Design of formal specifications by normal logic programs : merging formal text and good comments

Sophie Renault, Pierre Deransart

INRIA

INSTITUT NATIONAL DE RECHERCHE EN INFORMATIQUE ET EN AUTOMATIQUE

Design of formal specifications by normal logic programs : merging formal text and good comments

Sophie RENAULT
Pierre DERANSART

N° 1897
Mai 1993

PROGRAMME 2

Calcul Symbolique,
Programmation
et Génie logiciel

Design of Formal Specifications by Normal Logic Programs: Merging Formal Text and Good Comments.

Elaboration de spécifications formelles par des programmes logiques normaux : combinaison de texte formel et de bons commentaires.

Sophie Renault Pierre Deransart

Abstract

Formal specifications are a kind of art: they mixture formal text and informal comments in an unstructured manner. We present a method based on logic programming in which formal text and comments can be also formally related through proofs.
The paper is focused on the description of the proof system used in the design of such specifications. The specification language uses normal clauses (definite clauses with possibly negative literals in the body) for its formal part and is aimed at defining relations. The comments are expressed as logic formulas written with some of the specified relations and define some local properties that the relations must satify. They introduce some redundancy which helps to understand the specification.
The proofs are partial and partly automatized. They are not aimed at complete validation of the specification, but rather at debuging it and improving the comments such that its understanding is facilitated.
This approach is illustrated by the example of the design of draft standard Prolog on which this methodology is currently applied.

Résumé

L'écriture de spécifications formelles est une tâche délicate : elle doit habilement combiner le texte formel et les commentaires informels dans une composition structurée. Nous présentons une méthode fondée sur la programmation en logique et dans laquelle le texte formel et les commentaires sont tous deux impliqués dans des preuves.
Ce papier s'attache principalement à la description d'un systéme d'aide à l'élaboration et à la preuve de telles spécifications.
Le langage de spécification est, pour la partie formelle, le langage des clauses normales (clauses définies avec éventuellement des littéraux négatifs dans le corps des clauses). Les commentaires sont des formules logiques utilisant certaines des relations spécifiées. La redondance qu'elles introduisent permet une meilleure compréhension de la spécification.
Les preuves sont partielles et partiellement automatisées. Elles ne prétendent pas à une validation complète de la spécification mais plutôt à son déboguage et à l'amélioration et la clarification de l'expression des commentaires.
Cette approche est illustrée par l'exemple de l'élaboration d'une définition pour le standard de Prolog pour laquelle cette méthodologie est actuellement utilisée.

Design of Formal Specifications by Normal Logic Programs: Merging Formal Text and Good Comments.

Sophie Renault Pierre Deransart
INRIA-Rocquencourt
Domaine de Voluceau
78153 Le Chesnay, FRANCE
{Sophie.Renault, Pierre.Deransart}@inria.fr

Abstract

Formal specifications are a kind of art: they mixture formal text and informal comments in an unstructured manner. We present a method based on logic programming in which formal text and comments can be also formally related through proofs.

The paper is focused on the description of the proof system used in the design of such specifications.

The specification language uses normal clauses (definite clauses with possibly negative literals in the body) for its formal part and is aimed at defining relations. The comments are expressed as logic formulas written with some of the specified relations and define some local properties that the relations must satify. They introduce some redundancy which helps to understand the specification.

The proofs are partial and partly automatized. They are not aimed at complete validation of the specification, but rather at debuging it and improving the comments such that its understanding is facilitated.

This approach is illustrated by the example of the design of draft standard Prolog on which this methodology is currently applied.

1 Introduction

Formal specifications are a kind of art: they mixture formal text and informal comments in an unstructured manner. We present a method based on logic programming in which formal text and comments can be also formally related through proofs. This paper is focused on the description of the proof system used in the design of such specifications.

It is not the purpose of this paper to review all the formal specification methods based on abstract data types, denotational semantics, mathematics or axiomatic style, nor to investigate their respective merits. Let us only mention that logic programming is also a promising candidate (for a discussion on this aspect see [15], [4], [17], [18]) and recall some of the (good) qualities required for a specification method:

- Good modelling capabilities.
- Facilitating descriptions at the right level of abstraction.
- Subject to validation.
- Self contained and easy to read. (Also possibilities of interfacing with other kinds of specifications).
- Updating and incremental development facilities.
- Easy to test through executable versions of the specification.

We do not intend to discuss all these points which are mostly known. Let us only point out one of them which is the weakness of most of the known methods: the formal text needs to be commented to be readable. The size and the complexity of a formal specification must be appreciated not only by the size of the formal part but also the size of the informal or even formal comments which must be also read in order to understand the specification. This question is particularly relevant when the design objective is precisely to obtain a readable specification. It is the case for example in a standardization work: the result of the work is a specification.

In this paper we present an experimental proof manager system which permits to relate formally the formal part of a specification and its (formal or informal) comments.
The specification method is relational; that is to say the specification language uses normal clauses (definite clauses with possibly negative literals in the body) for its formal part and is aimed at defining relations. It has been presented in [5], [10].
The comments are expressed as logic formulas written with some of the already specified relations and define some local properties that the relations must satify. They introduce some redundancy which helps to understand the specification. They can be viewed as expected properties of the specification. The questions related to the language used to formulate the comments have been addressed in [3]. It will be considered in this paper without loss of generality that, from a theoretical point of view, all properties can be expressed as sets of ground atoms. In practice of course they will be expressed by assertions.
Comments and clauses may be related by proofs of correctness of the clauses with regards to the properties. The proof manager system helps the designer to state properties as comments and to prove the correctness. The proof methods are described in [3], [5], [8]. The proofs are partial and partly automatized. They are not aimed at complete validation of the specification, but rather at debuging it and improving the comments such that its understanding is facilitated.
This approach is illustrated by the example of the design of draft standard Prolog on which this methodology is currently applied ([10], [13]).

The paper is organized as follows:
In the first sections we recall what is the "declarative" semantics of a normal program, called the *actual semantics*, the semantics of the comments, called the *intended semantics* (sections 2, 3), and two proof methods. In the first one the comments are considered as invariants and the proof method is recalled as defined in [5] (section 4). In the second the comments are considered as sets of "input/output formulas" and called *annotation*. The proof method is recalled as presented in [3] with some extensions to handle normal programs (section 5).
The other sections are devoted to the proof manager system: its presentation (section 6) and our experience in its use in the work of developing a formal specification for the draft standard Prolog (section 7).

