Consistency of Clark's Completion and Existence of Stable Models

François Fages
LCR Thomson-CSF
and
LIENS, URA 1327 du CNRS
45 rue d'Ulm, 75005 Paris
fages@dmi.ens.fr

1 Abstract

The most general notion of canonical model for a logic program with negation is the one of stable model [9]. In [7] the stable models of a logic program are characterized by the *well-supported* Herbrand models of the program, and a new fixed point semantics that formalizes the bottom-up truth maintenance procedure of [4] is based on that characterization. Here we focus our attention on the abstract notion of well-supportedness in order to derive sufficient conditions for the existence of stable models. We show that if a logic program P is *positive-order-consistent* (i.e. there is no infinite decreasing chain w.r.t. the positive dependencies in the atom dependency graph of P) then the Herbrand models of $comp(P)$ coincide with the stable models of P. From this result and the ones of [10] [16] [2] on the consistency of Clark's completion, we obtain sufficient conditions for the existence of stable models for positive-order-consistent programs. Then we show that a negative cycle free program can have no stable model if it is not positive-order-consistent, while *order-consistency* alone [16] [2] (this condition generalizes call-consistency [10] [17], it is independent of positive-order-consistency) is sufficient to ensure the existence of a stable model. The last result is based on a characterization of stable models due to [5] and on a generalization of the result of [16] on the consistency of Clark's completion to infinite logic programs.

2 Introduction and Notations

A logic program P is a finite set of rules of the form $L_1, ..., L_n \rightarrow A$, where A is an atom, called the *conclusion* and denoted by $concl(R)$, and the L_i's are literals (i.e. atoms or negated atoms), called the *premises* and denoted by $prem(R)$. We denote the subset of positive atoms in $prem(R)$ by $pos(R)$, and the set of atoms under a negation by $neg(R)$. The set of ground instances of a logic program P is denoted by $Ground(P)$. Its Clark's completion [11] is

denoted by $comp(P)$.

The most general notion of canonical model for a logic program or its Clark's completion, is the one of stable model [9]. A Herbrand interpretation I of a logic program P is said to be a ***stable model*** of P if $I = M_{H(P,I)}$, where $M_{H(P,I)}$ denotes the least Herbrand model of the pure Horn program $H(P,I)$ defined by the stability transformation:

$$H(P,I) = \{pos(R) \rightarrow concl(R) \mid R \in Ground(P) \ \wedge \ neg(R) \cap I = \emptyset\}$$

This two step transformation eliminates the rules which have inconsistent negative premises w.r.t. I, and eliminates the negative premises in the remaining rules. A model is stable if it derives itself by this transformation. The stable models of P coincide with the default models of P in Reiter's default theory [15] [12]. Any stable model of a logic program P is a minimal Herbrand model of P and a model of $comp(P)$. Furthermore the stable model semantics coincides with various canonical model semantics defined for restricted classes of logic programs [9] [21].

The aim of this paper is to investigate sufficient conditions for the existence of stable models. The central notion is the dependency graph of a logic program on which are based recent results on the completeness of SLDNF-resolution [10] and on the consistency of Clark's completion [16] [2].

The *predicate dependency graph* [1] of a logic program P is a directed graph with signed edges. The nodes are predicate symbols occurring in P. There is a positive (resp. negative) edge from p to q if there is a rule in P with p in the premises and q in the conclusion.

We say q ***depends evenly (resp. oddly) on*** p, denoted $p \leq_+ q$ (resp. $p \leq_- q$), if there is a path in the predicate dependency graph from p to q with an even (resp. odd) number of negative edges. q ***depends on*** p, denoted $p \leq q$, if $p \leq_+ q$ or $p \leq_- q$. We say q ***depends positively on*** p, denoted $p \leq_0 q$, if there is a path from p to q with all edges positive (this definition differs from [2]).

A logic program P is *stratified* [1] [20] if no node depends on itself through at least one negative edge. P is *call-consistent* [10] [17] if no node depends oddly on itself, i.e. $\leq_-$ is irreflexive. For example the logic program $\{\neg a \rightarrow b, \ \neg b \rightarrow a\}$ is call-consistent but not stratified.

2.1. Theorem [9] [21]. If P is stratified then the stratified model of P (see [1]) is the unique stable model of P.

2.2. Theorem [17] [10] [2]. If P is call-consistent then $comp(P)$ has a Herbrand model.

The *atom dependency graph of* P [14], denoted by $G(P)$, is analogous to the predicate dependency graph. The nodes are the ground atoms of the Herbrand universe. There is a positive (resp. negative) edge from A to B if there is a rule $R \in Ground(P)$ with $A \in pos(R)$ (resp. $A \in neg(R)$) and $B \in concl(R)$. By abuse of notation we define also the relations $\leq$, $\leq_0$, $\leq_+$ and $\leq_-$, on ground atoms, in the same way as in the predicate dependency graph *.

A logic program P is said to be ***locally stratified*** [14] if the relation of dependency through at least one negative edge in $G(P)$ is well-founded. P is said to be *negative cycle free* [16] if $\leq_-$ is irreflexive in $G(P)$. P is said to be *order-consistent* [16] if the relation ($\leq_+$ and $\leq_-$) in $G(P)$ is well-founded (this condition is called local call-consistency in [2]). We say P is *positive-order-consistent* if $\leq_0$ is well-founded.

For example the logic program $\{p(s(X)) \rightarrow p(X), \ \neg p(s(X)) \rightarrow p(X)\}$ is negative cycle free, but not order-consistent, nor positive-order-consistent because of the first rule, nor locally stratified because of the second rule. Note that on one hand order-consistency and positive-order-consistency are independent conditions, and on the other hand both classes of call-consistent programs and of locally stratified programs are contained in the class of order-consistent programs, itself contained in the class of negative cycle free programs.

2.3. Theorem [9] [21]. If P is locally stratified then the unique perfect model of P (see [14]) is the unique stable model of P.

2.4. Theorem [16] [2]. If P is order-consistent then $comp(P)$ has a Herbrand model.

---

* Consistently with [16] we shall say that a binary relation (not necessarily a partial order) $\leq$ is well-founded if there is no infinite decreasing chain $x_0 \geq x_1 \geq ...$, in particular $\leq$ must be acyclic to be well-founded.

2.5. Theorem [16]. If P is negative cycle free and either function free or internal variable free (i.e. for any rule the variables in the premise appear in the conclusion) then $comp(P)$ has a Herbrand model.

The assumptions in theorems 2.4 and 2.5 are independent. It is an open problem whether negative cycle freeness alone implies the consistency of $comp(P)$ or not. In the sequel we study sufficient conditions for the existence of stable models that subsume (local) stratification.

3 Existence of Stable Models, Part 1

The stable models of a logic program can be characterized in terms of well-supported interpretations. We say a Herbrand interpretation I is *well-supported* iff there exists a strict well-founded partial order $\prec$ on I such that for any atom $A \in I$ there exists a rule $R \in Ground(P)$ with $concl(R) = A$, $I \models prem(R)$ and for any $B \in pos(R)$, $B \prec A$.

3.1. Theorem [7]. For a general logic program P, the well-supported models of P are exactly the stable models of P.

This was shown independently by [6] in the case where P is a propositional program. For example the program $P_2 = \{p \rightarrow p,\ \neg p \rightarrow q\}$ has two supported minimal models, $\{p\}$ and $\{q\}$ which are both models of $comp(P_2)$, but only one well-supported model $\{q\}$ which is also the unique stable model and the iterated least model in the stratified semantics of [1] [20]. The condition of well-supportedness eliminates cyclic and infinite supports. Well-supported models are finitely justified models. For instance the program $P_3 = \{p(s(x)) \rightarrow p(x)\}$, given with a constant a, has two supported Herbrand models, $\emptyset$ and $\{p(s^i(a)) | i \geq 0\}$. They are both models of $comp(P_3)$, but $\emptyset$ is the only finitely justified model of P_3, i.e. the unique stable model of P_3.

3.2. Theorem. If P is positive-order-consistent then the Herbrand models of $comp(P)$ coincide with the stable models of P.

Proof. A stable model of P is a Herbrand well-supported model of P, so it is a Herbrand minimal supported model of P, hence a model of $comp(P)$ [11]. Conversely let us reason by contradiction. Suppose M is a Herbrand model of $comp(P)$ but not a well-supported model of P. There exists an atom $A \in M$ that cannot be finitely justified. However as M is a supported model of P, there exists a rule $R \in Ground(P)$ with $concl(R) = A$ and $M \models prem(R)$. As

A cannot be finitely justified there exists $B \in pos(R)$, so $B \leq_0 A$, such that B cannot be finitely justified. Therefore we get an infinite decreasing chain w.r.t. $\leq_0$, i.e. a contradiction with the positive-order-consistency of P. □

3.3. Corollary. If P is positive-order-consistent, negative cycle free and either function free or internal variable free, then P has a stable model.

Proof. By 3.2 and 2.5. □

3.4. Corollary. If P is positive-order-consistent and order-consistent then P has a stable model.

Proof. By 3.2 and 2.4. □

3.5. Corollary. If P is call-consistent and the relation $\leq_0$ on predicate symbols is acyclic (these assumptions are clearly decidable) then P has a stable model.

Proof. By 3.4, as call-consistency implies order-consistency and $\leq_0$ acyclic on predicate symbols implies positive-order-consistency. □

These corollaries can be exploited by logic programming tools based on the stable model semantics, as in Truth Maintenance Systems [7] [6] or in Deductive Data Bases [19] [22], to detect semantic errors statically. They are also interesting from a programming point of view. Any logic program can be transformed into a positive-order-consistent program that preserves the program's completion semantics. The transformation consists in replacing a positive premise in a rule

$$L_1, ..., p(t_1, ..., t_k), ..., L_n \rightarrow A$$

by a negative premise

$$L_1, ..., \neg\overline{p}(t_1, ..., t_k), ..., L_n \rightarrow A$$

adding the rule

$$\neg p(x_1, ..., x_k) \rightarrow \overline{p}(x_1, ..., x_k)$$

It is obvious that this double negative does not change the program's completion semantics, but it makes $\leq_0$ acyclic on predicate symbols. That transformation appears in [18] to show that the program's completion semantics and the well-founded semantics have the same theoretical expressive power. From a logic

programming point of view under the stable model semantics that transformation is a way to relax the constraint of well-supportedness for some predicates, by requiring only that the intended interpretation of these predicates be supported, as in the models of the program's completion.

It is an open problem to know whether negative cycle free programs in the general case have a consistent completion. However we show with an example that this condition does not ensure the existence of a stable model.

3.6. Theorem. There exist (internal variable free) negative cycle free programs that have no stable model.

Proof. Let $P = \{p(s(X)) \rightarrow p(X), \quad \neg p(s(X)) \rightarrow p(X)\}$. This is an internal variable free negative cycle free program, so $comp(P)$ is consistent by 2.5. The only Herbrand model of $comp(P)$ takes p true everywhere, however this is not a well-supported model as it is not finitely justified. Therefore P has no stable model. □

So positive-order-consistency cannot be eliminated in the assumption of corollary 3.3. However positive-order-consistency (resp. $\leq_0$ acyclic) is not necessary in the assumption of corollary 3.4 (resp. 3.5). To show this we study first the Clark's completion of infinite logic programs.

4 Consistency of Completed Infinite Logic Programs

An *infinite logic program* P is an infinite set of (finite) rules

$$L_1, ..., L_n \rightarrow A$$

formed on a countable set of variables and a finite alphabet of constants, functions and predicates. In this way $Ground(P)$ is always countable. The atom dependency graph is defined as for finite programs. In this section we show that theorem 2.4 holds for infinite logic programs as well, with the same proof as in [16] [2] in the line of [10] [17].

The Clark's completion of an infinite logic program is naturally defined by an infinite formula of first-order classical logic. The *completed definition* of a predicate p, supposed to be unary here, is an infinite first-order formula of the form

$$\forall x\ p(x) \leftrightarrow \exists y_1 ... \exists y_j ...\ (x = t_1 \wedge C_1) \vee ... \vee\ (x = t_i \wedge C_i) \vee ...$$

where the disjunction can be infinite, but each member $(x = t_i \wedge C_i)$ is a finite conjunction corresponding to a rule $R \in P$ with $concl(R) = p(t_i)$ and $prem(R) = C_i$. The *Clark's completion* of an infinite logic program is composed of the (finite) conjunction of the completed definition of each predicate symbol, together with the strong equality axioms.

The immediate consequence operator is defined undifferently on infinite logic programs,

$$T_P(I) = \{concl(R) \mid R \in Ground(P),\ I \models prem(R)\}$$

Its fixed points are the Herbrand models of $comp(P)$.

4.1. Proposition. Let P be an infinite logic program. Then a Herbrand interpretation I is a model of $comp(P)$ if and only if I is a fixed point of T_P.

Proof. One can easily check that the proof in [11] for definite programs remains correct in presence of both negation [1] and infinite completed definitions. □

Now, pair mappings on the set of *partial interpretations* in $2^{B_H} \times 2^{B_H}$, first introduced in [8], can be associated to infinite programs without any modification. Let P be an infinite logic program and $(M, N) \in 2^{B_H} \times 2^{B_H}$ be a partial interpretation, let us define

$$T_P^+(M,N) = \{concl(R) \mid R \in Ground(P),\ pos(R) \subseteq M,\ neg(R) \subseteq N\}$$

$$T_P^-(M,N) = \{A \mid \forall R \in Ground(P)$$
$$concl(R) = A \ \Rightarrow\ pos(R) \cap N \neq \emptyset \vee neg(R) \cap M \neq \emptyset\}$$

The pair mapping $< T_P^+, T_P^- >$ on infinite logic programs enjoys the same property of monotonicity and gives Herbrand models of $comp(P)$ under certain conditions.

4.2. Proposition. If $M \cap N = \emptyset$ then $T_P^+(M,N) \cap T_P^-(M,N) = \emptyset$.

4.3. Proposition. $< T_P^+, T_P^- >$ is monotonic in the lattice $2^{B_H} \times 2^{B_H}$ ordered by pair inclusion $\sqsubseteq$, that is $(M_1, N_1) \sqsubseteq (M_2, N_2)$ (i.e. $M_1 \subseteq M_2$ and $N_1 \subseteq N_2$) implies $< T_P^+, T_P^- > (M1, N1) \sqsubseteq < T_P^+, T_P^- > (M2, N2)$.

4.4. Proposition. If $M \cap N = \emptyset$ and $(M,N) \sqsubseteq < T_P^+, T_P^- > (M,N)$ then there exists a fixed point (M',N') of $< T_P^+, T_P^- >$ such that $(M,N) \sqsubseteq (M',N')$ and $M' \cap N' = \emptyset$.

4.5. Proposition. If (M,N) is a fixed point of $< T_P^+, T_P^- >$, $M \cap N = \emptyset$ and $M \cup N = B_H$ then M is a Herbrand model of $comp(P)$.

Proof. As $M \cap N = \emptyset$ and $M \cup N = B_H$ we have $T_P^+(M,N) = T_P(M)$ by definition. As (M,N) is a fixed point of $< T_P^+, T_P^- >$ we have $T_P^+(M,N) = M$. Therefore M is a fixed point of T_P, i.e. a model of $comp(P)$ by 4.1. □

At this point it is clear that the useful properties of pair mappings and order-consistency exploited in [16] on finite logic programs are met by infinite logic programs. Therefore one can check that through the same series of lemmas with the same proofs as in [16] we get:

4.6. Theorem. If an infinite logic program P is order-consistent then $comp(P)$ has a Herbrand model.

On the other hand, note that the proof of existence of a Herbrand model for completed negative cycle free programs in [16] does make use of the compactness theorem of first-order classical logic, hence it cannot be lifted to infinite logic programs. In fact there exist infinite negative cycle free programs, like

$$\{\neg P(s^i(x)) \rightarrow P(x) \mid i > 0\},$$

that have an inconsistent Clark's completion.

