A FIRST ORDER SEMANTICS OF A CONNECTIVE SUITABLE TO EXPRESS CONCURRENCY

Pierpaolo Degano Stefano Diomedi

Dipartimento di Informatica

Universita' degli Studi di Pisa

Abstract. The paper presents an extension to PROLOG that allows to directly express concurrency and synchronization. This is achieved by introducing the concept of class, a sort of cluster made of concurrent atoms. In general, a set of clauses involving classes is equivalent to a denumerable infinite set of pure PROLOG clauses. First, syntax and operational semantics of our extension are defined. Then a first order semantics is given that slightly generalizes classical PROLOG model-theoretic semantics; a fixpoint semantics is also given. Finally, an example illustrate the expressive power of the extension.

1. INTRODUCTION

Recent achievements in hardware technology made it feasible the development of machines that can directly execute logic programming languages. Among these, PROLOG is the most relevant both for theoretical and for practical reasons [2,6]. However, PROLOG is not satisfactory enough to conveniently express the concurrent features that hardware provides nowadays. As a matter of fact, PROLOG procedures can be naturally executed either in a parallel or in a co-routining fashion. The former regimen is simply achieved by simultaneously replacing a set of independent atoms in the current goal. Co-routining occurs when the same variables are shared by different atoms, thus realizing a sort of asynchronous communication. Unfortunately, there is no explicit way of synchronizing the computations of two or more concurrent processes, as is required when they cooperate to solve the same problem.

In order to solve this limitation, a number of extensions to PROLOG have been introduced [3,4,7,9]. All these extensions allow to write clauses with more than one atom in their left-hand side, e.g.

A(x,y) & B(y,z) <-- C(x,y,z), D(z,w)

where variable y acts as a synchronous communication channel between atoms A and B. The intended operational meaning of such a clause is that suitable instances of atoms C and D can be replaced for an instance of A and B, only when both of them are present at the same time in the goal.

The aim of this paper is to give a formalization of the above operational meaning within a logic framework, so that all the appealing semantic features of PROLOG carry over this extension. Moreover, we claim that the notions of synchronization and communication will be better understood and expressed by precisely stating the meaning of clauses such as the one above.

First, the paper describes the syntax of both the left- and right-hand sides of clauses along with the language operational semantics; then it defines a first order semantics which is a straightforward generalization of the one given by vanEmden and Kowalski [5]. A fixpoint semantics is also given, and the three different semantics are shown to be equivalent. Finally, the paper shows how a concurrent program can be translated in a pure PROLOG program, generally composed by a denumerable set of clauses.

2. SYNTAX AND OPERATIONAL SEMANTICS

In this section we will give the syntax of our extension to PROLOG in two steps. First, we will introduce concrete syntax. It is an abbreviation for some constructs of the abstract syntax that will be defined later.

The concrete syntax of the language is the following.

A program is a set of clauses.

A clause is a sentence of the form

X <-- B1 + ... + Bm

where X is a class and each Bi is an atom.
The formula B1 + ... + Bm is the (possibly empty) body of the clause and X is its header.

A class either is an atom or has the form

(A&X)

where A is an atom and X is a class.
The notation (X&A) is completely equivalent to (A&X).

An atom has the form

$$A(t1,\ldots,tn)$$

where A is a predicate symbol and each ti is a term, i=1,...,n.

A term is built by variables and constructor applications to terms.

A goal is of the form

$$<\!-\!-\ B1 + \ldots + Bm \qquad m\geq 0.$$

The concrete syntax allows to abbreviate goals and bodies by using the connective +. Let us now define abstract syntax that gives to + a meaning in terms both of standard first order logic connectives, and of classes.