2 The design of the specification.

Following the methodology proposed by Deransart and Ferrand [4], a specification is designed by:

- a *normal program*, i.e. a finite set of normal clauses, defining at the same time a finite set of packets (a packet is a set of clauses with the same predicate symbol in the head literal). Following the definition of [11], a normal clause is denoted by $p_0(\vec{t_0}) \leftarrow L_1, \ldots, L_n$ where L_i is a *positive literal* (i.e. $L_i = p_i(\vec{t_i})$) or a *negative* one (i.e. $L_i = not\ (p_i(\vec{t_i}))$). Positive literals are called *atoms*. If $n = 0$ then the clause is called an *unit clause* or a *fact*. Given a language $\mathcal{L}$ and a program P, the *Herbrand base* is the set of all the atoms built with the predicates of P and the functors of P and $\mathcal{L}$.

- a set of comments associated to each packet of the specification. The comments are formulas expressed in some language $\mathcal{L}$ which is not restricted, but the comments must be written following a specific schema.

The actual semantics is defined by the meaning given to the set of normal clauses. The (possibly partial) intended semantics is given in another (possibly informal, or even in natural language)

language through the comments. Although called a "program", the formal part of a specification is not necessarily executable. In particular the use of the negation allows us for defining non computable relations (for a deeper discussion see [4]).

2.1 The specification language and its semantics (actual semantics).

We borrow from [5] some results concerning the meaning of a set of normal clauses.
Let P be a normal program. The semantics of P is formalized by two sets of ground atoms: the set of M_P^+ of *true* atoms and the set M_P^- of *false* ones. The complement of these two sets in the Herbrand base is the *undefined* atoms.
Our view of the actual semantics is based on a notion of semi proof-tree: given a normal program P, a semi proof-tree is a proof-tree built with ground instances of clauses such that the positive leaves are ground instance of facts (unit clauses).
Let J be a set of ground atoms. Let $sptr(P, J)$ be the set of the semi proof tree roots such that for each negative leaf $\neg A$, we have $A \in J$. Let now I be a set of ground literals, and let us denote I^+ the set of ground atoms in I and I^- the set of ground atoms B such that $\neg B$ is in I. In [5], it is proven that for all I the following operator Ψ_P:

$$\Psi_P(I) = sptr(P, I^-) \cup \neg \overline{sptr(P, \overline{I^+})}$$

is monotone and its least fixed point M_P coincides with the three valued *well-founded model* described in [14]. The two sets M_P^+ and M_P^- characterizing the actual semantics are then deduced from M_P.
It is naturally comfortable to deal with a semantics which is equivalent to one of the most popular ones at the present time. But the best justification of the semantics based on proof-trees is because it is absolutly well suited to the proof method.
Although all the recalled results concern three-valued semantics (with true, false and possibly some undefined atoms), we will restrict our attention in a formulation of the results for a bi-valued semantics (only true and false atoms).
In fact it will be assumed that the program of the specification is stratified. In this case the actual semantics is bi-valued. A program is stratified if no relation is defined (directly or indirectly) using some negation of itself. Typical example of a non stratified program is the formulation of the "Barber paradox": the Barber is the man who shaves all people which don't shave themselve, or more formally:
$shave(Barber, X) \leftarrow not\ shave(X, X)$.
In the well-founded semantics of [14] the atom $shave(Barber, Barber)$ is undefined, but any atom of the form $shave(Barber, Jack)$ is true.
A formal definition of the stratification is given in [1].

2.2 The comments (partial intended semantics).

Partial correctness and weak completeness: Let P be a normal program and let M_P^+ and M_P^- be respectively the set of true atoms and the set of false ones characterizing its actual semantics. Ideally we could imagine to formalize the intended semantics as a set IS^+ of atoms expected to be true in P and the set IS^- of atoms expected to be false, and then to compare M_P^+ with IS^+, and M_P^- with IS^-, such that true and false atoms coincide respectively.
In practice, during the design of a specification, only some expected and incomplete properties can be formulated, hence such identification is impossible. Only some inclusion is possible (in fact weaker properties define atoms which are supersets of the actual semantics). As the semantics is bi-valued, the set of the false literals is the complement in some Herbrand base of the set of the true ones. So it is sufficient to define the expected properties with regards to the true literals only.
The intended semantics will be formalized by two sets of ground atoms S and C. Assume that S contains all the ground atoms satisfying an intended property. Then we want to have the inclusion $M_P^+ \subseteq S$. P is said to be *partially correct* w.r.t. S iff $M_P^+ \subseteq S$. Conversely P is said to be *weakly complete* w.r.t. a set C of atoms which are expected not to be false (true or undefined) iff $C \subseteq \overline{M_P^-}$.
In our case, as the semantics is bi-valued, $M_P^+ = \overline{M_P^-}$, and the correctness of the program w.r.t. S and C can be expressed by the double inclusion : $C \subseteq M_P^+ \subseteq S$.

To summarize, the validation condition is formulated over

- the sets M_P^+ and M_P^- describing the actual semantics of a normal program P
- the sets S and C characterizing some properties describing a partial intended meaning of P

by the two inclusions:

1. $M_P^+ \subseteq S$ (partial correctness w.r.t the property associated to S)
2. $C \subseteq M_P^+$ (weak completeness w.r.t the property associated to C).

In terms of semi-proof-tree roots, it is formulated by:

1. $sptr(P, \overline{C}) \subseteq S$, called the correctness part
2. $C \subseteq sptr(P, \overline{S})$, called the completeness part

Expressing the comments: In practice we don't give the two sets S and C explicitely (by an enumeration of their elements). We express instead some intended properties by giving some formulas in some language $\mathcal{L}$: each formula characterizes a set of atoms.
The methodology suggests that the intended properties are relations between the arguments of the predicate symbols of P. Thus, to each predicate symbol p in P, we associate the two formulas S^p and C^p in $\mathcal{L}$. S^p and C^p are both relations between the arguments of P which are denoted in the formulas by the free variables.
Let $\mathcal{I}$ be the underlying term interpretation and $\vec{t}$ be a vector of ground terms in the Herbrand base. The translation in terms of sets of ground atoms is done as follows:

$$C = \{\ p(\vec{t})\ /\ \mathcal{I} \models C^p[\vec{t}]\} \tag{1}$$

and

$$S = \{p(\vec{t})\ /\ \mathcal{I} \models S^p[\vec{t}]\} \tag{2}$$

where $S^p[\vec{t}]$ (resp. $C^p[\vec{t}]$) is S^p (resp. C^p) where the free variables are assigned to their corresponding argument in $\vec{t}$.
The sets S and C can be viewed as an approximation of the actual semantics of P.
As we are dealing with ground atoms, $\overline{S}$ and $\overline{C}$ will be characterized respectively by the sets of formulas $\{\ \neg S^p\ \}$ and $\{\neg C^p\ \}$ for every p in P. This is illustrated in the example $ninclud(L1, L2)$ of the next section.
Notice that the comments may or not be formal. In [10] a methodology is exposed for organizing the writing of P with its comments following a hierarchical notion of *layers*. The idea is that all the relations defined in some layer use in their comments relations defined in the same or in the previous layers only. The notion of layer should not be confused with modules. It is just a trick to go from the simplest relations to the most complex ones when designing the relations and the associated comments.