5 Existence of Stable Models, Part 2

This section makes use of the characterization due to [5] of the stable models of a logic program P by the Herbrand models of its *fixpoint completion*. We recall here the basic definitions with our notations.

A *quasi-interpretation* Q is a possibly infinite set of ground rules without positive premise, i.e. of the form

$$\neg A_1, ..., \neg A_n \rightarrow A$$

where $n \geq 0$, and A and the A_i's are ground atoms formed over a finite alphabet.

Given a (finite) logic program P, the *immediate consequence operator* T_P is defined on quasi-interpretations by

$$T_P(Q) = \{neg(R_1), ..., neg(R_n), neg(R) \rightarrow A \mid R \in Ground(P)$$
$$concl(R) = A,\ pos(R) = \{A_1, ..., A_n\},\ n \geq 1,$$
$$\forall i,\ 1 \leq i \leq n,\ R_i \in Q,\ concl(R_i) = A_i\}$$

T_P is a continuous operator in the lattice of quasi-interpretations [5]. The least fixed point of T_P is denoted by $fix(P)$, i.e.

$$fix(P) = \bigcup_{i \geq 0} T_P \uparrow i$$

In general $fix(P)$ is an infinite quasi-interpretation, its Clark's completion is then defined as in the previous section. We shall use the equivalence theorem of [5] that relates the stable models of P with the Herbrand models of $comp(fix(P))$.

5.1. Theorem [5]. The Herbrand models of $comp(fix(P))$ are exactly the stable models of P.

5.2. Lemma. Let P and P' be two (infinite) logic programs based on the same alphabet. The relation $\leq_+$ (resp. $\leq_-$) in $G(P)$ is included in $\leq_+$ (resp. $\leq_-$) in $G(P')$ if and only if for each rule $R \in Ground(P)$ with $concl(R) = B$, for every $A \in pos(R)$ (resp. $A \in neg(R)$), we have $A \leq_+ B$ (resp. $A \leq_- B$) in $G(P')$.

Proof. If $\leq_+$ (resp. $\leq_-$) in $G(P)$ is included in $\leq_+$ (resp. $\leq_-$) in $G(P')$ then for any rule $R \in Ground(P)$ with $concl(R) = B$, and for every $A \in pos(R)$ (resp. $A \in neg(R)$) we have $A \leq_+ B$ (resp. $A \leq_- B$) in $G(P)$. Thus by hypothesis we get $A \leq_+ B$ (resp. $A \leq_- B$) in $G(P')$.

Conversely if $A \leq B$ in $G(P)$ then there is a path from A to B in $G(P)$. For each positive (resp. negative) edge (A_i, A_{i+1}) in that path there exists a rule $R \in Ground(P)$ with $concl(R) = A_{i+1}$ and $A_i \in pos(R)$ (resp. $A_i \in neg(R)$). So by hypothesis we have $A_i \leq_+ A_{i+1}$ (resp. $A_i \leq_- A_{i+1}$) in $G(P')$. Therefore if $A \leq_+ B$ (resp. $A \leq_- B$) in $G(P)$, there is an even (resp. odd) number of negative edges in the path, thus we get $A \leq_+ B$ (resp. $A \leq_- B$) in $G(P')$. □

5.3. Lemma. If $A \leq_- B$ in $G(fix(P))$ then $A \leq_- B$ in $G(P)$.

Proof. By 5.2 it suffices to show that for any rule $R \in fix(P)$, i.e. for any $i \geq 0$ and any $R \in T_P \uparrow i$, we have $A \leq_- B$ in $G(P)$ if $concl(R) = B$ and $A \in neg(R)$ (the case $A \in pos(R)$ does not arise as $fix(P)$ is a quasi-interpretation). The proof is by induction on i.

The base case is trivial. Let $R \in T_P \uparrow i+1$, let $A \in neg(R)$ and $concl(R) = B$. By definition of $T_P \uparrow i+1$ there exists a rule $S \in Ground(P)$ with $concl(S) = B$ and either $A \in neg(S)$ or $A \in neg(R')$ with $R' \in T_P \uparrow i$ and $concl(R') \in pos(S)$. In the former case we get immediately $A \leq_- B$ in $G(P)$. In the latter case we get $A \leq_- concl(R')$ in $G(P)$ by induction. As $concl(R') \leq_+ B$ in $G(P)$ we conclude again $A \leq_- B$ in $G(P)$. □

5.4. Theorem. An order-consistent logic program has a stable model.

Proof. If a logic program P is order-consistent then as $fix(P)$ is a quasi-interpretation, $\leq_0$ is the identity relation, so we get by lemma 5.3 that $fix(P)$ is also order-consistent. Therefore by theorem 4.6 $comp(fix(P))$ has a Herbrand model, hence P has a stable model by theorem 5.1 . □

6 Conclusion

We have shown that the main syntactic criteria that ensure the consistency of Clark's completion of logic programs [10] [17] [16] [2], ensure also the existence of stable models for these programs, with the noticeable exception of negative cycle free programs that can have no stable model. The most general conditions (order-consistency on one hand, negative cycle freeness and positive-order-consistency on the other hand) are probably undecidable, but they have a checkable counterpart, namely call-consistency. This can be directly exploited by logic programming tools based on the stable model semantics, as in Truth Maintenance Systems [7] [6] or in Deductive Data Bases [19] [22], to detect semantic errors statically.

One remaining problem is to find a direct proof of the existence of stable models for order-consistent logic programs. For example every stable model of a logic program can be obtained as an ordinal power of the justification maintenance operator associated to the program under a suitable strategy [7]. One would like to exhibit for order-consistent programs a class of strategies under

which the ordinal powers of the justification maintenance operator are increasing, whence reach a stable model. In [7] it is shown that any fair strategy constructs the 2-valued well-founded model of the program if it exists. Here the proof of existence of a stable model under the order-consistency assumption suggests another transfinite process: first construct the fixpoint completion by iterating T_P up to ordinal ω, then choose a signing, that gives a stable model, for example by iterating the justification maintenance operator under any conservative strategy (i.e. any strategy that selects in priority the rules which do not have for effect to retract an atom). A fundamental problem underlying these difficulties is the logical complexity of stable models for restricted classes of programs [18].

The existence of stable models for order-consistent (or call-consistent) logic programs can be transcripted in various theories of non-monotonic reasoning developed in AI. In Truth Maintenance System terminology, we have shown that if a set of justifications contains no odd loop in its dependency network then there exists a state of beliefs with well-founded supporting justifications, or equivalently (see [7]) there exists a strategy under which the Truth Maintenance procedure converges. In Reiter's default logic we get that an order-consistent default theory is consistent. In Moore 's autoepistemic logic, we have shown the existence of a stable autoepistemic expansion for any call-consistent set of premises.

Finally we conjecture that although negative cycle free programs can have no stable model, their Clark's completion is always consistent. In case of a positive answer to the conjecture, negative cycle freeness would subsume the other conditions that ensure the consistency of the Clark's completion, but with no guarantee on the existence of canonical model for the program.

References

[1] K.R. Apt, H.A. Blair, A. Walker, *Towards a theory of declarative knowledge*, in Foundations of deductive databases and logic programming, Minker, J. (ed.), Morgan Kaufmann, Los Altos (1987).

[2] A. Cortesi, G. Filé, *Graph properties for normal logic programs*, Proc. 5th Conv. Nat. sulla Programmazione Logica GULP'90, Padova, (1990).

[3] L. Cavedon, J.W. Lloyd, *A completeness theorem for SLDNF-resolution*, Journal of Logic Programming, 7(3), pp.177-192 (1989).

[4] J. Doyle, *A truth maintenance system*, Artificial Intelligence, vol. 12, pp.231-272 (1979).

[5] P.M. Dung, K. Kanchanasut, *A fixpoint approach to declarative semantics of logic programs*, NACLP'89. MIT Press (1989).

[6] C. Elkan, *A rational reconstruction of nonmonotonic truth maintenance systems*, Journal of Artificial Intelligence, vol. 43, pp.219-234 (1990).

[7] F. Fages, *A new fixpoint semantics for general logic programs compared with the well-founded and the stable model semantics*, 7th International Conference on Logic Programming, Jerusalem. MIT Press (June 1990). To appear in the Journal of New Generation Computing.

[8] M.R. Fitting, *A Kripke-Kleene semantics for logic programs*, J. of Logic Programming 2, pp.295-312 (1985).

[9] M. Gelfond, V. Lifschitz, *The stable model semantics for logic programming*, Proc. of the 5th Logic Programming Symposium, pp.1070-1080, MIT press (1988).

[10] K. Kunen, *Signed data dependencies in logic programs*, Journal of Logic Programming, 7(3), pp.231-245 (1989).

[11] J.W. Lloyd, *Foundations of Logic Programming*, Springer Verlag (1987).

[12] W. Marek, M. Truszcinski, *Stable semantics for logic programs and default theories*, NACLP'89. MIT Press (1989).

[13] T. Przymusinski, *On the declarative and procedural semantics of logic programs*, Journal of Automated Reasoning, 5, pp.167-205 (1989).

[14] T. Przymusinski, *On the declarative semantics of stratified deductive data-bases and logic programming*, in Foundations of deductive databases and logic programming, Minker, J. (ed.), Morgan Kaufmann, Los Altos (1987).

[15] R. Reiter, *A logic for default reasoning*, Artificial Intelligence, 13, pp.81-132 (1980).

[16] T. Sato, *Completed logic programs and their consistency*, Journal of Logic Programming, vol.9 (1), pp.33-44 (1990).

[17] T. Sato, *On the consistency of first-order logic programs*, ETL technical report TR-87-12, (1987).

[18] J.S. Schlipf, *The expressive powers of the logic programming semantics*, PODS'90, pp.196-204, (1990).

[19] D. Sacca, C. Zaniolo, *Stable models and non-determinism in logic programs with negation*, PODS'90, pp.205-217, (1990).

[20] A. Van Gelder, *Negation as failure using tight derivations for general logic programs*, in Foundations of deductive databases and logic programming, Minker, J. (ed.), Morgan Kaufmann, Los Altos (1987). Also in J. of Logic Programming 1989 pp.109-133.

[21] A. Van Gelder, K. Ross, J.S. Schlipf, *Unfounded sets and well-founded semantics for general logic programs*, Proc. of the Symp. on Principles of Databases Systems, ACM-SIGACT-SIGCOM (1988).

[22] D.S. Warren, *The XWAM: a machine that integrates Prolog and deductive database query evaluation*, Technical Report 89/25, Suny at Stony Brook, NY (1989).

A Well-founded semantics for Ordered Logic Programming‡

N. Leone and G. Rossi

CRAI - Loc. S. Stefano, 87036 Rende (Italy)

ABSTRACT. *This paper describes an extension of traditional logic programming, called* ordered logic *(OL) programming, and formerly presented in [LSV90], to support classical negation as well as constructs from the object-oriented paradigm. Such an extension allows to cope with the notions of object, inheritance and non-monotonic reasoning. After an informal description of the language, the main contribution of the present work amounts to the definition of a well-founded semantics for OL programs. This work has been carried out in the framework of the ESPRIT project KIWIS.*

1. INTRODUCTION

In the past years, most of the efforts of the logic programming community were devoted to improve the expressiveness of the logic languages. The main thrust in the research went in two principal directions: (i) integration with object oriented constructs, and, more recently, (ii) treatment of classical negation.

The contamination of the logic approach with object oriented features is actually a consolidated trend in the field of knowledge representation. Indeed, this is not at all surprising considered both the great interest in such fields as well as their complementarity. This has produced a plethora of new languages [Ait86, GLR89, CW89, KLW90, KL89, Con87, NT88] as extension of logic. They typically include the notion of object identity, (multiple) inheritance, object sharing, default values, as well as amenities for set terms and complex objects.

Very recently, a considerable amount of work has been devoted to exploit the treatment of classical negation in logic programming. The starting point was to recognize, as a basic limitation to the expressiveness of logic programming, the fact that negative information could not

‡ Research partially supported by *EEC* in the framework of ESPRIT II project EP2424 "KIWIS".

[6] C. Elkan, *A rational reconstruction of nonmonotonic truth maintenance systems*, Journal of Artificial Intelligence, vol. 43, pp.219-234 (1990).

[7] F. Fages, *A new fixpoint semantics for general logic programs compared with the well-founded and the stable model semantics*, 7th International Conference on Logic Programming, Jerusalem. MIT Press (June 1990). To appear in the Journal of New Generation Computing.

[8] M.R. Fitting, *A Kripke-Kleene semantics for logic programs*, J. of Logic Programming 2, pp.295-312 (1985).

[9] M. Gelfond, V. Lifschitz, *The stable model semantics for logic programming*, Proc. of the 5th Logic Programming Symposium, pp.1070-1080, MIT press (1988).

[10] K. Kunen, *Signed data dependencies in logic programs*, Journal of Logic Programming, 7(3), pp.231-245 (1989).

[11] J.W. Lloyd, *Foundations of Logic Programming*, Springer Verlag (1987).

[12] W. Marek, M. Truszcinski, *Stable semantics for logic programs and default theories*, NACLP'89. MIT Press (1989).

[13] T. Przymusinski, *On the declarative and procedural semantics of logic programs*, Journal of Automated Reasoning, 5, pp.167-205 (1989).

[14] T. Przymusinski, *On the declarative semantics of stratified deductive data-bases and logic programming*, in Foundations of deductive databases and logic programming, Minker, J. (ed.), Morgan Kaufmann, Los Altos (1987).

[15] R. Reiter, *A logic for default reasoning*, Artificial Intelligence, 13, pp.81-132 (1980).

[16] T. Sato, *Completed logic programs and their consistency*, Journal of Logic Programming, vol.9 (1), pp.33-44 (1990).

[17] T. Sato, *On the consistency of first-order logic programs*, ETL technical report TR-87-12, (1987).

[18] J.S. Schlipf, *The expressive powers of the logic programming semantics*, PODS'90, pp.196-204, (1990).

[19] D. Sacca, C. Zaniolo, *Stable models and non-determinism in logic programs with negation*, PODS'90, pp.205-217, (1990).

[20] A. Van Gelder, *Negation as failure using tight derivations for general logic programs*, in Foundations of deductive databases and logic programming, Minker, J. (ed.), Morgan Kaufmann, Los Altos (1987). Also in J. of Logic Programming 1989 pp.109-133.

[21] A. Van Gelder, K. Ross, J.S. Schlipf, *Unfounded sets and well-founded semantics for general logic programs*, Proc. of the Symp. on Principles of Databases Systems, ACM-SIGACT-SIGCOM (1988).

[22] D.S. Warren, *The XWAM: a machine that integrates Prolog and deductive database query evaluation*, Technical Report 89/25, Suny at Stony Brook, NY (1989).

A Well-founded semantics for Ordered Logic Programming‡

N. Leone and G. Rossi

CRAI - Loc. S. Stefano, 87036 Rende (Italy)

ABSTRACT. *This paper describes an extension of traditional logic programming, called* ordered logic *(OL) programming, and formerly presented in [LSV90], to support classical negation as well as constructs from the object-oriented paradigm. Such an extension allows to cope with the notions of object, inheritance and non-monotonic reasoning. After an informal description of the language, the main contribution of the present work amounts to the definition of a well-founded semantics for OL programs. This work has been carried out in the framework of the ESPRIT project KIWIS.*

1. INTRODUCTION

In the past years, most of the efforts of the logic programming community were devoted to improve the expressiveness of the logic languages. The main thrust in the research went in two principal directions: (i) integration with object oriented constructs, and, more recently, (ii) treatment of classical negation.