The formula

$$A1 + \ldots + An$$

is an abbreviation for

$$\begin{array}{l}(A1 \wedge \ldots \wedge An) \vee \\ (X11 \wedge \ldots \wedge X1k_1) \vee \\ \ldots\ldots \\ (Xp1 \wedge \ldots \wedge Xpk_p) \vee \\ (A1\& \ldots \& An)\end{array}$$

where:
- each Xij is a class built with atoms Ak;
- each Ak belongs exactly to one class Xij;
- p+2 is the number of all the possible conjunctions of distinct classes obtainable from A1, ... ,An. Actually,

$$p+2=\sum_{k=1}^{n} s(n,k)$$

  s(n,k) being the Stirling number of second kind that counts the number of partitions in k classes of n objects.

In the formulas above, we have intentionally omitted parenthesis, understanding that both & and + be right associative.

Example 1. The formula A + B + C abbreviates

$$(A \wedge B \wedge C) \vee (A\&B \wedge C) \vee (A\&C \wedge B) \vee (A \wedge B\&C) \vee (A\&B\&C)$$

The following distributive axioms hold that relate classical connectives and classes.

1. (A ∨ B)&C = (A&C) ∨ (B&C)
2. (A ∧ B)&C = ((A&C) ∧ B) ∨ (A ∧ (B&C))

A clause of the form

X <-- B1 + ... + Bm

is an abbreviation for one of the following

a) if m=0 X
b) if m>0 A1&(A2&(...&(Ak&X)...)) ∨
¬(A1&(A2&(...&(Ak& (B1 + ... + Bm))...)))

for each finite multiset of atoms {[A1,A2,...,Ak]} (compound brackets {[and]} enclose multiset elements).

The intuitive meaning of the clause

X <-- B1 + ... + Bm (*)

is that all the atoms occurring in class X must synchronize to be replaced with the body B1 + ... + Bm. Item (b) above can be better understood by considering that, if the atoms in class X occur as part of a larger class Y, they can still be replaced with B1 + ... + Bm that, in turn, will synchronize themselves with the remaining atoms of Y. On the contrary, if only some atoms of X are present in the goal, they cannot be replaced by clause (*). Hence, the symbol "&" occurring in a class does in no way be interpreted as a classical "∧", since the truth value of a class does not functionally depend on the truth values of the atoms it is composed with. We will come again on this issue in example 2 below.

A (concurrent) computation of a goal G is a sequence of goals G=G1,G2,..., where each G(i+1) is derived from Gi.

A (concurrent) refutation of G is a computation ending with the empty goal.

Given a goal G of the form

<-- G1 + ... + Gm

and a clause

A1&...&An <-- B1 + ... + Bk

we can derive a new goal g1

$$\leftarrow [B\ 1]_\lambda + \ldots + [B\ k]_\lambda + [G\sigma 1]_\lambda + \ldots + [G\sigma m]_\lambda$$

if and only if

- $n \leqslant m$

- σ is a permutation of the indexes of g and λ is a unifier such that
  $[G\sigma i]_\lambda = [Ai]_\lambda \quad i=1,\ldots,n.$

Example 2. Let us have the following ground clauses

1. A <-- D
2. A&B <-- E

and the goal

$$\leftarrow A + B + C \qquad (g)$$

The goal can be nondeterministically computed in the two following ways.

<-- A + B + C	<-- A + B + C
<-- D + B + C	<-- E + C
a)	b)

Let us examine what happens when abstract syntax is used in place of the concrete one. The goal is

$$\neg(A \wedge B \wedge C) \wedge \neg(A \wedge B \& C) \wedge \neg(A \& B \wedge C) \wedge \neg(A \& C\ B) \wedge \neg(A \& B \& C) \qquad (ag)$$

The clauses 1 and 2 will originate a denumerable set of clauses, but only the following can be applied to the goal.

1a. A $\vee \neg$D	2a. A&B $\vee \neg$E
1b. A&B $\vee \neg$D&B	2b. A&B&C $\vee \neg$E&C
1c. A&C $\vee \neg$D&C	
1d. A&B&C $\vee \neg$D&B&C	

For simplicity sake, let us consider only computation (b), which leads to

$$\neg(E \wedge C) \wedge \neg(E \& C)$$

which is expressed in concrete syntax exactly as

<-- E + C.