2.3 An example.

Here is a short example borrowed from [10] illustrating a hierarchical organization of a specification:

*** *First layer*: very simple relations. The comments are totally informal.

is-an-integer(N) : N is an integer built by the constant *zero* and the function *succ*.
is-an-integer$(zero)$ —
is-an-integer$(succ(I))$ — *is-an-integer*(I).

is-a-list(L) : L is a list built by any term of the language as elements and 'nil' and '.' as function symbols.
is-a-list(nil) —
is-a-list$(X.L)$ — *is-a-list*(L).

* * * *Second layer*: more complex relation. The comments use some formally defined predicates.

$plus(X, Y, Z)$: $is\text{-}an\text{-}integer(Y) \Rightarrow Z$ is the sum of X and Y (axioms of the addition).
$plus(zero, X, X) \leftarrow$
$plus(succ(X), Y, succ(Z)) \leftarrow plus(X, Y, Z)$.

$member(X, L)$: $is\text{-}a\text{-}list(L) \Rightarrow X$ is one of the elements of L.
$member(X, X.L) \leftarrow$
$member(X, Y.L) \leftarrow member(X, L)$.

* * * *Third layer*: the most complex relations.

$is\text{-}a\text{-}list\text{-}of\text{-}integers(L)$: $is\text{-}a\text{-}list(L) \wedge (\forall X, member(X, L) \Rightarrow is\text{-}an\text{-}integer(X))$.
$is\text{-}a\text{-}list\text{-}of\text{-}integers(nil) \Leftarrow$
$is\text{-}a\text{-}list\text{-}of\text{-}integers(N.L) \Leftarrow is\text{-}an\text{-}integer(N), is\text{-}a\text{-}list\text{-}of\text{-}integers(L)$.

$list\text{-}sum(L, S)$: $is\text{-}a\text{-}list\text{-}of\text{-}integers(L) \Rightarrow is\text{-}an\text{-}integer(S)$ and S is the sum of all the elements of L.
$list\text{-}sum(nil, zero) \leftarrow$
$list\text{-}sum(X.L, S) \leftarrow list\text{-}sum(L, M), plus(M, N, S)$.

3 Proof method based on invariants.

In this section, we borrow from [5] the proof method. The basic result stands in the following theorem which gives a sound and complete method for normal programs:

3.1 Theoretical results.

Theorem 3.1 *A normal program P is*
partially correct *w.r.t. S and* weakly complete *w.r.t. C,*
i.e. $M_P^+ \subseteq S$ and $C \subseteq \overline{M_P^-}$
iff
there exists S', C' such that $S' \subseteq S$ and $C \subseteq C'$ and the validation condition *of P w.r.t S', C' is satisfied.*

The *validation condition* of P w.r.t. S, C has two parts : called *correctness part* and *completeness part* which are now defined and can be proven separately. For this we need two additional definitions:

Definition 3.1 (Bottom Up Closure) *A* normal program *P is bottom-up closed w.r.t. S, C if for every ground instance $A \leftarrow L_1, \cdots, L_n$ of a clause of P the following holds :*
if, for all positive L_i, $L_i \in S$
and, for all negative $L_i = \neg A_i$, $A_i \in \overline{C}$
then $A \in S$.
(In particular if $n = 0$ this amounts to $A \in S$).

Definition 3.2 (Top Down Closure) *A* normal program *P is top-down closed w.r.t. C, S if there exists a function f defined on C into a well ordered domain (relation denoted $<$) such that, for every atom B of C, there is a ground instance $A \leftarrow L_1, \cdots, L_n$ of a clause of P such that $B = A$ and*
(i) for all positive L_i, $L_i \in C$
and, for all negative $L_i = \neg A_i$, $A_i \in \overline{S}$
(ii) for all positive L_i, $f(L_i) < f(A)$.
The condition (ii) is called decreasing criterion.

Both conditions are *local* : there is exactely one assertion to prove in every clause for each property. The two following theorems are proven in [5] and provide definition and proof method for the validation conditions.

Theorem 3.2 (Proof method for the correctness part) *A* normal program P *satisfies the* correctness part *for* S *and* C *if and only if there exists* S' *stronger than* S *(*$S' \subseteq S$ *) such that* P *is bottom-up closed for* S', C.

Theorem 3.3 (Proof method for the completeness part) *A* normal program P *satisfies the* completeness part *for* C *and* S *if and only if there exists* C' *weaker than* C *(*$C \subseteq C'$ *) such that* P *is top-down closed for* C', S.

The two notions of weaker and stronger specifications, explained above in terms of inclusions between sets can be defined by means of logical implications between the corresponding assertions: Assume that the set of atoms S (resp. S') for the program P is infered from the set of assertions $\{ S^p \}$ (resp. $\{ S'^p \}$) expressed in a language $\mathcal{L}$ with the interpretation $\mathcal{I}$, then:

S' is stronger than S iff for every predicate p $\quad \mathcal{I} \models S'^p \Rightarrow S^p$, and
S' is weaker than S iff for every predicate p $\quad \mathcal{I} \models S^p \Rightarrow S'^p$.