The contamination of the logic approach with object oriented features is actually a consolidated trend in the field of knowledge representation. Indeed, this is not at all surprising considered both the great interest in such fields as well as their complementarity. This has produced a plethora of new languages [Ait86, GLR89, CW89, KLW90, KL89, Con87, NT88] as extension of logic. They typically include the notion of object identity, (multiple) inheritance, object sharing, default values, as well as amenities for set terms and complex objects.

Very recently, a considerable amount of work has been devoted to exploit the treatment of classical negation in logic programming. The starting point was to recognize, as a basic limitation to the expressiveness of logic programming, the fact that negative information could not

‡ Research partially supported by *EEC* in the framework of ESPRIT II project EP2424 "KIWIS".

be directly derived, i.e. it could be expressed only through closed-world reasoning. All known proposals yield to logic programs where negation may appear also in the head of the rules (*negative programs*). Negative programs have been introduced in [BS89] and in subsequent proposals by [KS89, GL88] where both negation as failure and classical negation are retained in the language, and lately by [GS90] where negation is uniformly treated in the classical sense, neglecting negation as failure.

Both trends have been exploited within the *ordered logic* (OL) programming proposal [LSV90] which also constitutes the essential reference for this paper.

As a matter of fact, OL programming is centered around a fruitful combination between negative programs and object oriented data abstraction mechanisms, including multiple inheritance, exceptions, default values whereas preserving the full declarativity characteristic of logic languages.

Very briefly, an ordered program consists of a number of *components* (or equivalently *objects*), each one composed of a set of rules possibly with negative head literal. Rule inheritance is obtained by introducing a sort of "isa" hierarchy among objects, and it allows to deal with default properties and exceptions by specializing rules. Therefore, each object has its own perception of the knowledge base, given by its local rules plus some *global* rules, that is, those of the objects connected to it via the "isa" links. Contradictions between different 'perspectives' is handled by *overruling*, i.e. local rules hide global rules, and *defeating*, i.e. contradicting information inherited by two different components is rejected.

As a final remark, it is interesting to note that traditional logic programming is subsumed by ordered logic programming. In fact, a traditional logic program (with negation only within rule bodies) can be equivalently represented by a two level OL program, where all original rules go into the lower component whereas the upper component contains all possible negative literals, thus mimicking the closed world assumption [LSV90].

The main contribution of the present work is to define a rich well-founded semantics for ordered logic programs, extending the work presented in [LSV90, Lae90, LV90]

An overview of the language is presented in section 2. Thereafter, the subsequent section provides a well-founded semantics and a final paragraph shortly outlines the on going work and the relationships with the ESPRIT project KIWIS.

2. THE LANGUAGE

The section is devoted to give an overall presentation of ordered logic programming. The notion of ordered logic program is first supplied, then a few suitable examples show how multiple inheritance with exception is embedded into traditional logic programming.

A *negative program* is a set of rules of the form $A :- A_1, \ldots, A_n$ where $A, A_1, \ldots, A_n$ are literals (we point out that the head of a rule may be a negative literal)

An *object* is a pair $<o, C_o>$, where o is a unique *object identifier* and $C(o)$, the *component* (or *definition*) of o, is a negative program.

An *ordered logic program* P is a partially ordered set of objects $< O, \leq >$. Given two objects o_1 and o_2 we say that o_1 is an *instance* of o_2 if $o_1 \leq o_2$.

The following example discusses the basic principles of inheritance within this framework.

Example 1. Let us consider the following program

$< top,$ $\{thisyear(1990).\} >$
$< person,$ $\{age(X) :- birthyear(Y), thisyear(Z), X \text{ is } Z-Y.\} >$
$< john,$ $\{ name("John").$
$birthyear(1961).\} >$

where *top*, *john* and *person* are object identifiers and $john \leq person \leq top$ hold. Note that the only information contained in the definition of the object *john* do not allow to infer the property *age* for it. □

However, in the above example it is intuitively clear that *john* has the property *age*(29) and if we do allow rules to filter down from an object to its instances, then *john* inherits from *person* the rule for *age* and from *top* the fact *thisyear*(1990). Thus, by extending the definition of *john* with information "inherited" by *john*, we can correctly derive *age*(29) on the object *john*. In this way, we have introduced the notion of *inheritance* as an information flow from an object to its instances.

As already said, exceptions in the inheritance process are supported by the possibility of using negation in the head of the rules. In fact, the truth of a negative fact can be stated only if there exist a rule that derives it (true negation), as opposite to the negation as failure. Clearly, the presence of rules with negative head may lead to state contradicting facts about an object, as shown in the example below.

Example 2. Consider the simple program:

$< bird,$ $\{ fly. \} >$
$< penguin, \{ \neg fly. \} >$
$< tweety,$ $\{\} >$

and assume that $tweety \leq penguin \leq bird$. It is clear that, if both information, fly and $\neg fly$, were filtered down to *tweety* they would produce a contradiction. Therefore, we could not deduce whether the property fly, for *tweety*, holds or not. □

This drawback can be circumvented by introducing two mechanisms, namely *overruling* and *defeating*, that allow to deal with exceptions in the inheritance process.

The overruling process solves the contradiction according to the principle that the properties associated to an object are more specific, and then more reliables, of those inherited from another object. Hence, a piece of information at an object o_1 gets *overruled* at the object o_2, with $o_2 \leq o_1$, if it introduces some contradiction at o_2. As an example of overruling, consider the program of example 2. It is clear that, at the object *penguin*, the property $\neg fly$ holds whereas fly is overruled.

On the contrary, the defeating process rejects the whole contradiction. This is the case when the contradiction at o_1 is produced from information coming from two objects, say o_2 and o_3, s.t. $o_1 \leq o_2$ and $o_1 \leq o_3$ (multiple inheritance). In order to deal with such situations a

new truth value is needed. In particular, a literal A is said to be *undefined* if there is no way to state neither A nor $\neg A$.

Example 3. As an example of defeating, consider the following knowledge base

$< student, \quad \{ poor. \} >$
$< employee, \quad \{ \neg poor. \} >$
$< john, \quad \{ free\_ticket :- poor. \} >$

and assume that $john \leq student$ and $john \leq employee$. It is clear that *john* cannot inherit the contradicting information about *poor* from *student* and *employee*. In this case, since there is no criterion to estabilish which one of the two pieces of information is more reliable, we defeat both of them. Now, neither *poor* nor $\neg poor$ can be stated about *john*: *poor* is undefined. □

3. THE WELL-FOUNDED SEMANTICS

In this section we extend the well-founded semantics, first presented for traditional logic programs [GRS88, Prz89] to the class of ordered logic programs. Let us now start by supplying some useful notation and definition.

Given a program P, and an object o, let $\mathbb{P}_o = \bigcup_{o \leq o_i} C_{o_i}$ ($\mathbb{P}_o$ is the negative program containing all rules holding on o). The *Herbrand Universe* $H_{\mathbb{P}_o}$ of $\mathbb{P}_o$ is the set of all constant appearing in $\mathbb{P}_o$. A *ground instance* $\overline{r}$ of a rule r is the rule obtained by replacing each variable in r by a constant. We denote by $ground(\mathbb{P}_o)$ the set of all possible ground rules obtained from the rules in $\mathbb{P}_o$ and constants in $H_{\mathbb{P}_o}$. The *Herbrand Base* $B_{\mathbb{P}_o}$ of $\mathbb{P}_o$ is the set of all (negative and positive) ground literals appearing in $ground(\mathbb{P}_o)$. An interpretation for $\mathbb{P}_o$ is a subset of $B_{\mathbb{P}_o}$ which does not contain any pair of complementary[1] literals. A literal Q is *true* with respect to a given interpretation I if $Q \in I$, Q is *false* if its complementary literal is true w.r.t. I, Q is *undefined* if it is neither true nor false w.r.t. I.

Let $H(r)$ and $B(r)$ denote the head literal and the set of body literals of a rule r, respectively. Given a ground literal Q and an interpretation I, we say that a rule r *derives* Q w.r.t. I if there exists a ground instance $\overline{r}$ of r such that $H(\overline{r})=Q$ and $B(\overline{r}) \subseteq I$.

The well-founded semantics of ordered logic programs is based on a suitable extension to ordered logic programs of the notion of unfounded set given for traditional logic programs. Such an extension is needed to take into account the presence of *contradicting rules*, i.e., rules that may overrule or defeat derived literals. The basic idea is formalized by the following definition.

Definition 1. Given a ground instance $\overline{r}$ of a rule r belonging to a component C_i, we say that the ground instance $\overline{r'}$ of a rule $r' \in C_j$ is *contradicting* for $\overline{r}$ if both the following conditions hold:

[1] Two literals are complementary if they are of the form, say, A and $\neg A$

1. the heads of $\overline{r}$ and $\overline{r'}$ are complementary literals, and
2. either $C_j \leq C_i$ holds, or C_j and C_i are incomparable (i.e. neither $C_j \leq C_i$ nor $C_i \leq C_j$ hold). ■

Intuitively, given a ground rule r, the presence of a contradicting rule r', whose body does not contain any false literal, indicates that we can not utilize r to infer its head literal, as it is either defeated or overruled by the rule r'.

Example 4. For instance, let us consider the simple program consisting of just one object $<o_1, C_{o_1}>$, where:

$C_{o_1} = \{ \; p. \qquad (r_1)$
$\neg p :- q. \qquad (r_2)$
$\neg p. \; \} \qquad (r_3)$

It is easy to recognize that rule r_1 is contradicting for r_2 and r_3, and viceversa. In this case, intuitively, we can infer neither p nor $\neg p$. □

The notion of contradicting rule represents a first step towards the definition of unfounded sets. Indeed, it is reasonable to count as unfounded all literals for which every deriving rule admits a contradicting one whose body does not contain a false literal, i.e. such that it could eventually determine a contradiction. However, it seems intuitive to discard also the literals which are self contradictory, i.e. each literal A such that the truth of A would imply the truth of $\neg A$. This concept is formalized by the following definition.

Definition 2. Let I be an interpretation for a given program $\mathbb{P}_o$. A rule $r \in ground(\mathbb{P}_o)$ is *contradictory* in I if there exists a (finite) sequence $[r_1, \ldots, r_n]$ of ground rules such that:

1. r_n is contradicting for r,
2. $\forall 1 \leq i \leq n$ the body of r_i is true w.r.t. IH_i, and
3. $\forall 1 \leq i \leq n-1$ every contradicting rule for r_i contains a literal false w.r.t. IH_i in its body.

where $IH_1 = I \cup H(r)$ and $IH_i = IH_{i-1} \cup H(r_{i-1})$. ■

Example 5. Let us consider the simple program P_1 containing only the object $<o_2, C_{o_2}>$, where

$C_{o_2} = \{ \; p. \qquad (r_0)$
$\neg p :- p. \; \} \qquad (r_1)$

It is easy to recognize that the ground rule r_0 is contradictory in the empty interpretation $I=\emptyset$. Indeed, the sequence $[r_1]$ verify the conditions above, as r_1 is contradicting for r_0 and its body is true w.r.t. $IH_1=\{p\}$.
Note that the truth of p would implies $\neg p$ (via the rule r_1). □

Now we are ready to define the notion of unfounded set for ordered logic programs.

Definition 3. Let I be an interpretation. A set U of ground literals is an *unfounded* set w.r.t. I if for each ground rule r whose head is in U either

1. the body of r contains a literal which is in U, or
2. r is *contradictory* in I. ■

Clearly, the union of any pair of unfounded sets is an unfounded set. Thus, the *greatest unfounded set* w.r.t. I, denoted $U_{I,\mathbb{P}_o}$ is the union of all the unfounded sets w.r.t. I.

Intuitively, $U_{I,\mathbb{P}_o}$ is the set of all ground literals that we are sure not to be a logical consequence of $\mathbb{P}_o$ and I, since their truth either comes from assumptions (condition 1) or would cause inconsistency (condition 2).

The well-founded semantics of an ordered logic program, say P, with respect to an object o, is now defined by the following operator:

$$W_{\mathbb{P}_o}(I) = \{Q \in B_{\mathbb{P}_o} - U_{I,\mathbb{P}_o} \mid \exists r \in ground(\mathbb{P}_o)\ s.t.\ r\ derives\ Q, \text{ and } \forall\ contradicting\ rule\ r', B(r') \cap U_{I,\mathbb{P}_o} \neq \emptyset\}$$

Consider the sequence $W_0 = \emptyset$, $W_n = W_{\mathbb{P}_o}(W_{n-1})$. It is easily recognized that $\{W_n\}$ is a monotonic sequence of interpretations. Hence, there exists the first finite λ such that $W_\lambda = W_{\lambda-1}$ (recall that the Herbrand Base of an ordered logic program is finite). W_λ is a fixpoint for the $W_{\mathbb{P}_o}$ operator that we denote by $W^{\infty}_{\mathbb{P}_o}(\emptyset)$. $W^{\infty}_{\mathbb{P}_o}(\emptyset)$ is the *well-founded semantics* of P w.r.t. o, i.e. the set of properties holding at o, or conversely, the part of knowledge perceived by o.

Notice that there are two main differences between the well-founded semantics of ordered logic programs and that of traditional ones. The first is that in ordered logic programming also negative literals need to be derived. The other is that a literal A can be inferred from a rule r only when it is definitely acquired that A will never be contradicted (that is, all the contradicting rules for r have in their bodies an unfounded literal).

Let us now illustrate the application of the operator $W_{\mathbb{P}_o}$ by tracing the computation of the well-founded semantics in two simple cases.

Example 6. Let the program P be composed of three objects $<o_1,C_{o_1}>$, $<o_2,C_{o_2}>$, $<o_3,C_{o_3}>$, where

$C_{o_1} = \{\ p.\ \}$ (r_1)
$C_{o_2} = \{\ \neg q.\ \}$ (r_2)
$C_{o_3} = \{\ \neg p\ :-\ p.$ (r_3)
$\quad\quad q\ :-\ p.\ \}$ (r_4)

and the relationships $o_3 \leq o_1$, $o_3 \leq o_2$ hold. Hence, $\mathbb{P}_{o_3}$ contains all of the above rules, as r_3 and r_4 belong to the definition of o_3, while r_1 and r_2 are inherited. First of all, given the interpretation $W_0=\emptyset$, the greatest unfounded set w.r.t. W_0 is the set $S = \{p, \neg p, q\}$. In fact, S is an unfounded set as: i)the only rule for p (i.e. r_1) is contradictory (analogously to Example 5); ii)the only rule for $\neg p$ (i.e. r_3) contains a literal which is in S (i.e. p); and iii)in turn, the only rule for q (i.e. r_4) contains a literal in S, p again. Further, S is the greatest unfounded set, as its superset $S' = S \cup \{\neg q\}$ is not an unfounded set. This is because the rule r_2, whose head is in S', does not verify either conditions in Definition 3. Hence, $U_{W_0,\mathbb{P}_{o_3}}$ is the set S.