The result of computation (b) is a conjunction of the two clauses above since clause 2a and 2b may be applied to the third and the fifth conjuncts of the original goal, respectively. The other conjuncts can obviously be disregarded, since it is sufficient to refute a single conjunct to refute a whole conjunction.

Now we can better understand why the clause

A&B <-- E

corresponds to infinitly many clauses, each adding a finite class as "context" to A&B. The goal, when written in its abstract form (as), allows to better single out two conjuncts (the last two in (as)) which are worth to be noticed. In the first A and B are synchronized, in the other A and B are synchronized also with C. Hence, also the last conjunct (in which A&B&C occurs) must be replaced, resulting in E&C. The way + has beings defined assures that the synchronization between A&B and C is inherited by E.

Finally, remark that a clause in concrete syntax in general corresponds to infinite clauses in abstract syntax, but only a finite number of them will be actually used in a computation. The effectiveness of the definition of computation is then preserved.

Coming back to our example, notice that in computation (a) all the five conjuncts corresponding to the expansion of D + B + C will be obtained from (as). In fact, clause 1a applies to the first two conjuncts of (as), and 1b-d to exactly one of the remaining conjuncts.

3. MODEL-THEORETICAL AND FIXPOINT SEMANTICS

The construction of a Herbrand model for a set of clauses involving classes needs only to slightly change the one given by vanEmden and Kowalski [5]. The difference is related to the fact that the model of a class is not the intersection of the models of the atoms that occur in it. If so, "&" would be nothing more than the classical "∧", thus vanishing our proposal to describe a synchronization mechanism.

Par abus de langage, we will call Herbrand base for a program S the set of all multisets of ground atoms

$$\{[P_1^n(t_{11},\ldots,t_{1n}),\ldots,P_k^m(t_{k1},\ldots,t_{km})]\}$$

where:
- P_i are predicate symbols occurring in S,
- J is the rank of P_i^J,
- t_{rs} are ground terms.

A Herbrand interpretation is any subset of the Herbrand base.

Given a Herbrand interpretation I:

i) a ground class X is TRUE under I if and only if the multiset of its atoms belongs to I;

ii) a conjunction of ground clauses C1∧...∧Cm is TRUE under I if and only if all Ci's are TRUE under I;

iii) a disjunction of ground (both positive and negative) classes X1∨...∨Xm is TRUE under I if and only if at least one Xi is TRUE under I;

iv) the negation of a ground class ¬X is TRUE under I if and only if X does not belong to I;

v) a universally quantified clause C is TRUE under I if and only if all its ground instances are TRUE under I.

A Herbrand model of a program S is any interpretation under which all the clauses of S are TRUE.

The semantics of a program S is the minimal Herbrand model of S, which results to be the intersection of all the Herbrand models of S.

Note that the above definition of truth values of a formula under an interpretation is given in terms of abstract syntax only. Extending it to concrete syntax is an easy task. Let us simply give here the extension in the case of clauses.
A ground clause X <-- B1+...+Bm is TRUE under I if and only if for each finite multiset of ground atoms {[A1,...,Ak]}, k≥0, the disjunction

$$A1\&(A2\&(...\&(Ak\&X)...))\vee \neg(A1\&(A2\&(...\&(Ak\&\ (B1 + ... + Bm))...)))$$

is TRUE under I.

The definition of the fixpoint semantics for a program S in abstract syntax is quite standard.

The set of interpretations of a program S is partially ordered by standard set inclusion.

Given an interpretation I for a program S, the continuous transformation T associated to S yields a new interpretation I'. I' contains the multiset of ground atoms of a class X1 if

and only if there exists a ground instance of a clause of S

$$X1 \vee \neg X2 \vee \ldots \vee \neg Xn \qquad n>0$$

and the multiset of ground atoms of each Xi, i=2,...,n, belongs to I.

As usual, an interpretation I is closed under a transformation T if and only if I contains T(I).

The semantics of a program S is the intersection of all the closed interpretations of S, which can be easily proved to be the fixpoint of the above defined continuous transformation T.

The following theorem holds.