3.2 Example.

We borrow from [5] the following example:
Let P be the stratified program :

$$\begin{array}{lll} \mathit{includ}(L_1, L_2) & \leftarrow & \mathit{not}\ \mathit{ninclud}(L_1, L_2) \\ \mathit{ninclud}(L_1, L_2) & \leftarrow & \mathit{elem}(E, L_1), \qquad \mathit{not}\ \mathit{elem}(E, L_2) \\ \mathit{elem}(E, [E|L]) & \leftarrow & \\ \mathit{elem}(E, [H|L]) & \leftarrow & \mathit{elem}(E, L) \end{array}$$

Expected properties :

	S	C
$\mathit{includ}(L_1, L_2)$	L_1, L_2 *lists* $\Rightarrow L_1 \subseteq L_2$	L_1, L_2 *lists* $\wedge\ L_1 \subseteq L_2$
$\mathit{ninclud}(L1, L2)$	L_1, L_2 *lists* $\Rightarrow L_1 \not\subseteq L_2$	L_1, L_2 *lists* $\wedge\ L_1 \not\subseteq L_2$
$\mathit{elem}(E, L)$	L *list* $\Rightarrow E \in L$	L *list* $\wedge\ E \in L$

Example of a verification for bottom up closed on the (ground) instance
$\mathit{ninclud}(L_1, L_2) \leftarrow \mathit{elem}(E, L_1), \mathit{not}\ \mathit{elem}(E, L_2)$:
Suppose $\mathit{elem}(E, L_1) \in S$ and $\mathit{elem}(E, L_2) \notin C$
We have to verify that $\mathit{ninclud}(L_1, L_2) \in S$ i.e. L_1, L_2 *lists* $\Rightarrow L_1 \not\subseteq L_2$
Suppose L_1, L_2 *lists*
Since $\mathit{elem}(E, L_1) \in S$ we have $E \in L_1$
Since $\mathit{elem}(E, L_2) \notin C$ we have $E \notin L_2$
So $L_1 \not\subseteq L_2$ q.e.d.

Example of a verification for top down closed (i) on the (ground) atom $\mathit{ninclud}(L_1, L_2) \in C$:
We have to check that there is a (ground) clause instance
$\mathit{ninclud}(L_1, L_2) \leftarrow \mathit{elem}(E, L_1), \mathit{not}\ \mathit{elem}(E, L_2)$
with $\mathit{elem}(E, L_1) \in C$ and $\mathit{elem}(E, L_2) \notin S$.
Since $\mathit{ninclud}(L_1, L_2) \in C$, L_1, L_2 are *lists* so it remains to find a ground term E such that $E \in L_1$ and $E \notin L_2$.
Such a term exists because $L_1 \not\subseteq L_2$ since $\mathit{ninclud}(L_1, L_2) \in C$. q.e.d.

Example of a verification for top down closed (ii) :
Only the clauses of the predicate *elem* are concerned. There is an easy decreasing criterion: the length of the second argument, which is a ground list.

4 Modular proof based on annotations.

4.1 Performing the correctness part with annotations.

In [3] a new proof method is presented to establish the correctness part of the validation conditions: the proof method by annotations. It is studied for definite programs and an extension to stratified

programs is proposed in [12]. The method by annotations is a refinement of the above method, adapted from the Attribute Grammar's field. By annotations, proofs become more modular and tractable, and thus it is a factor of simplification. Both methods are proven sound and complete, hence they have the same theoretical power. In practice, the validation task for the correctness part is performed by annotations in a semi-automatic way.

4.2 Principles of the method.

To facilitate a good understanding, we give an informal presentation of the method with a simple example.
Let S be a set of assertions written in a language with the interpretation $\mathcal{I}$, associated with the normal program P. We now assume that every formula S^p of S has the form of implication $A_1^p \wedge A_2^p \wedge ... \wedge A_m^p \rightarrow B_1^p \wedge B_2^p \wedge ... \wedge B_n^p$ where $m, n \geq 0$. The formulas $A_1^p ... A_m^p$ are called the inherited assertions of p while the formulas $B_1^p ... B_n^p$ are called the synthesised assertions of p. An alternative way of defining specification is thus to associate with each predicate p a finite set of formulas (where all the free variables correspond to the argument of p) and to divide the formulas into *inherited* assertions and *synthesised* assertions. The partitionned assertions associated to the predicates will be called *annotation*. Now the assertions can be thought of as attributes of the predicates. The annotation proof method can be summarized as follows.
For each clause create the assertion instances by replacing variables in predicate assertions by corresponding argument terms of the occurrences of predicate arguments. They can be divided into input assertions and output assertions, as follows: the input assertions are the inherited assertions of the head and the synthesised assertions of the body atoms. The remaining assertions are output assertions. The idea of correctness proof is to show that in every clause each output assertion is implied by a (possibly empty) subset of input assertions. This can be described by a local dependency relation imposed on the assertions of a clause: some output assertion depends on an input assertion if the input assertion must be used to prove it.
It is required that the corresponding dependency scheme is non-circular, i.e. in all the partial semi-proof trees built with clause instances, there is no instance of some assertion which depends on itself, following the local dependencies in all the clause instances used to build this partial semi-proof tree. A program P is partially correct with respect to the specification given in the form of annotation if

- For every clause every output assertion is valid in the interpretation considered providing that the input assertions on which it depends are valid.
- The dependency scheme is noncircular.

For a complete presentation of the method see [3].

4.3 Example.

We use the annotation method to prove that the following program is well-typed.

$$
\begin{array}{lll}
leng([], zero) & \leftarrow & \\
leng([A|L], N) & \leftarrow & leng(L, M) \quad plus(s(zero), M, N) \\
plus(zero, X, X) & \leftarrow & \\
plus(s(X), Y, s(Z)) & \leftarrow & plus(X, Y, Z)
\end{array}
$$

with the following specifications $S = \{S^{plus}, S^{leng}\}$, where

S^{plus} : $Plus3 = Plus1 + Plus2$ and
S^{leng} : $Leng2 = length(Leng1) \wedge is\_a\_list(Leng1)$

and $Plus3$ denotes the third argument of *plus*. $Plus2$ the second argument, ...
The symbols *length* , = and + denote functors and relations used in the assertion language $\mathcal{L}$ whose obvious interpretation is suggested by the names. Let *is_an_integer* and *is_a_list* be two others relations of $\mathcal{L}$ with the obvious interpretation $is\_an\_integer(0), is\_an\_integer(1), ...$ and $is\_a\_list([]), is\_a\_list([t]), ...$ t beeing any integer or term.
We consider the following annotation Δ for *plus* and *leng* :

- The inherited assertions are:

  $is\_an\_integer(Plus2)$.