By applying the operator $W_{\mathbb{P}_{o_3}}$ to W_0, we obtain the interpretation $W_1 = \{\neg q\}$. In fact, rule r_2 derives $\neg q$ and the only contradicting rule r_4 contains an unfounded literal in its body. No more literals can be added to W_1, as all others are unfounded. Thus, W_1 is the well-founded semantics of P w.r.t. o_3. □

Example 7. Consider again the objects of example 6 but with a different $\leq$ relation. In particular, assume that $o_1 \leq o_3 \leq o_2$
$\mathbb{P}_{o_1}$ contains the rules $r_1 \ldots r_4$ as above. It is easy to recognize that the greatest unfounded set w.r.t. $W_0 = \emptyset$ is the set $U_{W_0,\mathbb{P}_{o_1}} = \{\neg p\}$. Thus, $W_1 = W_{\mathbb{P}_{o_1}}(W_0) = \{p\}$. Now, the greatest unfounded set w.r.t. to W_1 is the set $\{\neg p, \neg q\}$, as rule r_2 is contradictory (the sequence $[r_4]$ trivially verifies all conditions of Definition 2 for r_2)
The subsequent application of the well-founded operator computes the interpretation $W_2 = \{p, q\}$ which is the least fix-point of $W_{\mathbb{P}_{o_1}}$, i.e. the well-founded semantics of P w.r.t. o_1. □

We conclude by reporting a rather nice and short example ([LSV90]) to show the expressiveness of the language for uncertain knowledge.

Example 8. Let the program P be composed of four objects $<o,C_o>$, $<o_1,C_{o_1}>$, $<o_2,C_{o_2}>$, $<o_3,C_{o_3}>$ where

$C_{o_1} = \{\ take\_loan \leftarrow inflation(X), X>11.\}$ (r_1)
$C_{o_3} = \{\ \neg take\_loan \leftarrow loan\_rate(X), X>14.\}$ (r_3)
$C_{o_2} = \{\ take\_loan \leftarrow inflation(X), loan\_rate(Y), X>Y+2.\}$ (r_2)
$C_o = \{\ inflation(12).\}$

and the relationships $o \leq o_1$, $o \leq o_2 \leq o_3$ hold. In practice, the knowledge of three experts —namely *Expert* 1, *Expert* 2, *Expert* 3— has been embodied respectively into the objects o_1, o_2, o_3, while the 'object' o knows only simple facts and asks for sensible suggestions. Further, *Expert* 2 bases his advice on *Expert* 3's knowledge, even if his rule can sometime contradict the rule of *Expert* 3 (o_2 refines the knowledge of o_3).

Now, at o it is possible to infer *take_loan* from the knowledge of *Expert* 1 (as the body of rule r_1 is true for X=12), while neither r_2 nor the contradicting rule r_3 apply (as they contain the unfounded literal *loan_rate* in their body).

Imagine, now, that o is able to establish a new fact for *loan_rate*, so that C_o becomes:

$C_o = \{\ inflation(12).$
$\quad\quad loan\_rate(16).\ \}$

As a consequence, both *take_loan* and $\neg$*take_loan* can be derived (from rule r_1 and r_3, respectively): they *defeat* each other and no certain information can be asserted at o.

Finally, take the following situation:

$C_o = \{\ inflation(19).$
$\quad\quad loan\_rate(16).\}$

The rule r_3 derives $\neg$*take_loan* but it is overruled by rule r_2. This allows to infer *take_loan* at o. □

4. CONCLUSIONS AND FUTURE WORK

The work presented in this paper has been developed within the framework of the ESPRIT project KIWIS, an advanced knowledge-base environment for large database systems. The system can be used as a sophisticated stand-alone "personal knowledge machine", as well as a "window on the world" that provides a seamless integration of information from a wide variety of external sources with the local knowledge base.

The idea of ordered logic programming has been fully embodied into LOCO, the main language of KIWIS, which also includes typing, updates, transactions, and triggers. Notice that the ability of dealing with contradicting, uncertain knowledge, essentially provided by ordered logic programming, makes the system suitable also for AI-flavored applications, such as expert systems.

As to the operational semantics for ordered logic programs, it is not surprising that its computation is quite demanding, mainly for the overhead involved in the resolution of contradictions. To this regard, some work has been done in [Leo90], where an operational semantics for computing a model not as rich as the well-founded model (i.e. the *least* model of ordered logic programs) was presented.

At present, work is on going to single out a meaningful class of programs, basically characterized by a regular form, whose well-founded semantics can be computed in a monotonic fashion.

Acknowledgments

We want to thank P. Rullo for the many suggestions and encouragements while preparing this work. We are also in debt with E. Laenens and D. Vermeir for useful discussions and contributions.

References

[Ait86]
Ait-Kaci, H. and Nasr, R. "LOGIN: A Logic Programming Language with Built-in Inheritance", *Journal Logic Programming*, no. 3(3), Oct. 1986.

[BS89]
Blair, H. and Subrahmanian, V.S. "Paraconsistent Logic Programming", *Theoretical Computer Science*, vol. 68, pp. 135-154, 1989.

[CW89]
Chen, W. and Warren, D.J. "C-logic for Complex Objects", *Proc. ACM SIGACT-SIGMOD-SIGART Symp. on Principles of Database Systems*, March 1989.

[Con87]
Conery, J.S. "Object Oriented Programming with First Order Logic", Tech. Report CIS-TR-87-09, Univ. of Oregon

[GL89]
Gelfond, M. and Lifschitz, V. "Logic Programs with Classical Negation", Sept. 1989, Stanford University.

[GLR89]
Greco, S., Leone, N., and Rullo, P. "COMPLEX: An Object-Oriented Logic Programming System", CRAI, Research Report, Sept. 1989.

[GS90]
Greco, S. and Saccà, D. "Negative Logic Programs", *Proc. of North American Logic Programming Conference* , 1990.

[KL89]
Kifer, M. and Lausen, G. "F-logic: A Higher-Order Language for Reasoning About Objects, Inheritance, and Scheme", *Proc. of the ACM SIGMOD Int. Conf. on the Management of Data*, Portland, Oregon, 1989.

[KLW90]
Kifer, M. Lausen, G., and , J. Wu "Logical Foundation of Object-Oriented and Frame-Based Languages", Tech. Report 90/14, dept. of Computer Science, New York State Univ. at Stony Brook, June 1990.

[KS89]
Kowalski, R.A. and Sadri,. F. "Logic Programs with Exceptions ", Tech. Report. Imperial College, Dept. of Computing, London, November, 1989.

[LV90]
Laenens, E. and Vermeir, D. "Assumption-free semantics for ordered logic programs: on the relationships between well founded and stable partial models", Univ. of Antwerp, Tech Report 90-19, 1990.

[LSV90]
Laenens, E., Saccà, D., and Vermeier, D. "Extending Logic Programming", Proceedings of ACM SIGMOD, May 1990.

[Lae90]
Laenens, E. "Foundations of ordered logic", Ph.D. thesis, Univ. of Antwerp, 1990.

[Leo90]
Leone, N., Mecchia, A., Romeo, M., Rossi, G. and Rullo, P. "From DATALOG to Ordered Logic Programming", Proc. of the 13th Int. Seminar on DBMS, Mamaia (Romania), Sept. 1990.

[NT88]
Naqvi, S. and Tsur, S. *A Logical Data Language for Data and Knowledge Bases*, Computer Science Press, 1988, New York.

[Prz89]
Przymusinski T.C. "Every logic program has a natural stratification and an iterated fixed point model", in PODS, 1989.

[GRS88]
van Gelder A., Ross, K. and Schlipf, J.S., "Unfounded Sets and Well-Founded Semantics for General Logic Programs", Proc. ACM SIGMOD-SIGACT Symp. on Principles of Database Systems, pp. 221-230, March 1988.

PARALLELISMO E CONCORRENZA

(*Parallelism and Concurrency*)

A PARALLEL LOGIC PROGRAMMING LANGUAGE FOR DISTRIBUTED ARCHITECTURES

Antonio Brogi (*), Anna Ciampolini (**), Evelina Lamma (**), Paola Mello (**)

(*) Dipartimento di Informatica
Università di Pisa
Corso Italia 40
56125 Pisa - Italy

(**) Dipartimento di Elettronica, Informatica e Sistemistica
Università di Bologna
Viale Risorgimento 2
40136 Bologna - Italy

ABSTRACT

The distributed implementation of a new communication model for parallel logic programming is presented. The main novelty with respect to STREAM-parallel logic languages is that inter-process communication is not performed via variables shared among AND processes but by means of multiple-headed clauses. A compilation technique on an abstract machine inspired to the Warren Abstract Machine is described where new instructions and data structures are introduced for process creation and communication and control of non-determinism. To show that this model is suitable for distributed architectures, we discuss a first prototype for a transputer-based architecture.

1 INTRODUCTION

In recent years, many efforts have been devoted to the definition of new parallel logic programming languages.
In the process interpretation of logic programming [Sha83] a goal is viewed as a network of processes communicating via shared variables, while the clauses of the program correspond to alternative definitions of the processes.
Although shared variables among parallel logic processes provide a basic communication mechanism, they are not adequate for synchronizing concurrent computations.
In STREAM-parallel languages [Sha89], constraints on the unification mechanism have been introduced to synchronize processes via shared variables. The sharing of variables between parallel processes represents, however, a centralization point in the resulting computational model.
In the context of the Japanese project for Fifth Generation Computer Systems, where several implementations of concurrent logic languages have been developed, the sharing of variables between parallel processes constitutes the main problem to be faced when aiming at realizing distributed implementations [Nak89] [Ued90].
Other approaches propose clauses with a conjunction of atoms in the head (multiple-headed clauses) [Fal84] or events [Mon84] for synchronizing parallel processes in logic programming.
Multiple-headed clauses are defined to be simultaneously applied to matching conjunctions of parallel goals.

In this paper we present a new communication model for parallel logic programming where AND parallel processes do not share variables, and inter-process communication is performed via multiple-headed clauses.
The distributed nature of the resulting language (named ROSE [Bro90]), has been formally demonstrated by the definition of a true concurrent operational semantics.
This paper, complementary to [Bro90], mainly focuses on implementation issues by presenting the process model of ROSE and its implementation scheme on a distributed architecture.
Avoiding the sharing of variables between AND processes notably simplifies the realization of a distributed implementation with respect to STREAM-parallel languages. On the other hand, the unification phase is more complex and, correspondingly, a different structure for the process network with respect to STREAM-parallel languages is adopted.
In particular, the AND/OR tree of processes modeling parallel logic programming [Con85] is here a graph, while a tree of "unification" processes is introduced to perform multiple-head unification in an incremental way.
There are several issues to be addressed in order to obtain an effective, distributed implementation of this parallel model.
One issue involves the design of an abstract machine and the compilation of ROSE programs on it. Another one concerns the dynamic construction of the AND/OR process graph and the unification tree, assigning processes to processors and scheduling processes once a particular distributed architecture has been chosen.
The implementation of ROSE is obtained in terms of compilation on an abstract machine (called ROSE Abstract Machine - RAM - in the following) which is a rather natural extension of the one for Prolog (Warren Abstract Machine - WAM [War83]).
Essentially, in RAM new instructions are introduced for process creation and communication, and non-determinism control. Accordingly, new data structures are added for process handling.
The architectural scheme is a parallel Multiple Instruction Multiple Data (MIMD) machine with distributed memory, and consequently the parallel model is based on message-passing. In particular, for the first prototype, we have adopted an architecture based on the transputer technology [INM88].

2 ROSE

In this section an overview of ROSE is given. A complete description of the language, including its semantics and several programming examples, can be found in [Bro90].
ROSE is a proper extension of Horn Clause Logic, where multiple-headed clauses of the kind:
(*) $A_1+...+A_m <- B_1+...+B_n$
are allowed.
The "+" parallel composition operator (inspired by [Fal84]) may occur both in left- and right-hand side of a clause. The language forbids the sharing of variables among parallel goals in the body of a clause (see [Bro90]) to always obtain independent AND processes.
In order to apply a multiple-headed clause, like (*), a parallel composition of atoms, $A_1'+..+A_m'$ say, unifying with the clause heads has to occur in the current goal, so that there exists a substitution σ: $\sigma=mgu((A_1+...+A_m),(A_1'+..+A_m'))$.
For example, given the clause:
a(X) + b(Y) <- g1(X) + g2(Y)
the goal: <- a(1)+a(2)+b(3) can be reduced to <-g1(1)+g2(3)+a(2) by unifying the head (a(X)+b(Y)) with (a(1)+b(3)), or to <-g1(2)+g2(3)+a(1) by unifying the same head with (a(2)+b(3)).

The focus of the present paper is on the distributed aspects of the language, thus only the parallel composition operator ("+") will be considered by omitting the description of the sequential part of ROSE ([Bro90]).

The definition of ROSE has been extended in [Bro90a] to introduce a committed-choice behaviour. Don't care non-deterministic computations are defined by means of a guard mechanism as in the family of STREAM-parallel languages [Sha89].
Guarded multiple-headed clauses have the form:
(**) Head <- Guard | Body
where Head and Body are parallel conjunctions of atoms (i.e. $A_1+...+A_m$) and Guard is composed of system predicates only to make the implementation easier.
The meaning of the guard is that a clause like (**) is applicable (or candidate) to a given goal iff both the head unification succeeds and the guard is satisfied. For instance, the clause:
a(X)+b(Y) <- X>Y | Body.
can apply to the goal <- a(3)+b(1), but not to the goal <-a(1)+b(1).

The operational semantics of the language is given by the following rewriting rule, according to the true concurrent model presented in [Bro90].
The initial goal is a parallel conjunction of atoms, denoted by a multiset of atoms (e.g. den(a(1)+a(2)+b(3)) = {a(1)}$\cup$ {a(2)} $\cup$ {b(3)}.
The application of a clause to a goal is described by:

MultiHead <- Guard | Body is a clause of the program,
σ=mgu(MultiHead,$(A_1+...+A_n)$), Eval((Guard)σ)=σ'

$\{A_1\}\cup ... \cup \{A_n\}$ —> den((Body)$\sigma\sigma'$)

where the function Eval denotes the evaluation of a guard possibly resulting in a computed substitution σ'.
This rewriting rule specifies how a subset of the current goal (denoted by $\{A_1\}\cup ... \cup \{A_n\}$) in the current state can evolve in den((Body)$\sigma\sigma'$), without affecting the remaining part of the goal.
A derivation is a (possibly infinite) sequence of states obtained starting from the initial goal and repeatedly applying the rule above.
The true concurrent description is achieved by allowing simultaneous applications of several rules in parallel to non-intersecting partitions of the state. More in detail, given a program P and a goal (state) S_{i-1}, we get a new state S_i as follows:

S_{i-1} : G1—>G1' ... Gn—>Gn' : S_i

where Gi (i=1..n) are non-intersecting partitions of S_{i-1}, and
$S_i = S_{i-1} \setminus (G1 \cup ... \cup Gn) \cup (G1' \cup ... \cup Gn')$.
A computation is successful if the empty state is reached.

Notice that AND-processes sharing variables can be translated into equivalent ROSE processes that do not share variables, but communicate through multiple-headed clauses.
For instance, let us consider the following program:
p(X)<- G1|B1.
q(X)<- G2|B2.
and the goal :

<-p(X?)+q(X) where processes p and q share the X variable that is produced by q and consumed by p.
It can be translated into the following, equivalent one:
p(X) + e(X)<- G1|B1.
q(X)<- G2| e(X) + B2.
and the goal:
p(X) + q(Y) where now processes p and q do not share variables, but communicate through the explicit event e(X).