EQUIVALENCE THEOREM.
The operational, model-theoretic and fixpoint semantics are all equivalent.

The proof of the theorem relies on the following lemmas.

LEMMA 1.
The model theoretic semantics is equivalent to the fixpoint semantics.

This lemma is a corollary of the more general theorem stating that the set of the Herbrand models of a program S is equal to the set of all the interpretations closed under the continuous transformation T associated to S.

LEMMA 2.
The operational semantics is equivalent to the fixpoint semantics.

This lemma can easily be proved, since when there is a refutation of a program S and a ground class X, the multiset of ground atoms of X belongs to the fixpoint of the transformation T associated to S.

4. CONCURRENT PROGRAMS AND PROLOG PROGRAMS

We will now briefly discuss the relationships between a pure PROLOG program and a concurrent program in which classes occur. Actually, for each concurrent program there exists an equivalent PROLOG program which is denumerably infinite.

As defined above, a clause of the form

X <-- B1 + ... + Bm

corresponds to a denumerable set of clauses

A1&(A2&(...&(Ak&X)...)) ∨
¬(A1&(A2&(...&(Ak& (B1 + ... + Bm))...)))

each Ai being an atom.
Let us now translate a clause in which classes occur into a pure PROLOG clause, i.e. let us translate classes.

First, a total ordering relation > is imposed on the predicate symbols. Then, the class

$A1(t11,...,t1n_1)\&...\&Ak(tk1,...,tkn_K)$

where A(i+1)>Ai for all i=1,...,k-1, is translated into the single atom

$Q(t11,...,t1n_1,t21,...,tkn_k)$

where Q belongs to a denumerable infinite set of new predicate symbols. The translation function must be a bijection.

Note that the rank of Q is determined as the sum of the ranks of all the Ai's occurring in the class. For instance, the class

A1(x,y)&A2(x,z,w)

is translated into the following atom

Q(x,y,x,z,w).

Notice also that the condition on the ordering among atoms in a class is not a restriction, since the relative position of atoms in a class is both syntactically and semantically irrelevant.

The following fact is obviously true.

FACT. Given a translation from classes to atoms and two classes X and Y unifiable by λ, the translations of X and Y are still unifiable by λ.

We will now show that a concurrent computation of a goal is equivalent to a finite set of PROLOG computation. As mentioned above, the infinity of the translated program does not affect the effectiveness of the computations, because only a finite number of the clauses obtained by translation will actually be used in a computation.

Recall that a concurrent goal

<-- B1 + ... + Bm

corresponds to the following conjunction of PROLOG goals (let Bi be the translation of atom Bi, Qij the translation of the ij-th class, Q the translation of B1&...&Bm).

(<-- B1 ∧ ... ∧ Bm) ∧
(<-- Q11 ∧ ... ∧ Q1k_1) ∧
...
(<-- Qp1 ∧ ... ∧ Qpk_p) ∧
<-- Q

A step in a concurrent computation of a goal is then equivalent to a step of standard PROLOG computation on suitable selected goals coming from the translation. Of course, these must contain an instance of the header of the clause to be applied. Needless to say, a concurrent refutation corresponds to a set of PROLOG computations, one of which is a refutation.

The above remarks allow us to state the following theorem.

COMPLETENESS THEOREM
Any unsatisfiable (i.e. having no model) set consisting of a concurrent goal and a concurrent program has a refutation.

5. AN EXAMPLE

In order to illustrate the expressive power of our proposal, let us write a program that implements a "semaphore", through which a set of jobs can be synchronized. The program consists of four clauses defining the two classical primitives on semaphores p and v, and of two clauses implementing a queue.

```
1. p(sem_id,Job_id)&sem(sem_id,0,q)
            <-- enqueue(Job_id,q,q') + sem(sem_id,0,q')
2. p(sem_id,Job_id)&sem(sem_id,s(n),NIL)
            <-- sem(sem_id,n,NIL) + ack(Job_id)

3. v(sem_id,Job_id)&sem(sem_id,0,Job_id'.q)
            <-- sem(sem_id,0,q) + ack(Job_id) + ack(Job_id')
4. v(sem_id,Job_id)&sem(sem_id,n,NIL)
            <-- sem(sem_id,s(n),NIL) + ack(Job_id)

5. enqueue(Job_id,NIL,Job_id.NIL) <--
6. enqueue(Job_id,Job_id'.q,Job_id'.q')
            <-- enqueue(Job_id,q,q')
```