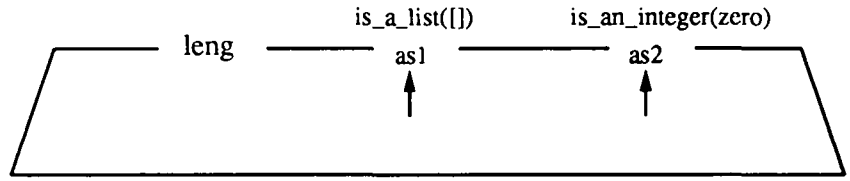
- The synthesised assertions are:

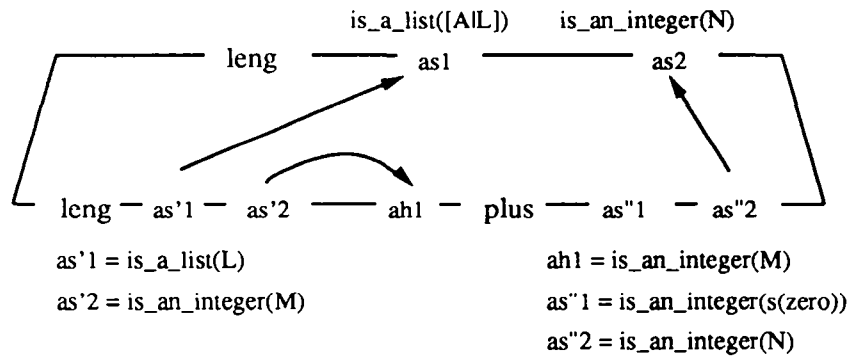
  $is\_an\_integer(Plus1)$, $is\_an\_integer(Plus3)$, $is\_a\_list(Leng1)$, $is\_an\_integer(Leng2)$.

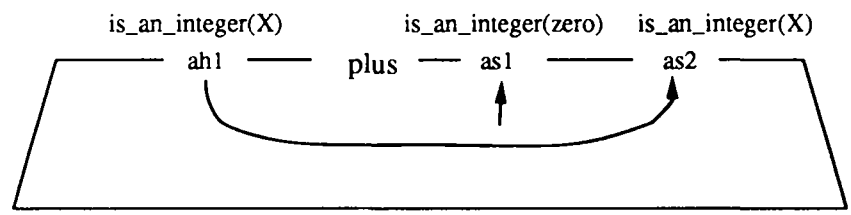
This annotation is not the direct translation of the specification S, i.e. the validation of Δ will not imply the validity of the specifications S, but the validity of some weaker specifications S':

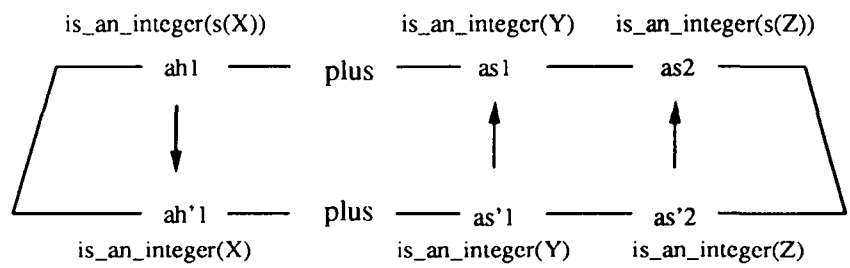
S'^{plus} : $is\_an\_integer(Plus2) \Rightarrow is\_an\_integer(Plus1) \wedge is\_an\_integer(Plus3)$
S'^{leng} : $is\_a\_list(Leng1) \wedge is\_an\_integer(Leng2)$

For each clause, we summarize in the following schema the dependencies between the input and the output. The dependency relation is materialized by the arrows. The arrows without antecedent mean that the corresponding output formula must be proven without hypothesis. The inherited (resp. synthesized) assertions appear on the left (resp. right) side of the associated predicate name.









The proof is performed by showing for each clause and for each output assertion that it is valid in the interpretation providing that the input assertions on which it depends are valid. For example, for the fourth clause of the program one has to prove the validity in $\mathcal{N}$ of the following implications:

- $is\_an\_integer(X) \Rightarrow is\_an\_integer(s(X))$
- $is\_an\_integer(Y) \Rightarrow is\_an\_integer(Y)$
- $is\_an\_integer(Z) \Rightarrow is\_an\_integer(s(Z))$

which follows immediately by the definition of the interpretation (assuming that the interpretation domain is the naturals, that $is\_an\_integer(X)$ is true if X is a natural, and s is the successor function in $\mathcal{N}$).
The proofs for other clauses are also very easy.
The other component of the annotation proof is checking non-circularity of the dependency scheme used in the first part of the proof. This can be done automatically using the algorithms developed for attribute grammars (see [7] [5] for more details). In our example the dependency scheme is noncircular hence the annotation is valid.

5 The proof manager system.

The actual implementation (in Prolog) of the annotations proof method is comprehensively described in [12] and illustrated with examples of some interactive sessions.

5.1 Principles.

Input: The system requires for input a file containing the clauses of the program and a file containing the associated annotation to be validated. The verification of the stratification can be done automatically thanks to a program written by A.Eddbali which verifies the condition and which computes the minimal stratification (see [9]).

The annotation is written as a collection of facts giving for every predicate the set of its formulas with their name and their mode (inherited or synthesized).

Construction of the LDS: The first part of the treatment consists in choosing the dependencies between the formulas in every clauses. It is done sequentially, clause by clause, as follows:

- For the clause c, the system computes the set of output formulas $Output(c) = \{\beta_i\}_i$ and the set of input ones $Input(c) = \{\alpha_j\}_j$.
- For every β_i, it computes $I(c, \beta_i)$ a subset of $Input(c)$ containing all the formulas sharing (directly or indirectly) a common variable with β_i.
- If the set $I(c, \beta_i)$ contains a formula α_j which is exactly the formula β_i, then the system builds automatically for the output β_i the unique dependency: $\alpha_j \rightarrow \beta_i$. Else, the user is sollicited to select in $I(c, \beta_i)$ a subset (possibly empty) $\{\alpha_k\}_k$ and then the system infers all the corresponding dependencies $\{\alpha_k \rightarrow \beta_i\}_k$. Notice that it is possible to choose an option for a non interactive session, i.e. such that the user is never sollicited to select the dependencies. In that case, the system takes by default $\{\alpha_k\}_k = I(c, \beta_i)$. The problem is that in general all the formulas of $\{\alpha_k\}_k$ are not useful to prove the soundness in the clause. Hence too many dependencies are generated and it increases the chances to introduce a cycle in the LDS. Moreover, an interactive session enables the user to oversee step by step (dependency by dependency) the advancement of the proof and to operate some modifications in the annotation at each step of the proof.
- When every output has been considered, the same treatment is iterated for the following clause, *and so on until every clause has been taken into account*.

Modification of the annotation during the proof: Evidently, for every output, the user will select the input formulas which are pertinent with regards to the conditions of soundness and well-formedness that the LDS must verify.

If he does not find some suitable formulas, or not enough, the system offers to him the possibility to operate some modifications over the annotation by deleting or adding some formulas. Then, the modifications are taken on board into a incremental way: the system reconsiders all the lapsed dependencies until the reactualisation of the LDS is established.