As further example, we discuss the well known problem of the airline reservation system. Suppose there are n travel agencies connected on a computer network. The agencies share information concerning flights, and concurrently ask for the number of available seats on a flight or try to reserve a number of seats on a certain flight.
If one wants to exploit the fine-grained parallelism supported by ROSE, all the information concerning the flights can be distributed on the system by representing each flight record with a distinct atom.
The initial ROSE goal has the form:
<- agency(ag1) + .. + agency(agn) + flight(f1,100)+ .. + flight(fn,80)
Each travel agency, X say, produces a request R by some sequential derivation (eventually via an user interface) and generates a pending request where its own name is recorded. Then the agency waits for an answer to its request, consumes it by some sequential computation and recalls itself.
agency(X) <- produces(R), request(X,R)
answer(X,A) <- consumes(A), agency(X)
The airline company receives messages and processes the requests (pending on the current goal) by possibly modifying the database (e.g. by reserving seats on a flight).
This behaviour is specified by two two-headed clauses which apply to a request(X,R) and to a flight record containing the number of free seats on that flight (sequential conjunction is represented by comma).

request(X,no_free_seats(Flight)) + flight(Flight, Free)<- answer(X,Free) + flight(Flight, Free)
request(X,reserve(Flight,No_seats))+flight(Flight,Free)<-
mkreservation(No_seats,Free,Reserved,Free1), answer(X,Reserved) + flight(Flight, Free1)

3 A FULLY DISTRIBUTED EXECUTION MODEL

In this section we define a fully distributed execution model for ROSE. The model is based on AND and OR processes which execute asynchronously and communicate by message-passing.
Since the language forbids the sharing of variables among parallel goals in the body of a clause, AND processes are always independent. We are aware that, in the general case, it is not possible to guarantee this property by means of syntactic checks only, but we can take advantage techniques already exploited in the literature (see [Mut90] and [Her90a]) have to be applied.
The innovative issue is the implementation of the multiple-head unification which is the basic mechanism of ROSE for process interaction.
In particular, an AND/OR graph of processes and an incremental, parallel unification scheme are introduced.

3.1 AND/OR graphs of processes

In order to describe the process model of ROSE, the AND/OR process tree of parallel logic programming [Con85] is replaced by a directed acyclic AND/OR graph.
The initial goal (<-g_1+...+g_k) is represented by an AND node with k OR nodes as children, one for each atom composing the goal.
Each of these OR nodes is thus labeled by some atom g_i and has a child AND node for each clause such that one atom of its head unifies with g_i.
In the resulting graph, each clause is represented by several AND nodes corresponding to different evaluations of its applicability. Each of these nodes has (possibly) as many fathers as the atoms composing the head of the clause.
When a clause is chosen for being applied, as many children OR nodes as the goals composing the body of the clause are created.
For an example, see figure 1.

A process is assigned to each node of the AND/OR graph. As in the model presented in [Con85], each AND node corresponds to an OR process and each OR node to an AND process.

Let us consider the clause C:
a(X)+b(Y)<- g1(X)+g2(Y).
and the goal <- a(1)+a(2)+b(3).
We can map this program in the following AND/OR graph of processes:

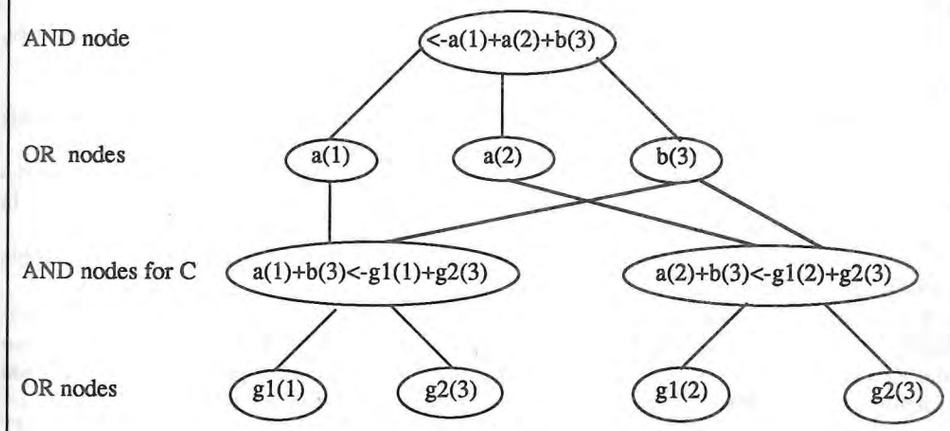


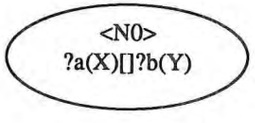
Figure 1: An AND/OR graph

It is worth observing that the graph description is equivalent to the operational one given in section 2, but the AND/OR graph offers a finer representation, taking into account also the OR parallel alternative evaluations.

As in all "committed-choice" languages [Sha89], in ROSE OR-parallelism is limited to head unification and guard evaluation, which is sequential for the sake of simplicity. "Committing" to only one OR process simplifies the AND/OR graph structure and therefore the implementation. In practice, only the "committing" OR process will continue the computation by generating the corresponding AND processes, while sibling OR processes can be killed.
However, the commit implementation is more complex with respect to STREAM-parallel languages due to multiple-head unification.
Whenever an OR process tries to commit, after the successful execution of the clause guard (candidate OR process), it has to send an explicit notification to more than one AND process, i.e. to all its parent nodes.
For example, with reference to figure 1, both the candidate OR processes labeled by a(1)+b(3)<-g1(1)+g2(3) and a(2)+b(3)<-g1(2)+g2(3), after the successful guard execution, try to notify the commitment to b(3), but of course only one of them can successfully commit by reducing the goal:
<-a(1)+a(2)+b(3).
A two-phase lock protocol between the candidate OR processes and the corresponding AND processes is provided in order to guarantee that only one candidate OR process successfully commits. Moreover, to avoid deadlock, an order in sending messages during the lock protocol is imposed. More details will be given in section 5.

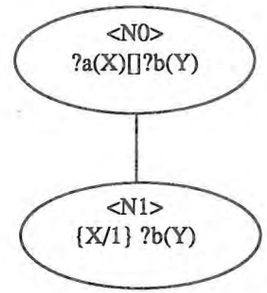
3.2 Incremental unification

An incremental parallel unification scheme is introduced for creating the AND nodes of the AND/OR graph, and their associated OR processes.
The process model is, in practice, a dynamic creation of different, communicating processes, corresponding to the same clause C, each one responsible for a single head unification.
For each clause, a "manager" process is created to perform multiple-head unification.
Each AND process, responsible for an atomic goal g, sends a message (g) to each manager process of a clause C= h_1+...+h_n <- Goall Body having a head h_j with the same predicate symbol of g.
After receiving the message, the manager of C creates a new child process in order to perform the unification of g with h_j and routes the received message to all the already created children. They, in turn, can create new children to perform subsequent unifications at new message receptions.
Such incremental unification corresponds to a tree of "unification" processes, each one responsible for a single head unification.
Let us consider, as an example, the clause C:
a(X)+b(Y)<- truel g1(X)+g2(Y).
Initially, a manager process, N0, is created for C. N0 non-deterministically waits for some atom which may unify with a(X) or b(Y). It can be represented as the following node:

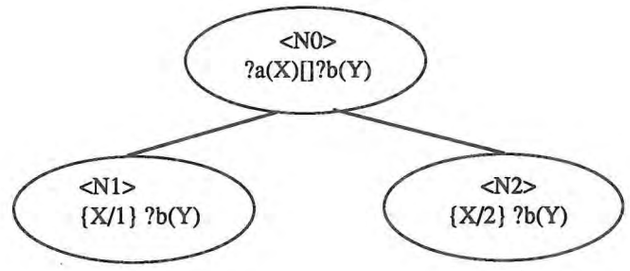


where "[]" denotes the alternative command, and "?A" indicates that the process waits for an atom A.
Given the parallel goal:
<-a(1)+a(2)+b(3).
three AND processes, P1, P2 and P3, are created for a(1), a(2) and b(3) respectively.

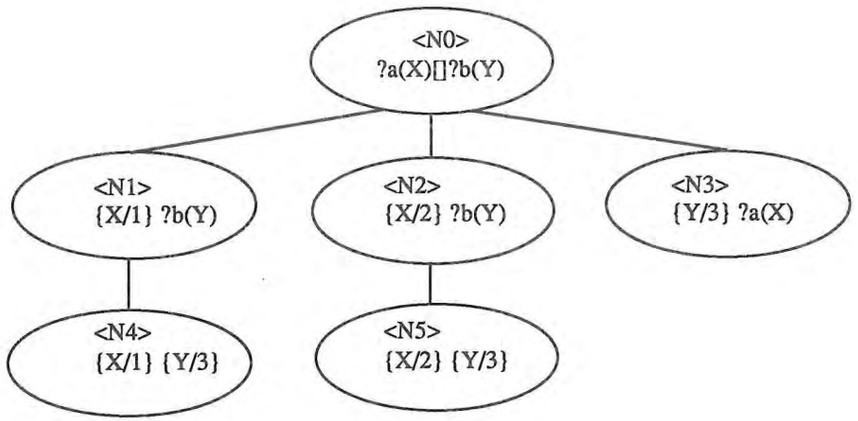
If P1 sends the corresponding message (a(1)) to N0, N0 creates one child, N1, for unifying a(1) with the a(X) variant.



If now P2 sends the corresponding message (a(2)) to N0, N0, as previously, creates one child process (N2) for unifying a(2) with the first head of C, and routes the message also to the already created child N1.



If now P3 sends the corresponding message (b(3)) to N0, N0 creates one child process (N3) for unifying b(3) with the second head of C, and routes the message also to the already created children N1 and N2.
N1 and N2, in turn, create the child processes N4 and N5 to perform the new head unification.
Thus, the unification tree grows as shown in the next figure.
Notice that processes N4 and N5 both complete the multiple-head unification, and therefore they correspond to the OR processes of the AND/OR graph. They start the guard evaluation and only one of them will commit.
Moreover, parallel unifications of a(1), a(2) and b(3) with the clause heads take place, thus giving origin to (parallel) instances of the sam clause.
All the other nodes of the tree represent processes which have not yet completed the multiple-head unification and wait for other messages.



Let us note that the configuration of the tree can be different, depending on the scheduling of the messages. In the example above, we assume the order a(1), a(2), b(3) in receiving the messages. A different order would generate a different unification tree.
However, the following property holds: given a set of messages (goals) and a clause C, whatever the order of messages, the leaves ofm the tree with depth equal to the number of heads are always the same.

There is a direct correspondence between this scheme for incremental unification and the operational semantics given in section 2. In particular, the resulting OR processes of a unification tree correspond to alternative applications of the rewriting rule.
Actually, if an OR process <Nj> (corresponding to a clause MultiHead <- Guard | Body) has received all its messages (which correspond to a goal G) and has succeeded in evaluating its guard, it can try to commit. Whenever it commits, it can be proved that the substitution associated to the process itself is exactly the substitution $\sigma\sigma'$, where σ=mgu(MultiHead,G) and σ'=Eval((Guard)σ) as specified in the rewriting rule.

4 THE ROSE ABSTRACT MACHINE

The abstract machine developed for the parallel execution of ROSE (hereinafter called ROSE Abstract Machine - RAM for short) draws ideas from the Warren Abstract Machine (WAM) [War83], although there are some significant differences in the code generated due to the process model and the need for a multiple-head unification.

4.1 Environment variables

When implementing parallel and distributed logic programming models, the environment representation has to be deeply revised (see [Con87, Kal88]).

In our particular case, the notion of *environment* of a clause as defined for standard WAM has to be further revised, since we deal with extended clauses, where we can have multiple heads and no variable sharing among AND parallel goals. In particular, environment variables (called permanent in the WAM) are only those shared among different heads of a clause or those shared between one (or more) head(s) and one goal in the body.
Since in a parallel implementation of logic programming permanent variables represent data shared among processes, the revised notion of environment variable in ROSE implies that each AND process cannot share any environment variable with any other AND process, but only with some OR process. Moreover, environment variables can be shared among the processes belonging to the same unification tree.
Notice that the commit-choice nature of the language, adopting "don't care" non determinism, does not require the use of multiple environments to follow alternative paths originated , which would assign different values to the parent variables (see [Con87, Bos90]).
In a fully distributed implementation, this sharing of variables implies that both forward and backward unification have to be performed properly through dereferencing and message passing [Ram88].
Forward unification takes place whenever a unification process is created and successfully performs unification. Input values for unification are contained in the message sent by the corresponding AND process and routed, if needed, by other unification processes.
Backward unification takes place whenever an OR process successfully ends its body. Output values for variables are therefore returned through message passing to the involved AND processes.
The distributed implementation of backward and forward unification requires a great number of communications and therefore a high overhead. However, thanks to the ROSE computational model that forbids the sharing of variables among parallel goals, backward unification can be performed only for the variables of the top-level goal and variables bound to them. We will refer to those variables as "output" variables.

Let us consider the following ROSE program:
C1: a(X)+b(Y)<-g1(X),g2(Y).
C2: c(K)+d(K)<-a(K)+b(J).
C3: g1(1).
C4: g2(2).
and the top-level goal:
<-c(Z)+d(1).
In C1, both X and Y are environment variables, but during the computation only X is an output variable for which backward unification has to be performed. In fact, only the output value of X (i.e. 1), bound to the variable Z of the top goal, is of interest.

Notice that output variables, in the most general case, can be determined only at run-time, since they depend on the top-goal.
For this reason, we extend the representation of environment variables by an additional tag, in order to distinguish output variables. This optimization, not applicable in "STREAM"-based languages, implies a significant improvement of the distributed implementation of ROSE, since backward unifications are performed only for tagged variables and therefore the communication load is highly reduced.

4.2 Process Representation

Initially, a top-level AND parallel goal is spawned in a number of AND processes and one manager process is created for each multiple-headed clause.
After the creation, each AND process sends a message containing its corresponding goal to each manager process of interest and then it suspends, waiting for a success or a failure message.
In the case of success, the resulting message will contain the bindings for the "output" variables.
Each unification process, manager included, at message reception, if the message is of interest, creates a new process to perform the next head unification (see section 3). When the last head unification is successfully performed (by an OR process), the guard and possibly the commit protocol is executed (see section 5).
Of course, data structures of the WAM have been extended in RAM to deal with AND, manager, unification and OR processes.
A descriptor is associated to each process. In particular, the AND process descriptor contains:

1) the identifier of the father process (**Father**), in order to propagate success or failure;
2) the program instruction pointer (**P**);
3) a flag (**Commit**) that is set when one of the corresponding OR processes commits and a flag (**Lock**) to perform consistently the two phase lock protocol; 4) a list of manager process identifiers (ManagerList) of managers whose clause heads can unify with the AND goal predicate. This kind of information can be statically obtained.

Each OR process descriptor contains the following information:

1) a list of identifiers (**FatherList**) of the AND processes whose messages have been successfully unified, in order to properly perform the two-phase lock protocol (in case of COMMIT) and to propagate success (and "output" variable bindings) or failure results;
2) the program instruction pointer (**P**);
3) a counter (**count**), as in [Nys88], representing the number of AND processes that have this node as their father, for propagating success. This counter is initially set to the number of child AND processes. Each time one success message is received, the counter is decremented. If the counter has reached zero, the success is propagated from the OR process to the father AND processes.

Let us notice that for success and failure propagation after commitment, OR processes need to know only the AND processes whose messages have been successfully unified, thus forgetting any information about their (unification process) creators.
When receiving a message from an AND process, the Manager generates one or more unification processes with the same descriptor structure. The manager or unification process descriptor contains the following information:

1) the program instruction pointer (**P**);
2) a list of identifiers (**FatherList**) of the AND processes whose messages have been successfully unified. A copy of this structure is passed to each new process created by the unification/manager process (suitably upgraded with the new unification information);
3) a list (**ChildList**) of the processes corresponding to the child nodes in the unification tree, in order to perform the routing of messages, together with their unification state in order to perform a "more intelligent" routing of the messages. In fact, the incremental unification mechanism implies a very heavy message traffic across the network. An efficient routing of messages between processes in the tree can be obtained by forwarding messages only to those processes which can really consume them, i.e. that can unify.