Natural numbers are represented by 0 and successor (s); queues by lists ending with NIL (the empty queue); semaphores by their name, a natural number variable and a queue. Semaphores are handled through p and v. A job job_id calling p on a semaphore sem_id is allowed to proceed running if the value of the semaphore (the second argument of sem_id) is not 0. Otherwise it is stopped and its identifier is enqueued. A job calling v either (re)starts a stopped job, if any, and dequeues its identifier, or increments the semaphore value. In both cases the calling job is resumed by sending it an acknowledgement (the definition and use of clauses ack is not shown here).

While clauses 5 and 6 are quite standard, clauses 1-4 are concurrent. Note that processes p (or v) and sem share the variable sem_id, and synchronize by communicating through it. This example shows that this kind of interaction, and also more complicated ways of synchronous communication, can be naturally and explicitly described by having more than one atom in a clause header. In fact, the specification of process sem, that manages the value and the queue of any semaphore, is isolated from those processes (p and v) that actually exploit the semaphore mechanism.

6. CONCLUSIONS

We have defined a first order semantics for an extension to PROLOG, based on a synchronization and communication primitive. The expressive power of the resulting language is stronger than the one of PROLOG. An intuitive argument to this claim can be found in the fact that a program involving such a feature corresponds to a denumerable infinite set of pure clauses. Furthermore, standard PROLOG programs can be structured as modules, and the possibly concurrent interactions among them can be naturally described in terms of the above primitive.

A similar solution to the problem of expressing concurrent programs in logic has been presented by Monteiro [8]. In his proposal, PROLOG is extended with the concept of event, thus leading to a temporal logic programming language.

Our future work will concern the possibility of introducing a sequential operator, following [7], and of giving it a precise logic meaning. Furthermore, we intend to enrich concurrent programs with the capability of processing infinite streams of data, as done in [1]. Finally, it is worth investigating on a concept of module that provide mechanisms to encapsulate logic programs.

REFERENCES

1. Bellia,M., Dameri,E., Degano,P., Levi,G., and Martelli,M. Applicative communicating processes in First Order Logic. Proc. 5th Int. Symposium on Programming, Torino 1982, Springer Verlag LNCS, 1-14.

2. Colmerauer,A., Kanoui,H., Pasero,R., and Roussel,P. Un sistéme de communication homme-machine en français. Groupe d'Intelligence Artificielle, Universite' d'Aix-Marseille, Luminy (1972).

3. Degano,P. Una classe di schemi ricorsivi non-deterministici paralleli. Calcolo, 14 (1977), 97-119.

4. Diomedi,S. Sulle basi teoriche di comunicazione e concorrenza nei linguaggi basati sulla logica. Tesi di laurea, Dip. di Informatica, Univ. di Pisa, 1983.

5. vanEmden,M.H., and Kowalski,R.A. The semantics of predicate logic as a programming language. J.ACM 23, 1976, 733-742.

6. Kowalski,R.A. Logic for Problem Solving. Artificial Intelligence Series, N.J. Nilsson ed., North-Holland, 1979.

7. Monteiro,L.F. A Horn-like logic for specifying concurrency. Proc. 1st Int. Logic Programming Conf., Marseille (1982), 1-8.

8 Monteiro,L.F. A proposal for distributed programming in logic. Unpublished manuscript.

9. Pereira,L.M. and Monteiro,L.F. The semantics of parallelism and co-routining in logic programming. COLLOQUIA MATHEMATICA SOCIETATIS JANOS BOLYAI, no. 26, North-Holland, Amsterdam (1981), 611-657.

This work has been partially supported by Ministero della Pubblica Istruzione.