Output: When the LDS is completely built, the system has produced a file containing all the dependencies. For every clause and every output β_i, it has generated an implicative formula of the form $\bigwedge\{\alpha_k\}_k \Rightarrow \beta_i$. Among these formulas, some are of the form $\alpha_j \Rightarrow \beta_i$ with $\alpha_j = \beta_i$. Then their validity is trivially obtained. It remains to check the validity of all the others formulas and to check the well-formedness of the LDS.

Checking the well-formedness: This task consists in a test of non circularity over the graph of the dependencies. This is done automatically. The test is intrinsically exponential but some non trivial subclasses of LDS can be decided in polynomial time [1]. Our test is based on an algorithm given in [6] and we have adapted for our purpose an implementation of the algorithm by J.L.Bouquard (see [2]).

Checking the soundness: This part is not automatized: for this purpose, we would need a theorem prover associated with the chosen assertion language. Our system only validates automatically the trivial formulas (this is illustrated in the following example of a session). The user has to handle "manually" with all the other formulas to establish their validity. In practice, we try to evaluate a best way to write an annotation such that the part of trivial formulas generated by the dependencies is the biggest one. Two criterias are very influent for such a purpose: the first one is the number of formulas resulting from the splitting of the initial assertion into inherited and synthesized smaller ones. It is clearly more comfortable to handle with many little formulas. The second is the reusability of the small formulas: we must try when splitting the initial assertions to use some relations that have already been used for the splitting of a previous formula.

5.2 Example of a session.

Let us describe what would be an interactive session corresponding to the above studied example. The message displayed by the system are written with the font: `font`. The answers given by the user are prefixed by ":" and use the usual font. The comments of the session are prefixed by % or written with the font: *font*.

Beginning of the session.

```
>> Do you want to see the program (Y/N) ?
```

: Y

```
(1) :  leng([],zero).
(2) :  leng([A|B],C) :-
           leng(B,D).
           plus(s(zero),D,C).
(3) :  plus(zero,A,A).
(4) :  plus(s(A),B,s(C)) :-
           plus(A,B,C).
```

```
>> Do you want to see the annotation (Y/N) ?
```

: Y

```
>> For which predicate ?
(enter ''P/N'' for predicate P with arity N, and ''all'' for all predicates).
```

: all

[1] [7] contains a survey of the main results related to the question of circularities.

```
No inherited assertion for predicate leng/2.
Synthesized assertions for predicate leng/2:
    as1 :  is_a_list(Leng1)
    as2 :  is_an_integer(Leng2)
Inherited assertions for predicate plus/3:
    ah1 :  is_an_integer(Plus2)
Synthesized assertions for predicate plus/3:
    as1 :  is_an_integer(Plus1)
    as2 :  is_an_integer(Plus3)
```

```
>> Do you want to modify the annotation (Y/N) ?

: N

*** *** Begining of the construction of the Logical Dependencies Scheme: *** ***
```

```
clause (1)...
```

Nothing appears on the screen for the treatment of the first clause: the dependencies are infered automatically because all the output formulas also appear in the input.

```
clause (2)...
```

The corresponding clause is displayed.

```
leng([A|B],C) :-
    leng(B,D),
    plus(s(zero),D,C).
```

The formula for the first output is given, together with the corresponding assertion (its name and its position in the clause. 0 for the head, 1 for the first literal in the body, ...).

```
output = is_a_list([A|B])                    ( = as1 / 0)
```

The pertinent subset of input is given. The others input are given (optionnally) just for further informations.

```
Suggested input:
    1 :  is_a_list(B)                        ( = as1 / 1)
Others input:
    2 :  is_an_integer(C)                    ( = as2 / 2)
    3 :  is_an_integer(s(zero))              ( = as1 / 2)
    4 :  is_an_integer(D)                    ( = as2 / 1)
```

```
>> Enter the number of the chosen input (return if no selection).

: 1
```

```
clause (3)... same situation as in the clause (1).
clause (4)... same treatment as in the clause (2).
```

```
plus(s(A),B,s(C)) :-
    plus(A,B,C).
```

```
output = is_an_integer(s(A))                ( = as1 / 0)

     Suggested input:
         1 :  is_an_integer(A)                   ( = as1 / 1)
     Others input:
         2 :  is_an_integer(C)                   ( = as2 / 1)
         3 :  is_an_integer(B)                   ( = ah1 / 0)

>> Enter the number of the chosen input (return if no selection).

: 1

output = is_an_integer(s(C))                ( = as2 / 0)

     Suggested input:
         1 :  is_an_integer(C)                   ( = as2 / 1)
     Others input:
         2 :  is_an_integer(A)                   ( = as1 / 1)
         3 :  is_an_integer(B)                   ( = ah1 / 0)

>> Enter the number of the chosen input (return if no selection).

: 1
```

End of the construction.

Presentation of the resulting dependencies scheme. For every clause, all the dependencies are given, and below the associated formulas. The trivial formulas are recognized and prefixed by an asterisque.

```
***                                                                    ***
***  List of the constructed dependencies and their associated formulas:  ***
***                                                                    ***

        >>>>>> Dependencies for clause (1) :
            (ah1,0) ⟶ (ah1,1)
            (as2,1) ⟶ (as2,0)
            (as1,1) ⟶ (as1,0)
        <<<<<< Formulas to validate:
            (*) is_an_integer(A) ⇒ is_an_integer(A)
            is_an_integer(A) ⇒ is_an_integer(s(A)) % occurs twice.


        >>>>>> Dependencies for clause (2) :
            (ah1,0) ⟶ (as2,0)
        <<<<<< Formulas to validate:
            (*) is_an_integer(A) ⇒ is_an_integer(A)% occurs twice.
            is_an_integer(zero)


        >>>>>> Dependencies for clause (3) :
            (as2,1) ⟶ (ah1,2)
            (as2,2) ⟶ (as2,0)
            (as1,1) ⟶ (as1,0)
        <<<<<< Formulas to validate:
            (*) is_an_integer(A) ⇒ is_an_integer(A)
            is_a_list(A) ⇒ is_a_list([B|A])
```

```
>>>>>> No dependency for the clause (4)
<<<<<< Formulas to validate:
    is_an_integer(zero)
    is_a_list([])
```

The test of well-formedness of the LDS is performed over the file where all the dependencies have been saved. It can be required by the user before the verification of the soundness, or at the end of the session.

5.3 Using the system.

We hope that the above example is significant enough to justify our choice to deal in practice with the annotations method. While the formulation of the Bottom Up Closure generates one formula per clause (which is manually untractable as soon as they are many litterals in the body of the clause), the corresponding formulation of the soundness by an annotation generates many little formulas. Moreover, we may expect than a subset of these formulas will be automatically proven as it is the case for some of them in the example.
Actually, the validation task in practice must be seen as an incremental design of the initial annotation and the properties to prove. We can imagine assuming some improvements of the validation task as follows: we begin a session with an initial program and its initial associated annotation, then during the execution of the proof we operate some converging changes on the program or the annotation such that we achieve the session with a corrected program and a valid annotation.

