systems Design*, in IEEE Expert, Fall 1986, 23-30

[5] Ehrig, H. and Mahr, B.: Fundamentals of Algebraic Specification Vol. 2, Springer 1990

[6] Lloyd, J.W.: *Foundations of Logic Programming* 2. ed., Springer 1987

[7] Neches, R., Fikes, R., Finin et. al.: *Enabling Technology for Knowledge Sharing*, AI Magazine 12,3, 36-56

[8] Sierra, C., Agusti-Cullell, J. and Plaza, E.: *Verification by Construction in MILORD*, Proc. EUROVAV'91

[9] Sperschneider,V. & Antoniou,G.: *Logic: A Foundation for Computer Science*, Addison-Wesley 91

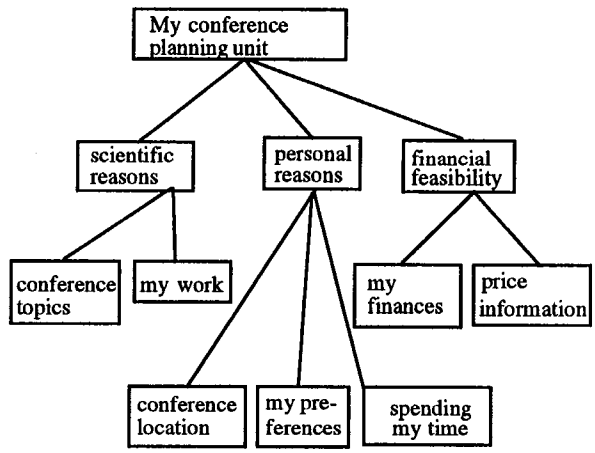

Figure 1

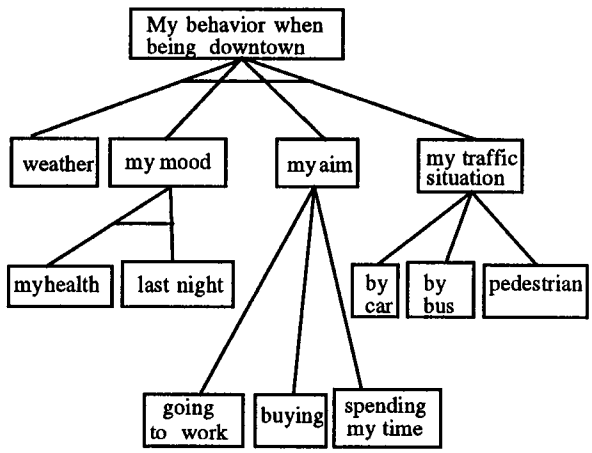

Figure 2

EXP

Exported predicates: path(X,Y)
Auxiliar predicates: none
Definitions:

```
path(X,X).
path(X,Y):-edge(X,Z),diff(X,Z),path(Z,Y).
```

BODY

Auxiliar predicates:

```
restrictedPath(X,Y,ForbiddenList)
```

Definitions:

```
path(X,Y):-restrictedPath(X,Y,[X]).

restrictedPath(X,X,L).
restrictedPath(X,Y,L):-
        edge(X,Z),
        new(Z,L),
        restrictedPath(Z,Y,[Z|L]).
```

IMP

Imported predicates: new(X,L)
Auxiliar predicates: none
Definitions:

```
new(X,[]).
new(X,[Y|L]):-diff(X,Y),new(X,L).
```

PAR

Predicates: edge(X,Y),diff(X,Y)

Constraints:

$diff(X,Y) \leftrightarrow \neg X=Y$

Figure 3

Verification strategy for $\{\varphi\}p(X_1,\ldots,X_n)\{\psi\}$

Step 0: Variable condition

Let the free variables of φ and ψ be among $X_1,\ldots,X_n,Y_1,\ldots,Y_m$. $Y_1,\ldots,Y_m$ should not occur in logic program P. Also, no quantification in φ and ψ over a variable occurring in P. Rename if necessary.

Step 1: Consistency with the facts

For every fact $p(t_1,\ldots,t_n)$ in P, show validity in M_P of:

$$(\varphi\{X_1/t_1,\ldots,X_n/t_n\} \rightarrow \psi\{X_1/t_1,\ldots,X_n/t_n\})$$

Step 2: Intermediate assertions for the rules

Equip each rule $p(t_1,\ldots,t_n):-L_1,\ldots,L_k$ of P with "intermediate assertions":

$\{q_0\}\ L_1\ \{q_1\}\ldots\{q_{k-1}\}\ L_k\ \{q_k\}$

with $q_0=\varphi\{X_1/t_1,\ldots,X_n/t_n\}$

and $q_k=\psi\{X_1/t_1,\ldots,X_n/t_n\}$.

Step 3: Match with pre- and postcondition of p

For an asserted rule body as in step 2 and an $i\in\{1,\ldots,k\}$ with $L_i=p(t_{i1},\ldots,t_{in(i)})$, show validity in M_P of the following:

"match pre" $((q_0\wedge\ldots\wedge q_{i-1}) \rightarrow \varphi\{X_1/t_{i1},\ldots,X_n/t_{in(i)}\})$

"match post" $((\psi\{X_1/t_{i1},\ldots,X_n/t_{in(i)}\}\wedge q_0\wedge\ldots\wedge q_{i-1}) \rightarrow q_i)$

Step 4: Subverification

For an asserted rule body as in step 2 and an $i\in\{1,\ldots,k\}$ with $L_i= p_i(t_{i1},\ldots,t_{in(i)})$ with $p_i\neq p$, show validity in M_P of:

$$\{(q_0\wedge\ldots\wedge q_{i-1}\wedge Z_1=t_{i1}\wedge\ldots\wedge Z_{n(i)}=t_{in(i)})\}\ \ p_i(Z_1,\ldots,Z_{n(i)})\{q_i\}$$

with new variables $Z_1,\ldots,Z_{n(i)}$. This is done either by applying the same strategy again, or directly using the completion of P, induction, and, if necessary, a suitable conservative extension.

Figure 4