4.3 New Instructions

Each goal connected by the parallel composition operator ("+") is compiled into an AND process creation by using the new **create_and** instruction.
The new **create_or/create_unif** instructions are introduced to create respectively OR and unification processes. All the create instructions specify as arguments the label of the code to be executed.
Each AND process will execute a **send_msg pred,N** instruction to send the corresponding goal to each manager process of interest, and then suspends.
If h1,...,hn <- GlB. is the clause associated with a manager process P, P executes a **wait [[h1,L1],...,[hn,Ln]]** instruction waiting, in a non deterministic way, for some message of interest. If the message hi is selected, a jump to the label Li is performed.
In the compiled code, each guard is followed by the new **commit** instruction.
As an example, the compilation of the clause C:
a(X)+b(Y) <- truelg1(X)+g2(Y).
produces the following code:

```
% code for the manager process
C    wait [[a/1,La],[b/1,Lb]]
La   create_unif L1a
     execute C
Lb   create_unif L1b
     execute C

% code for unification processes (depth 1)
L1a  allocate 2
     get_variable A1,Y1
     wait [a/1, L2b]
L2b  create_or LL2b
     execute L1a

L1b  allocate 2
     get_variable A1,Y2
     wait [a/1, L2a]
L2a  create_or LL2a
     execute L1b

% code for OR processes (depth 2)
LL2b get_variable A1,Y2
     commit
     execute B

LL2a get_variable A1,Y1
     commit
     execute B

% code for the body of C
B    put_value Y1,A1
     create_and Lg1
```

```
        put_unsafe_value Y2,A1
        deallocate
        create_and Lg2

% code for AND processes
Lg1   send_msg g1,1

Lg2   send_msg g2,1
```

Notice that only processes of depth 1 explicitly allocate the environment (allocate instruction). This environment will be copied in the descriptor of each child process whenever the create_or/create_unif instruction is executed. For AND processes, since they do not share variables, the only information needed are parameters which are passed through argument registers. In theory, the environment can be deallocated after the value of evironment variables has been loaded into registers.

5 IMPLEMENTATION ON A DISTRIBUTED ARCHITECTURE

In this section, we sketch the real implementation of ROSE on a parallel architecture. More details can be found in [Bro91].
The chosen architectural scheme is a parallel MIMD machine with distributed memory, and the process interaction scheme is based on message-passing. In particular, we will refer to a Meiko Computing Surface architecture, which is based on the transputer technology [INM88]. The configuration adopted for the first implementation of ROSE is composed by twelve transputers, each one with 4 Mbyte of private memory.
With reference to the programming tools and languages to be used, the OCCAM language [Bur88] is too limited since it allows neither the dynamic creation of processes nor asynchronous communication. In the first ROSE implementation we used CSTools [CST89], a more flexible programming tool obtained by extending the C language with primitives for process creation and interaction.

A number of problems have been considered and faced when implementing ROSE:

- First of all, as in [Bos90], our execution model is characterized by an "eager" creation of processes, which is in contrast, for example, with the policy adopted in [Sze89] and [Her90b] where the number of parallel activities is naturally limited to the number of physical processors. Clearly, this choice implies, in the worst case, a combinatorial explosion of processes but the programmer can directly control the parallelism, since ROSE supports both sequential and parallel computations. For the first prototype, we rely on the scheduler of each transputer for scheduling processes allocated on the same node. In the future, we plan to investigate more sophisticated policies by explicitly implementing scheduling algorithms.

  Another topic to be faced concerns a correct distributed implementation of the commit phase, which must determine the exclusive assignment of a pool of AND "resources" to only one OR "consumer" process.The protocol we designed guarantees a deadlock-free committment by maintaining a strict global ordering between AND processes derived from their identifiers. More in detail, an OR process which has successfully evaluated its guard, sequentially sends a "reservation" to all its father AND processes following their global order, and subsequently a "confirmation"/"cancellation" to each reserved father AND process if the first has been

succeeded/failed. During the commit phase, an AND process waits for messages sent by its child OR processes. When a "reservation" message is received, the AND process suspends until a "confirmation"/"cancellation" from the same reserving process arrives. This implies that any other OR process sending a reservation message to it, will be blocked. When a "confirmation" message arrives, the other potential OR consumers of the reserved AND resource are killed. If a "cancellation" message arrives, the AND resource is made free and new requests are taken into account.

- Each clause of a ROSE program is represented by a single, independent process (the Manager) which could be a bottle-neck with respect to the messages sent by the AND processes. Notice that, however, that the main activity of the Manager is message routing, while unification is executed by dedicated unification processes.

- The incremental unification mechanism implies a very heavy message traffic across the network. In the scheme described above, we assumed that manager and intermediate nodes in the unification tree have to forward incoming messages to all children independently of the message structure. A more efficient routing of messages between processes in the tree has been obtained by forwarding messages only to those processes which can really consume them, i.e. that can unify. To this purpose, we enrich the descriptor structure for manager and unification processes, taking into account also the status of the head unifications for each child process.

- Another problem is related to the increasing load, due to the increasing number of processes that make unuseful computations. For example, when an OR process P commits, all its ancestors in the unification tree, except the manager, become unuseful since they will create only processes that consume at least one of the resources definitely assigned to P. In order to reduce this load, we choose to introduce a sort of "garbage collection" mechanism on the unification tree based on the forward propagation of kill messages from the manager towards the tip nodes.

- As regards to the allocation policy, in the first prototype of ROSE we have chosen to map all AND processes generated by an OR process P on the same physical node of P. Two reasons motivate this choice. First, the computational load of an OR process after the creation of its AND children is very light since it only waits for a result deriving from its successors. The second reason is due to the sharing of environment variables between the OR process and its children. If they are allocated on the same processor this is easily implementable without message passing.
  With reference to manager processes, our decision is to allocate them on separate nodes, whenever possible, together with all the unification processes of the tree. There is, in fact, a high coupling degree between processes in the same unification tree, while the computational load of each unification process is light. OR processes, instead, are allocated on separate nodes since they carry on the greater part of the computation that consists in the guard evaluation and the body code execution.
  We are aware that allocation is strictly related to the user program. For this reason, we plan to investigate the application of program analysis techniques such as abstract interpretation [Deb88] to obtain better allocation policies.

CONCLUSIONS

The parallel model of the logic programming language ROSE is suitable for a fully distributed implementation since communication does not rely on the sharing of logic variables. Communication

between processes is performed via multiple-headed clauses in a way similar to the "rendez-vous" model.
This new model of communication implies that the AND/OR process tree peculiar of parallel logic programming models becomes a graph. Moreover, the new concept of unification process tree is introduced to deal with unification of multiple-headed clauses in a very incremental and parallel way.
We have designed an abstract machine, based on the Warren Abstract Machine [War83], for the parallel execution of ROSE programs on non-shared memory machines.
The real implementation is obtained by mapping the ROSE abstract machine on a transputer-based architecture.
The first prototype, under development, has been built for testing the model rather than deeply discussing performance evaluations.
In the future we plan to execute significant bench-mark tests in order to better evaluate this implementation, the allocation strategies and compare performance results with STREM-based language implementations on distributed architectures (as the ones described in [Sha87]).
Moreover, we plan to implement the overall ROSE language taking into account also the sequential part and its integration with the parallel one.

Acknowledgments:

This work has been partially supported by the "Progetto Finalizzato Sistemi Informatici e calcolo Parallelo" of C.N.R. under grants n. 890004269 and n. 8900121.

REFERENCES:

[Bos90] P. G. Bosco, et al., "Logic and Functional Programming on Distibuted memory Architectures", Proc. 7th Int.l Conf. on Logic programming ICLP'90, Jerusalem, (D.H.D. Warren and Peter Szeredi eds), The MIT Press, pp. 325-339, 1990.

[Bro90] A. Brogi, "AND-Parallelism without Shared Variables", in Proc. Seventh International Conference on Logic Programming, Jerusalem, (D.H.D. Warren and Peter Szeredi Eds.), pp. 306-324, The MIT Press, 1990.

[Bro90a] A. Brogi, "Distributed Logic Programming in ROSE", Technical Report, Dipartimento di Informatica, Universita' di Pisa, 1990.

[Bro91] A.Brogi, A.Ciampolini, E.Lamma, P.Mello: "A Distributed Implementation for Parallel Logic Programming", to appear in Proceedings COMPEURO91, IEEE Computer Society Press, May 1991.

[Bur88] A. Burns, "Programming in *occam-2*", Addison-Wesley 1988.

[Con85] J.S. Conery, D.F.Kibler: "AND parallelism and non-determinism in logic programs", in New Generation Computing n.3, pp. 43-70, 1985.

[Con87] J.S.Conery: "Binding Environments for Parallel Logic Programs in Non-Shared Memory Multiprocessor", IEEE Symposium on Logic Programming, 1987.

[CST89] MEIKO, "CSTools: A Technical Overview", Meiko Technical Report S0205-12S, 1989.

[Deb88] S.K. Debray: "Static Analysis of Parallel Logic Programs", Proc. 5th International Conference on Logic Programming, Seattle (USA), The MIT Press, 1988.

[Fal84] M. Falaschi, G. Levi, C. Palamidessi, "A Synchronization Logic: Axiomatics and Formal Semantics of Generalized Horn Clauses", Information and Control, 60, 1-3, pp. 36-69 (1984).

[Her90a] M.V. Hermenegildo, F. Rossi, "Non-Strict Independent And-Parallelism", in Proc. 7th Int.l Conf. on Logic Programming, Jerusalem, (D.H.D. Warren and Peter Szeredi Eds.), pp. 237-252, The MIT Press, 1990.

[Her90b] M.V. Hermenegildo, K.J. Greene, "&-Prolog and its Performance: Exploiting Independent And-Parallelism", in Proc. 7th Int.l Conf. on Logic Programming, Jerusalem, (D.H.D. Warren and Peter Szeredi Eds.), pp. 253-268, The MIT Press, 1990.

[INM88] INMOS, "Transputer Reference Manual", Prentice Hall, 1988.

[Kal88] L.W.Kale, B.Ramkumar, W.Shu: "A Memory Organization Independent Binding Environment for AND and OR Parallel Execution of Logic Programs", Proc. 5th International Conference on Logic Programming, Seattle (USA), (R.A. Kowalski, K.A. Bowen eds.), The MIT Press, 1988.

[Mon84] L. Monteiro, "A proposal for distributed programming in logic", in *Implementations of Prolog* (J.A. Campbell ed.), Ellis Horwood, 1984.

[Mut90] K. Muthukumar, M.V. Hermenegildo, "The CDG, UDG and MEL Methods for Automatic Compile-time Parallelization of Logic Programs for Independent And-parallelism", in Proc. 7th Int.l Conf. on Logic Programming, Jerusalem, (D.H.D. Warren and Peter Szeredi Eds.), pp. 221-236, The MIT Press, 1990.

[Nak89] K.Nakajima: "Distributed Implementation of KL1 on the Multi-PSI/V2", Proc. 6th International Conference on Logic Programming, Lisbon (P), (G. Levi, M. Martelli eds.), The MIT Press, 1989.

[Nys88] S.O. Nystrom: "Control Structures for Guarded Horn Clauses", in Proc. 5th International Conference on Logic Programming, Seattle, (R.A.Kowalski, K.A.Bowen eds.), The MIT Press, 1988.

[Ram88] P.Raman, E.W.Stark: "Fully Distributed, AND/OR Parallel Execution of Logic Programs", in Proc. 5th International Conference on Logic Programming, Seattle, (R.A.Kowalski, K.A.Bowen Eds.), The MIT Press, 1988.

[Sha83] E. Y. Shapiro, "A Subset of Concurrent Prolog and its Interpreter", in *Concurrent Prolog: Collected Papers*, The MIT Press, Vol. 1, pp. 27-83, 1987.

[Sha87] E. Y. Shapiro (ed), "Concurrent Prolog: collected papers", The MIT Press, 1987.

[Sha89] E. Y. Shapiro, "The Family of Concurrent Logic Programming Languages", *ACM Computing Surveys*, 21:3, pp.412-510, 1989.

[Sze89] P. Szeredi, "Performance Analysis of the Aurora OR-parallel Prolog system", in Proc. 1st North-American Conf. on Logic Programming, Jerusalem, (E. Lusk and R. Overbeek Eds.), The MIT Press, 1989.

[Ued90] K.Ueda, M.Morita: "A New Implementation Technique for Flat GHC", Proc. Seventh International Conference on Logic Programming, Jerusalem, (D.H.D. Warren, Peter Szeredi eds.), The MIT Press, 1990.

[War83] D.H.D. Warren: "An Abstract Prolog Instruction Set", SRI Technical Note 309, SRI International, 1983.

Constraints for Synchronizing Coarse-grained Sequential Logic Processes

Antonio Brogi, Maurizio Gabbrielli, G. Levi

Dipartimento di Informatica, Università di Pisa
Corso Italia 40, 56125 Pisa, Italy
e-mail: {brogi,gabbri,levi}@dipisa.di.unipi.it

Abstract

We present a constraint based language designed to support the parallel execution of sequential logic theories. Each sequential computation is embedded in a (coarse-grained) process. According to the concurrent constraint paradigm, the parallel execution of processes is performed by using constraints for inter-process communication and synchronization. The operational semantics of the language is modelled according to the true concurrency approach.

1 Introduction

Recently, some efforts [2] have been devoted to realize concurrent systems where several logic theories (for instance Prolog programs) work in parallel by synchronizing via another shared theory. The granularity of processes (i.e. the theories) to be elaborated in parallel is coarse, mostly when compared with the fine-grained parallelism of concurrent logic languages [16]. Other proposals [1,9] define models for combining concurrent reactive and nondeterministic transformational languages, where a tight integration of don't know and don't care nondeterminism makes the computational model rather complex.

In this paper we propose a constraint based framework for a logic language which supports the parallel execution of sequential logic theories based on a smooth integration of the committed choice and the don't know nondeterminisms. Coarse-grained parallel processes synchronize and communicate via a shared set of constraints, as originally proposed in the concurrent constraint logic programming paradigm [14,15]. Differently from [14,15], each parallel process encapsulates a sequential theory, for example represented by a CLP (Constraint Logic Programming) [10] program. Several parallel processes synchronize via the shared state by checking the satisfiability of some constraints ("ask") in order to activate the theory, and by adding new constraints ("tell") resulting from the sequential derivation. The don't know nondeterminism is exploited within the sequential theories only and no distributed backtracking is needed. The operational semantics of the language can be modelled according to the true concurrent operational semantics approach [5,4].

A key issue is the neat separation between the sequential language and the synchronization primitives. The consequence of such a separation is twofold. On one hand, the

sequential language can be any particular instance of the CLP framework as well as any other language or constraint solving system. It can also be any other programming language, if equipped with a suitable external interface defined in terms of constraints. The communication mechanism can thus be used for integrating different programming languages within a well-defined concurrent framework. On the other hand, from the implementation point of view, the communication issues can be addressed by exploiting standard tools (such as imperative concurrent languages or operating system primitives). The proposed framework can also be viewed as a rational redefinition of other existing proposals of multi-threaded logic languages. The remaining part of the paper is organized as follows. In the next section the formal definition of the language is given. Section 3 is devoted to discuss programming examples. Section 4 defines the operational semantics according to the true concurrency approach. Finally section 5 contains a discussion on the relation of our framework to other existing proposals and on some implementation issues.

2 Formal Definition

In this section we formally define the language $\mathcal{L}(\mathcal{C}, \mathcal{S}_\mathcal{C})$, which is parametric w.r.t. a constraint system $\mathcal{C}$ and a sequential language $\mathcal{S}_\mathcal{C}$, which is based on $\mathcal{C}$ too. The language $\mathcal{L}(\mathcal{C}, \mathcal{S}_\mathcal{C})$ allows to define separate $\mathcal{S}_\mathcal{C}$ sequential computations, which are encapsulated into $\mathcal{L}(\mathcal{C}, \mathcal{S}_\mathcal{C})$ logic processes to be executed in parallel and which synchronize via a shared set $\mathcal{C}$ of constraints. Let us first introduce the definition of constraint system according to [15].