6 Empirical results for Prolog Standard.

6.1 Presentation of the studied specification.

Our most significant experience of validation through this proof method and its implementation concerns the elaboration of a formal specification for draft standard Prolog (cf. [13]).
This formal specification has been written following the methodology presented in this paper: it consists in a set of normal clauses with their associated comments. The normal program has five levels of stratification. It is not an executable specification, because some of the relations are not completely specified. The whole specification is about four hundred clauses. The comments are partly formal and use some relations defined inside the specification. The design part of the specification has been exposed and discussed in [16], [10].

6.2 Performing some correctness proofs.

The aim of the formal specification is to specify the operationnal semantics of standard Prolog. It consists in describing the evolution of the search tree during the execution of an initial goal over the initial database. The abstract representation of the database, the goal, the search tree, the environment, ... are described in a part of the formal specification which is called the *data structures*. References to data structures appear frequently inside the comments. For example the main predicate of the specification and its associated comment are:

> *semantics*(P, G, E, T) : " **if** *P is a database, G is a goal, and E is an environment* **then** *T is a PVST (Partial Visited Search Tree) up to some node which is any leaf before or on the first infinite branch or a CVST (Complete Visited Search Tree) if there is no infinite branch*".

In this comment, one refers to data structures *is-a-database*(P), *is-a-goal*(G), and *is-an-environment*(E), which are formally described in the specification. Obviously, the last part of the comment cannot be axiomatized through a data structure because it is precisely the aim of the four hundred clauses of the specification to describe what is the associated search tree like. Consequently, this part of the comment remains informal, and during the elaboration of the proof we have to handle with this

informal assertion. When performing the proof, all the informations contained into the comments, (and translated into some formal or informal assertions) will be required. When checking the soundness, it may happen that some formulas are trivial as it is the case in the previous studied example. Hence those formulas are proven automatically, and by the way it reduces the manual tasks of the proof. Obviously we are looking forward that such a situation occurs the most often, and a good criterion is to reduce the set of different assertions belonging to the annotation.
For example, it is possible to describe some features common to all the search trees, (independently of the associated P, G, and E). They all have a same structure, which has been formally defined by the relation *is-a-forest*(F), i.e. it is the chosen abstract representation for the search tree.
In practice, it will be easier to perform proofs of weaker properties of *semantics*(P, G, E, T)), like : *is-a-database*(P) $\wedge$ *is-a-goal*(G) $\wedge$ *is-an-environment*(E) $\Rightarrow$ *is-a-forest*(F)
As all the data structures are defined formally we may expect to improve the automatized parts of the proof. It is also the fact that thanks to its weakness, the property is likely to be proven without requiring some assumption over the negative literals. If it is the case, i.e. if the properties required for every predicates to validate the initial property are weak enough to be validated without assuming any completeness results for the negative literals, then the correctness part can be achieved by itself. In other words, it could enable us to establish the simple inclusion $M_P^+ \subseteq S$, considering another set $C = \emptyset$ (or the assertion "false") which satisfies without further proof $C \subseteq M_P^+$.
We illustrate some interesting situations in the following examples.

6.3 Examples.

Example 1: We give here an extract of a session over the formal specification with a typing annotation (we assume that the classical notions of forest of trees and Dewey-like notation of the tree nodes are known).

```
Inherited assertions for predicate buildforest/3 :
   ah1 :  is-a-forest(Buildforest1)
   ah2 :  is-a-dewey-number(Buildforest2)
Synthesized assertions for predicate buildforest/3 :
   as1 :  is-a-forest(Buildforest3)
Inherited assertions for predicate treatment/3 :
   ah1 :  is-a-forest(Treatment1)
   ah2 :  is-a-dewey-number(Treatment2)
Synthesized assertions for predicate treatment/3 :
   as1 :  is-a-forest(Treatment3)


clause (3)...
      buildforest(A,B,C) :-
          not(completely-visited-tree(A,B)),
          treatment(A,B,C).


>>>>>> Dependencies for clause (3) :
    (as1,2) —— (as1,0)
    (ah1,0) —— (ah1,2)
    (ah2,0) —— (ah2,2)
<<<<<< Formulas to validate:
    (*) is-a-forest(A) ⇒ is-a-forest(A)
    (*) is-a-forest(A) ⇒ is-a-forest(A)
    (*) is-a-dewey-number(A) ⇒ is-a-dewey-number(A)
```

Here, the three dependencies of the LDS are generated automatically by the system, and they give rise to three trivial implicative formulas. We can observe that no information is required for the negative literal.

Example 2: Following the same idea of avoiding the use of informations over the negative literals, and proving nevertheless some interesting properties, we can deal with groundness properties. For instance, on the previous clause we could handle with the following annotation:

```
Inherited assertions for predicate buildforest/3 :
   ah1 :  ground(Buildforest1)
   ah2 :  ground(Buildforest2)
Synthesized assertions for predicate buildforest/3 :
   as1 :  ground(Buildforest3)
Inherited assertions for predicate treatment/3 :
   ah1 :  ground(Treatment1)
   ah2 :  ground(Treatment2)
Synthesized assertions for predicate treatment/3 :
   as1 :  ground(Treatment3)
```

It is very easy to see that the soundness of the LDS for the studied clause with the above annotation holds trivially.
Imagine now that we add to this annotation an inherited assertion **ground(Not1)** for the predicate **not**. Then it creates a new output in the clause: **ground(completely-visited-tree(A,B))**. This formula is easily validated by the input formulas **ground(A)** and **ground(B)** associated with literal **buildforest(A,B,C)**.
This gives the suggestion that properties of groundness can easily and automatically be proven with an annotation.
To summarize, we have experimented two kinds of proofs:

- the validity of an annotation corresponding to an exact translation of the comments. The assertions are partly formal and partly informal. The partial correctness property which is the purpose of the proof is the strongest one, but all the assertions of completeness required for all the negative literals are assumed. At the end of the session, about 30% of the implicative formulas are proven. The others must be proven manually.

- the validity of an annotation corresponding to a formal translation of some weaker property included in the comments: only the informations concerning the data structures are kept. Nearly no information of completeness is required, so nearly no assumption of completeness results is done. About 70% of the implicative formulas are proven automatically by the system.