Definition 2.1 (Constraint system)
A first order constraint system *is a quadruple* $\mathcal{C} = \langle \Sigma, \mathcal{A}, Var, \Phi \rangle$ *where* Σ *is a many sorted vocabulary with associated set of sorts* S *and ranking function* ρ, $\mathcal{A}$ *is a* Σ*-structure,* Var *is an* S*-sorted set of variables and* Φ*, the set of* admissible constraints, *is some non-empty subset of* (Σ, Var)*-formulas, closed under conjunction.* □

As usual, the $\mathcal{A}$-valuation is a mapping from Var to the domain of $\mathcal{A}$. A constraint can be considered as the implicit definition of the possibly infinite set of its solutions, that is the valuations which satisfy the constraint. A widely used constraint system in logic programming is the Herbrand system which interprets the constraints on the free Σ-algebra.

The language $\mathcal{S}_\mathcal{C}$ can be any sequential language which is defined on the constraint system $\mathcal{C}$, that is any language whose semantics is defined by an input-output function on the domain of the constraint system $\mathcal{C}$. Therefore we give the following general definition.

Definition 2.2 ($\mathcal{S}_\mathcal{C}$)
Let $\mathcal{C} = \langle \Sigma, \mathcal{A}, Var, \Phi \rangle$ *be a constraint system where* $\Pi_c \subseteq \Sigma$ *is the set of constraint predicate symbols. Let* Π_{st} *be a set of many sorted predicate symbols with* $\Pi_{st} \cap \Pi_c = \emptyset$*. Then a sequential constraint language* $\mathcal{S}_\mathcal{C}$ *on* $\mathcal{C}, \Pi_{st}$ *is any language such that:*

1. *computations are activated by a pattern* $\langle [p_1(\tilde{X}_1), \ldots, p_h(\tilde{X}_h)], \sigma \rangle$ *where* $p_i \in \Pi_{st}$ *and* σ *is a set of constraints on* $\mathcal{C}$*,*
2. *the result of a computation is a pair* $\langle \Psi, \sigma' \rangle$ *where* $\Psi \in \{success, failure\}$ *is a termination mode and* σ' *is a set of constraints on* $\mathcal{C}$*.*

□

A CLP language [11,10], instantiated on the $\mathcal{C}$ domain, is definitely the most appropriate candidate for the $\mathcal{S}_\mathcal{C}$ language of sequential theories, since the constraint based sequential computation of CLP naturally fits into the concurrent constraint framework. Sequential computations can exploit the internal don't know nondeterminism (implemented by backtracking), which does not affect the (external) nondeterminism of the $\mathcal{L}(\mathcal{C}, \mathcal{S}_\mathcal{C})$ concurrent processes, ruled by the usual committed choice policy. According to the previous definition, CLP programs implementing sequential theories given an initial goal possibly terminate with a single answer (constraint). Multiple answers could be considered as well by allowing disjunction of constraints in the constraint system.

Even if it is the most appropriate, CLP is not the only kind of language that can be used in $\mathcal{L}(\mathcal{C}, \mathcal{S}_\mathcal{C})$ for the sequential part. Any sequential language whose input-output can be translated into a constraint form can be considered. Therefore we give the definition of $\mathcal{L}(\mathcal{C}, \mathcal{S}_\mathcal{C})$ parametrically w.r.t. a generic $\mathcal{S}_\mathcal{C}$ language.

A generic $\mathcal{L}(\mathcal{C}, \mathcal{S}_\mathcal{C})$ process $p(X)$ is defined by a set of clauses of the form

`p(X)` :- ask : tell → SeqGoal ; Tail

where *ask* and *tell* are constraints on $\mathcal{C}$ specifying the synchronization conditions for activating the sequential computations. *SeqGoal* is the initial goal for a $\mathcal{S}_\mathcal{C}$ sequential theory. *Tail* contains the set of processes which are created after the termination of the sequential computation. Let us now give the formal definition of $\mathcal{L}(\mathcal{C}, \mathcal{S}_\mathcal{C})$.

Definition 2.3 (Syntax of $\mathcal{L}(\mathcal{C}, \mathcal{S}_\mathcal{C})$)
Let $\mathcal{C} = \langle \Sigma, \mathcal{A}, Var, \Phi \rangle$ *be a constraint system, where* $\Pi_c \subseteq \Sigma$ *is the set of constraint predicate symbols. Let* Π_{st}, Π_{com} *be sets of many sorted predicate symbols (with their signatures) such that* $\Pi_{st} \cap \Pi_{com} = \emptyset$ *and* $(\Pi_{st} \cup \Pi_{com}) \cap \Pi_c = \emptyset$*. Let* $\mathcal{S}_\mathcal{C}$ *be any sequential language on* $\mathcal{C}, \Pi_{st}$*. Then the formal syntax of* $\mathcal{L}(\mathcal{C}, \mathcal{S}_\mathcal{C})$ *is the following.*

$$
\begin{array}{lll}
Clause & ::= & Head \ :- \ Agent \\
Head & ::= & p(\tilde{X}) \\
Agent & ::= & nil \mid \exists \tilde{X}.ask : tell \rightarrow SeqGoal; Tail \\
ask & ::= & \mathbf{c} \in \Phi \\
tell & ::= & \mathbf{c} \in \Phi \\
SeqGoal & ::= & nil \mid b(\tilde{X}) \mid b(\tilde{X}), SeqGoal \\
Tail & ::= & nil \mid Head \mid Tail \parallel Tail
\end{array}
$$

where $p \in \Pi_{com}$, $b \in \Pi_{st}$, $\rightarrow$ *is the commit operator,* $\parallel$ *is the AND-parallel conjunction, which is assumed to be commutative.* $\tilde{X}$ *is a subset of* Var *.* $\exists \tilde{X}$ *denotes the existential quantification of the (possibly empty) set of variables* $\tilde{X}$*.* Π_{com} *is the collection of symbols for communication predicates,* Π_{st} *is the (disjoint) set of sequential theories predicate symbols. A top-level goal specifies the initial configuration of a* $\mathcal{L}(\mathcal{C}, \mathcal{S}_\mathcal{C})$ *system and is defined as follows:*

$$
TopGoal \ ::= \ Tail :: \sigma
$$

where $\sigma \in \wp_f(\Phi)$*,* Φ *denotes the set of all constraints (definition 2.1) and* $\wp_f(\Phi)$ *denotes the set of the finite parts of* Φ*.* □

According to the previous definition, an agent represents a coarse-grained parallel process, which contains an *ask:tell* constraint, a sequential goal (*SeqGoal*) and a rewriting pattern (*Tail*) to activate other processes. Existential quantification is introduced to allow local variables. A commit operator ($\rightarrow$) follows the *ask:tell* guard.

A formal operational semantics for the $\mathcal{L}(\mathcal{C}, \mathcal{S}_\mathcal{C})$ language is given in section 4. Informally, a $\mathcal{L}(\mathcal{C}, \mathcal{S}_\mathcal{C})$ computation can be described as follows. According to the cc framework [15], at the communication processes level, the computation can be considered as the concurrent execution of agents which interact via a global set of constraints (named *store*) in order to synchronize and exchange the results of their computations. The basic actions performed by the agents, are asking for a constraint c being entailed by the store σ, and telling a constraint c to the store σ, if $\mathbf{c} \cup \sigma$ is consistent. These basic computation steps allow to continue, suspend or fail the computation of an agent, depending on the result of the ask:tell evaluation. Such an evaluation is specified as follows.

Evaluation of ask:tell
The evaluation of an ask constraint a in a store σ

- succeeds if $\mathcal{C} \models (\forall)\sigma \Rightarrow a$ (σ entails a);
- fails if $\mathcal{C} \models (\forall)\sigma \Rightarrow \neg a$ ($\sigma \wedge a$ is inconsistent);
- suspends if $\mathcal{C} \models (\exists)(\sigma \wedge a)$ and $\mathcal{C} \not\models (\forall)\sigma \Rightarrow a$ ($\sigma \wedge a$ is consistent but σ does not entail a).

$(\exists)(c)$ and $(\forall)$ are the existential and the universal closure of c respectively.
The evaluation of a tell constraint t in the store σ:

- succeeds if $\mathcal{C} \models (\exists)(\sigma \wedge t)$;
- fails if $\mathcal{C} \models (\forall)\sigma \Rightarrow \neg t$.

The evaluation of an $a : t$ (ask:tell) element is performed as an atomic action. It succeeds if the evaluations of both a and t succeed. It fails if the evaluation of either a or t fails and suspends otherwise. If $a : t$ succeeds, the store is updated, the new store being $\sigma' = \sigma \wedge t$. If the evaluation fails the store is unchanged.

The $\|$ operator has the usual meaning of parallel execution of agents, where the evaluation of $A_1 \parallel A_2$ succeeds iff the evaluation of both A_1 and A_2 succeeds, fails iff the evaluation of either A_1 or A_2 fails and suspends iff neither A_1 nor A_2 fail and A_i suspends, $i \in \{1, 2\}$. The nondeterminism arising from clause selection is handled according to the committed choice rule.

Rule of commitment
Given a procedure call $p(\tilde{X})$ in a store σ, in order to continue the computation a clause $p(\tilde{Y}) : -Agent$ is nondeterministically selected among those whose guard (i.e. $ask : tell$) evaluation in the store $\sigma \cup \{\tilde{X} = \tilde{Y}\}$ can be successfully performed. If the evaluation of the guard of every clause for p suspends, then the computation of $p(\tilde{X})$ suspends (and can possibly be resumed later, if the store will provide further informations for the successful evaluation of the ask).

After the successful execution of the $ask : tell$ constraint in an agent:

$$A = (ask : tell \rightarrow SeqGoal; Tail)$$

the sequential computation starts according to the given sequential goal $SeqGoal$. This computation is performed on a local environment of the sequential theory. Different sequential computations can be performed in parallel by different agents provided that there has been a synchronization with the store. If the sequential computation for $SeqGoal$ successfully terminates, its result is a constraint on local variables which has to be "projected" on the global variables determined by the initial activation pattern. The resulting constraint α is then added to the global store σ by an implicit tell operation. If either $\sigma \wedge \alpha$ is not consistent or the result of the sequential computation is a finite failure, the whole computation terminates with failure. Otherwise, if $\sigma \wedge \alpha$ is consistent (i.e. $\mathcal{C} \models (\exists)(\sigma \wedge \alpha)$) the store becomes $\sigma \wedge \alpha$ and the (parallel) computation of the elements of the $Tail$ is started. The element nil has the obvious operational meaning. A generic top-goal initial goal has the form $G :: \sigma$, where G is a $Tail$ element and σ is a store. The computation terminates with *failure* when the evaluation of an agent finitely fails.

3 Programming Examples

Before introducing the formal operational semantics of $\mathcal{L}(\mathcal{C}, \mathcal{S}_\mathcal{C})$, let us discuss some programming examples. The main motivation for the definition of $\mathcal{L}(\mathcal{C}, \mathcal{S}_\mathcal{C})$ is to provide a general framework where sequential processes work in parallel and synchronize by means of constraints. The degree of parallelism in a program execution strictly depends on the granularity of the sequential computations. If the sequential processes are coarse-grained then the degree of parallelism is high since synchronization steps are rather unfrequent. On the other hand, if the grain size of sequential processes decreases then the number of synchronizations grows.

One of the best known application areas of A.I. is the so called *distributed problem solving*. Generally speaking, the task of solving a problem on some domain is approached by means of several agents. Knowledge is partitioned into several distinct agents which cooperate in accomplishing a task. The agents can work in parallel and, in many cases, each agent contains detailed knowledge on a particular (sub) domain. Given a general problem, the process of problem solving consists of separately solving sub-problems in parallel as well as of coordinating parallel activities (synchronization). A particularly suitable programming example for $\mathcal{L}(\mathcal{C}, \mathcal{S}_\mathcal{C})$ is the problem of solving systems of constraints, which can be tackled by solving several subsystems of constraints in parallel by specialized constraint solvers.

A large class of applications for $\mathcal{L}(\mathcal{C}, \mathcal{S}_\mathcal{C})$ can be inspired by blackboard systems [6]. The language Shared Prolog [2] is an example of a concurrent logic language based on the blackboard model of problem solving. Relations between $\mathcal{L}(\mathcal{C}, \mathcal{S}_\mathcal{C})$ and Shared Prolog are discussed in section 5. Here, the definition of a blackboard system given in [2] is adapted to $\mathcal{L}(\mathcal{C}, \mathcal{S}_\mathcal{C})$.

In the blackboard model of problem solving, knowledge is divided into separate agents which work in parallel communicating (and synchronizing) via a shared blackboard. The blackboard is a shared data structure containing the current state of the solution. A special agent, called *control*, monitors the changes on the blackboard and decides which agent to start. Let us consider an example from a general class of blackboard systems where the blackboard data structure has a hierarchical structure. In particular, we discuss a problem

of *data fusion*, originally presented in [3] and then programmed in Shared Prolog [2]. The problem of data fusion consists of merging data coming from different sensors and other information sources. Data are organized into a hierarchical structure, where input data belong to the lowest level of the hierarchy. Each agent inputs data at one level of the hierarchy, elaborates them, and produces higher level data as results.

Let us consider a simplified version of the system described in [3], which elaborates data coming from different sensors (e.g. radars) and tries to identify the objects moving in a given region of space. Suppose that the initial *TopGoal* of the system activates three processes in parallel:

```
data_acquisition(Plots, Rcs) ||
rcs_filter(Rcs, Rcsval, Status) ||
plot_filter(Plots, Altitude, Speed, Status')
```

The *data acquisition* process collects packets of plots coming from several sensors and packets of radar cross sections (rcs). The technical meaning of data is rather irrelevant here (plots denote tracks of detected objects and so on). The task of the *rcs_filter* process is to input a packet of rcs data (*Rcs*), to filter out measurements errors and to produce filtered rcs data (*Rcsval*). When the *rcs_filter* process is initialized (i.e. during its first activation), it dynamically activates a new process *rcs_expert*. Such a process will elaborate the data produced by the *rcs_filter* process and produce, in turn, higher level data. For simplicity of notation, in this example all the variables occurring in the *ask* part of a clause but not in the *head* are intended to be existentially quantified. Moreover, the *ask* constraints should be repated in the *tell* part, too.

```
rcs_filter(Rcs, Rcsval, Status) :-
  Rcs=R.NextR, Rcsval=RV.NextRV, Status=initializing :
  Status'=initialized →
  rcs_filtering_procedure(R, RV);
  rcs_filter(NextR, NextRV, Status') ||
  rcs_expert(Rcsval, Rcs_hyp, Status)

rcs_filter(Rcs, Rcsval, Status) :-
  Rcs=R.NextR, Rcsval=RV.NextRV, Status=initialized :
  true : →
  rcs_filtering_procedure(R, RV);
  rcs_filter(NextR, NextRV, Status)
```

A committed-choice behaviour rules the selection of which definition of*rcs_filter* to choose depending on the internal status of the process itself.
The *plot_filter* process after filtering out measurement errors, derives data from the plots such as *Altitude* and *Speed*. The definition of the process consists of two alternative definitions as in the previous case.

```
plot_filter(Plots, Altitude, Speed, Status) :-
  Plots=P.NextP, Altitude=A.NextA, Speed=S.NextS, Status=initializing:
  Status'=initialized →
  filtering_procedure(P, A, S);
```



```
  plot_filter(NextP, NextA, NextS, Status') ||
  altitude_expert(Altitude, Altitude_hyp) ||
  speed_expert(Speed, Speed_hyp)

plot_filter(Plots, Altitude, Speed, Status) :-
  Plots=P.NextP, Altitude=A.NextA, Speed=S.NextS, Status=initialized:
  true →
  filtering_procedure(P, A, S);
  plot_filter(NextP, NextA, NextS)
```

Notice that the filtering procedures are standard Prolog or CLP programs, while in real-time systems they might more realistically be C routines [3]. Since the task of the system is to dynamically monitor a region of space, packets of data continuously arrive and have to be elaborated. Correspondingly, the processes recursively activate themselves on streams of data at the end of each cycle. Other processes working at higher levels of abstraction have a similar structure. The *speed_expert* process, for instance, given the detected speed of an object, draws an hypothesis on the type of the object. Different knowledge bases and different selection criteria are used depending on the value of the speed.

```
speed_expert(Speed, Speed_hyp) :-
  Speed=S.NextS, S ≤ 40:
  Speed_hyp=Sh.NextSh →
  search_slow_obj(S, Sh);
  speed_expert(NextS, NextSh)

speed_expert(Speed, Speed_hyp) :-
  Speed=S.NextS, S > 40:
  Speed_hyp=Sh.NextSh →
  search_fast_obj(S, Sh);
  speed_expert(NextS, NextSh)
```

Another case of systems where coarse-grained processes synchronize are operating systems. A very common situation is to have a manager of some resources (e.g. files, printers, databases) and several processes running in parallel and sharing those resources. Accesses to shared resources are ruled by mutual exclusion protocols. The problem of solving mutual exclusion protocols in concurrent logic programming has been addressed by many authors (see for instance [16]) and can be rephrased according to the constraints paradigm (on the line of [14]). The major difference w.r.t. concurrent logic languages is that after a process has worked in mutual exclusion from others, it can perform a *purely* sequential computation.