6.4 Discussion

Each of this two kinds of proof can be seen as a part of the whole proof we are looking forward to hold definitely. They bring together some new informations allowing us to be convinced, more and more of the correctness of the program.
Notice that these two kinds of proofs suggest a methodology to improve some further investigations: begining with an annotation of the second type, the idea would be to refine it stedely, adding progressively some new formulas over negative literals, such that the associated completeness properties are as strong as possible such that their proof (which can be performed by top-down closure only) remains easy.
Our experience shows us the capital importance of the choice of the comments. Especially, it is useful to focuse on the "reusability" of the assertions chosen when writing the comments. For each comment, we must try to find a formulation of the intended meaning such that the assertions used for it have been already used elsewhere in other comments. Following this criterion, we may probably be led to translate an assertion into an equivalent conjunction of smaller ones, already used elsewhere, but with the perspective to simplify the proof afterwards.

7 Conclusion.

We have presented a methodology to make formal specifications with logic programming. We don't claim of course that this way of specifying is more powerful than others nor well suited to any kind of specifications. It is still limited to relatively short specifications as it is unimodular. But it has interesting features in particular the possibility for the specifier to perform a validation work over the specification by a formal proof method. This approach guarantees until some extend the good adequation betwen comments and formal text, and as a result the size of the whole specification may be reduced. However the comments describe relatively local properties, and we don't claim that all the comments which may be desirable can be described in such a way. The tendency is thus to add general comments (i.e. comments which express very general properties) as a separate part of the specification.

We have focused on the manager proof system which permits to handle big proofs of local properties in an incremental way. The proofs stand in two parts: the correctness part and the completeness part. The existence of a more tractable formulation for the correctness part, simplifies the performance of this part of the proof. Moreover, its implementation enables us to realize a part of the proof into a semi automatical way.
At present time, we have no refinement for the completeness part and consequently this part of the proof is more complicated to achieve in practice. Although only some aspects of the proofs can be handled in practice, the system has already demonstrated its hability in helping to perform proofs on relatively big pieces of software, which are untractable manually.

It should be also noticed that this methodology is not limited to the use of the proof management system to improve the specification and its comments. It permits also to investigate other properties. It inherits also many attributes of a good specification method, in particular it is relatively easy to produce an executable specification which respects the actual semantics of the specification. More elements on this topic can be found in [4].

This methodology has been investigated for the elaboration of a formal specification for the standardization of Prolog and it has already shown its usefulness by helping to remove innumerous bugs in the formal specification, its comments and consequently in the whole text of the draft standard ([9]).
Some future improvements could be introduced at some different levels of the methodology on both practical or theoretical sides. The methodology has a wide scope, hence the possible improvements would be pertinent in many directions: the writing of the comments, the formulation of the validation conditions, the exploitation of an helpful environement for validation, and the increasing of the automatized part.

References

[1] Apt K. R.: *Introduction to Logic Programming.* TR-87-35, Department of Computer Sciences, The University of Texas at Austin, September 1987.

[2] Bouquard J.L.: *Links between Attribute Grammars and Logic Programming* (in French). PhD thesis. University of Orléans, 180 p. (January 1992)

[3] Deransart P.: *Proof Methods of Declarative Properties of Definite Programs.* INRIA RR 1248, 1991 (to appear in Theoretical Computer Science).

[4] Deransart P., Ferrand G.: *An operational Formal Definition of Prolog : a specification method and its application. New Generation Computing*, 10 (1992) 121-171.

[5] Deransart P., Ferrand, G.: *Proof Method of Partial Correctness and Weak Completeness for Normal Logic Programs.* Proceedings of the JICSLP'92, Washington DC, Nov 1992.

[6] Deransart P., Jourdan M., Lohro B.: *Speeding Up Circularity Tests for Attribute Grammars.* RR INRIA (1983).

[7] Deransart P., Jourdan M., Lohro B.: *Attribute Grammars : Definitions, Systems and Bibliography* LNCS **323**, Springer Verlag, (Aug 88).

[8] Deransart P., Maluszinski J.: *Grammatical View of Logic Programming.* The MIT Press, (to appear) 1993.

[9] Ed-Dbali A.: *Formal and Executable Specification in Logic Programmation: Application to the standardization of Prolog* (in French). PHD thesis, University of Orléans, February 1993.

[10] Ed-Dbali A., Deransart P.: *Software Formal Specification by Logic Programming: The example of Standard PROLOG.* Logic programming in action, LNIA 626 Springer Verlag, LPSS'92, Zürich (Sep 1992).

[11] Lloyd J. W.: *Foundations of logic programming.* Springer-Verlag, Berlin, (1984)

[12] Renault S.: *Logic Programs Validation.*(in French) University of Paris 7, DEA Report, 65p (Sep 1991).

[13] Scowen R. (ed.): PROLOG Part 1 - General Core. ISO/IEC JTC1 SC22 WG17 **N92** (Mar 92).

[14] Van Gelder A., Ross K.A., Schlipf J.S.: *The well-founded Semantics for General Logic Programs.* Journal of the ACM, Vol. 38, No. 3, July 1991, 620-650.

[15] Fuchs N.E.: *Specifications Are (Preferably) Executable* Software Engineering Journal, September 1992.

[16] Richard G.: *Contribution to the realisation of a formal specification for PROLOG* (in French). PHD thesis, University of Orléans, June 1989

[17] Deville Y.: *Logic Programming: Systematic Program Development.* Addison-Wesley, International Series in Logic Programming, March 1990.

[18] Sterling L.: in *Logic Programming in Software Engineering.* Tutorial N° 7 - 8th ICLP. Inria. Paris (June 1991)

Unité de Recherche INRIA Rocquencourt
Domaine de Voluceau - Rocquencourt - B.P. 105 - 78153 LE CHESNAY Cedex (France)

Unité de Recherche INRIA Lorraine Technopôle de Nancy-Brabois - Campus Scientifique
615, rue du Jardin Botanique - B.P. 101 - 54602 VILLERS LES NANCY Cedex (France)
Unité de Recherche INRIA Rennes IRISA, Campus Universitaire de Beaulieu 35042 RENNES Cedex (France)
Unité de Recherche INRIA Rhône-Alpes 46, avenue Félix Viallet - 38031 GRENOBLE Cedex (France)
Unité de Recherche INRIA Sophia Antipolis 2004, route des Lucioles - B.P. 93 - 06902 SOPHIA ANTIPOLIS Cedex (France)

EDITEUR
INRIA - Domaine de Voluceau - Rocquencourt - B.P. 105 - 78153 LE CHESNAY Cedex (France)

ISSN 0249 - 6399