4 Operational Semantics

In this section we define an operational semantics for the language. The first purpose is to provide a formal description of the operational behavior. The second aim is to discuss the properties of the language from the viewpoint of distribution. This is the reason why we have decided to define the operational semantics following the true concurrency approach,

by using the constructive technique presented in [5], which extends the Plotkin's Structured Operational Style [12] to deal with distributed states. It would be also possible to define a fixpoint semantics based on an immediate consequence operator by generalizing the techniques used in [7] and [8]. Such a construction, however, would be based on interleaving and would not properly outline the above mentioned distribution properties.

According to [5], a $\mathcal{L}(\mathcal{C}, \mathcal{S}_\mathcal{C})$ system is decomposed into a set of configurations representing processes, possibly located in different places, which cooperate in accomplishing a task. A set of configurations represents the distributed state of the system. The behavior of a system is defined by a set of rewriting rules to be applied to sets of configurations to get computations. A rewriting rule specifies how a subset of the configurations composing a distributed state may evolve independently. A computation can be described by a sequence of distributed states (sets of configurations) and of rewriting rules, representing the evolution of system sub-parts.

Rewriting rules are adopted to describe the synchronization of parallel processes. The sequential $\mathcal{S}_\mathcal{C}$ derivations encapsulated into processes have to be modelled as well. To this purpose, we assume that a transition system describing $\mathcal{S}_\mathcal{C}$ computations is defined. Then, rewriting rules possibly resort to the transition system for $\mathcal{S}_\mathcal{C}$ when dealing with sequential computations. In other words, this corresponds to describing the semantics of $\mathcal{L}(\mathcal{C}, \mathcal{S}_\mathcal{C})$ by means of a hierarchy of two abstract machines (as in [2]). The lower level machine, defined by a transition system, describes the evolution of sequential computations. The higher level machine, defined by a set of rewriting rules, describes the behavior of $\mathcal{L}(\mathcal{C}, \mathcal{S}_\mathcal{C})$ programs.

Each sequential computation consists of the execution of a $\mathcal{S}_\mathcal{C}$ program for a given sequential goal $SeqGoal$. The derivation of $SeqGoal$ can be either successful or failing. Given a store σ, the possible computations of $SeqGoal$ can be described by the two following transitions, according to 2.2:

$$\langle SeqGoal, \sigma\rangle \mapsto_{\mathcal{S}_\mathcal{C}} \langle success, \sigma'\rangle$$

$$\langle SeqGoal, \sigma\rangle \mapsto_{\mathcal{S}_\mathcal{C}} \langle failure, \sigma'\rangle$$

The first one means that given a store σ, $SeqGoal$ can be proved by computing the new store σ'. The second transition models the finite failure of the computation.

We are mostly interested in modelling the aspects of concurrency, so that the definition of the transition system for $\mathcal{S}_\mathcal{C}$ is omitted here. Notice, however, that it can be easily defined (once $\mathcal{S}_\mathcal{C}$ has been suitably specified) as it has been done in [13,4] for the Prolog language.

Given that we know how to model sequential derivations, we have to define how a system composed of a set of concurrent agents and a store behaves. A set of rewriting rules is introduced to describe how sets of configurations can evolve. The set of configurations for the $\mathcal{L}(\mathcal{C}, \mathcal{S}_\mathcal{C})$ language must describe all the possible states of the processes, that is of the store and of the agents. In order to describe a store σ, the configuration $\langle\sigma\rangle$ is adopted, so that a store is denoted by the set of constraints it contains. During a computation, an agent can be in one of the three following states:

1. waiting for starting a sequential computation

2. performing a sequential computation

3. waiting for adding to the store the results of its terminated sequential computation.

The first state is denoted by a configuration of the form $\langle Head\rangle$ where $Head$ is the name of the process waiting for synchronizing with the store. An agent which is currently performing a sequential computation is denoted by the pair $\langle(SeqGoal; Tail), \sigma\rangle$ where $SeqGoal$ is the sequential goal currently being executed, $Tail$ is the continuation of the process and σ is the local store of the process. Finally, an agent which is waiting for telling to the store the result of its succesfully terminated computation is denoted by the pair $\langle Tail, \sigma\rangle$, where $Tail$ is the tail recursion and σ is the local store, as updated by the sequential computation. Moreover, the special configuration $\langle failure\rangle$ represents a failure.

Definition 4.1 (Configurations)
The set of configurations is defined as follows:

$$\Gamma = \{\langle\sigma\rangle, \langle Head\rangle, \langle(SeqGoal; Tail), \sigma\rangle, \langle Tail, \sigma\rangle, \langle failure\rangle\}$$

where Head, SeqGoal and Tail are defined according to definition 2.3. Configurations are ranged over by γ. □

A set of configurations can syntactically be obtained from the definition of a $\mathcal{L}(\mathcal{C}, \mathcal{S}_\mathcal{C})$ goal.

Definition 4.2 (From $\mathcal{L}(\mathcal{C}, \mathcal{S}_\mathcal{C})$ goals to sets of configurations)
Let Θ *be the set of all possible* TopGoal *and* Tail *(definition 2.3). Then the function* $dec : \Theta \rightarrow 2^\Gamma$ *is defined by induction on the syntax of* $\mathcal{L}(\mathcal{C}, \mathcal{S}_\mathcal{C})$ *as follows:*

$$\begin{array}{lcl} dec(Tail :: \sigma) & = & dec(Tail) \cup \{\langle\sigma\rangle\} \\ dec(Tail \parallel Tail') & = & dec(Tail) \cup dec(Tail') \\ dec(Head) & = & \{\langle Head\rangle\} \\ dec(nil) & = & \emptyset \end{array}$$

□

Definition 4.3 (Evaluation of ask:tell)
Let Φ *the set of all constraints (definition 2.1) and let* Δ *the set of all the ask:tell elements. Then the function* $\nu : \wp_f(\Phi) \times \Delta \rightarrow \wp_f(\Phi)$, *where* $\wp_f(\Phi)$ *denotes the set of the finite parts of* Φ, *defines the ask : tell evaluation rule as follows:*

$$\nu(\sigma, \exists\tilde{Y}.ask : tell) \begin{cases} \sigma \wedge tell & if\ C \models (\forall)\sigma \Rightarrow \exists\tilde{Y}.ask\ and\ C \models (\exists)(\sigma \wedge ask \wedge tell) \\ fail & if\ C \models (\forall)\sigma \Rightarrow (\neg\exists\tilde{Y}.ask \wedge tell) \end{cases}$$

The function $\rho : \wp_f(\Phi \times SeqGoal) \rightarrow \wp_f(\Phi)$, *given a store* σ *and a SeqGoal, returns a restriction of* σ *which depends on SeqGoal.* □

In the previous definition, a specific definition of ρ depends on the specific sequential language. For example, $\rho(\sigma, SeqGoal)$ can be the restriction of σ to the variables in $SeqGoal$. More generally (by slightly modifying the syntax) the goal $SeqGoal$ can contain an explicit information on the restriction to be operated on σ. Let us now give the set of rewriting rules describing $\mathcal{L}(\mathcal{C}, \mathcal{S}_\mathcal{C})$.

Definition 4.4 (Rewriting Rules)
Let P be a $\mathcal{L}(\mathcal{C},\mathcal{S}_\mathcal{C})$ program, the $\mathcal{L}(\mathcal{C},\mathcal{S}_\mathcal{C})$ derivation relation $\Rightarrow_P$ is defined as the least relation satisfying the following rewriting rules.

(1) $$\frac{\exists\,(p(\tilde{X}) : -\exists Z.ask : tell \rightarrow SeqGoal; Tail) \in P \text{ s.t. } \nu(\sigma \wedge (\tilde{X} = \tilde{Y}), \exists Z.ask : tell) = \sigma'}{\{\langle\sigma\rangle\} \cup \{\langle p(\tilde{Y})\rangle\} \Rightarrow_P \{\langle\sigma'\rangle\} \cup \{\langle (SeqGoal; Tail), \sigma'\rangle\}}$$

(2) $$\frac{\langle SeqGoal, \sigma\rangle \mapsto_{\mathcal{S}_\mathcal{C}} \langle success, \sigma'\rangle}{\{\langle (SeqGoal; Tail), \sigma\rangle\} \Rightarrow_P \{\langle Tail, \rho(\sigma', SeqGoal),\rangle\}}$$

(3) $$\frac{\langle SeqGoal, \sigma\rangle \mapsto_{\mathcal{S}_\mathcal{C}} \langle failure, \sigma'\rangle}{\{\langle (SeqGoal; Tail), \sigma\rangle\} \Rightarrow_P \{\langle failure\rangle\}}$$

(4) $$\frac{\nu(\sigma, \{\} : \sigma') = \langle\sigma''\rangle}{\{\langle\sigma\rangle\} \cup \{\langle Tail, \sigma'\rangle\} \Rightarrow_P \{\langle\sigma''\rangle\} \cup dec(Tail)}$$

(5) $$\frac{\nu(\sigma, \{\} : \sigma') = fail}{\{\langle\sigma\rangle\} \cup \{\langle Tail, \sigma'\rangle\} \Rightarrow_P \{\langle failure\rangle\}}$$

(6) $$\frac{}{\{\gamma\} \cup \{\langle failure\rangle\} \Rightarrow_P \{\langle failure\rangle\}}$$

□

Remarks
As usual, we leave the parameter P implicit in the definition of the $\Rightarrow_P$ relation since the program does never change during the derivation. Rule 1 defines the activation of a process $p(\tilde{X})$. Rules 2 and 3 define the derivation of the result of a sequential computation, both resorting to the transition system for $\mathcal{S}_\mathcal{C}$. Rules 4 and 5 correspond to the *tell* operation of the local store which has been computed by a successful sequential computation. The tail recursion is defined by resorting to the definition of *dec* (4.2). Rule 6 makes every configuration evaporating in presence of a failure. □

Let us now define what a computation is, by following [4].

Definition 4.5 (Computation)
Let P be a $\mathcal{L}(\mathcal{C},\mathcal{S}_\mathcal{C})$ program, and G *be a* TopGoal. *The sequence of states and rewriting rules:*

$$\xi = S_0\ R_0\ S_1\ R_1\ \ldots$$

is a computation *iff:*

- $S_0 = dec(G)$
- $R_j = \{I_{1,j} \Rightarrow_P I'_{1,j}, \ldots, I_{n,j} \Rightarrow_P I'_{n,j}\}$ *such that:*
  - $I_{i,j} \Rightarrow_P I'_{i,j}$ *belongs to the* $\Rightarrow_P$ *derivation relation* $\forall i = 1, \ldots, n$
  - $I_{1,j} \oplus \ldots \oplus I_{n,j} \subseteq S_j$
  - $S_{j+1} = (S_j \backslash (I_{1,j} \oplus \ldots \oplus I_{n,j})) \oplus I'_{1,j} \oplus \ldots \oplus I'_{n,j}$

where $\oplus$ denotes the union of multisets.
A successful computation *is a finite sequence:*

$$\xi = S_0\ R_0\ S_1\ R_1\ \ldots\ R_{n-1}\ S_n$$

such that $S_n = \{\langle\sigma\rangle\}$ where σ is a store. The result of the computation is $\rho(\sigma, G)$, that is the projection of the store on the initial goal (definition 4.3).
Analogously, a failed computation *is a finite sequence as above where: $S_n = \{\langle failure\rangle\}$* □

Remarks
A computation is modelled by a (possibly) infinite sequence of states and rewriting rules. At each step of the computation, the configuration denoting the global store represents the current global state of the computation. Each sequential computation which succeeds or finitely fails is denoted by one transition only (rule 2 and 3 of definition 4.4). Infinite sequential computations are interpreted as a local deadlock of the corresponding process, which is denoted by a configuration of the form $\langle (SeqGoal; Tail), \sigma\rangle$. Local deadlocks do not cause global deadlocks. A failure can arise either because a sequential computation finitely fails (rule 3) or because the store which has been locally computed by a sequential derivation is inconsistent with the global one (rule 5). According to rule 6, failures are propagated to all the other components of the system. Notice however, that other choices would have been possible, for instance a finitely failing sequential computation may be interpreted by the corresponding process remaining stuck. □

A true concurrent operational semantics shows the degree of (independent) parallelism which is possible in a given language, that is the number of transitions which can concurrently take place. In the above definition this happens for all the transitions related to the execution of the (sequential) components of different processes. The concurrent constraint framework is clearly a special case, where there are no sequential goals. In such a case, the above defined operational semantics shows that one transition only is possible at any given time. In other words, the true concurrent semantics would be very similar to an interleaving semantics. The relevant issue is therefore exactly in the internal (sequential) structure of processes. Note also that the number of transitions is controlled by the fact that only one process can communicate with the store. In other words, the unique shared set of constraints is a bound to concurrency. This can be improved, by splitting the store into a collection of separate stores, as already suggested in the case of Shared Prolog [2].

5 Comparisons with other Concurrent Languages and Implementation Issues

It is mandatory to compare $\mathcal{L}(\mathcal{C},\mathcal{S}_\mathcal{C})$ with the family of concurrent logic languages [16] and, in particular, with concurrent constraint programming [14]. The main difference concerns the coarseness of the processes involved in a computation. In the other concurrent logic languages, processes are very fine-grained, that is they correspond to atomic goals. In our framework, AND-parallel processes are intended to be coarse-grained. The degree of parallelism derives from the size of sequential derivations. Indeed, since each parallel process encapsulates a sequential computation, if the sequential computations are short, the granularity of the processes is finer and the degree of parallelism decreases because

of the many interactions with the store. The neat separation between the deduction and synchronization steps in $\mathcal{L}(\mathcal{C}, \mathcal{S}_\mathcal{C})$ is reflected also by the operational semantics. In fact, the operational semantics of $\mathcal{L}(\mathcal{C}, \mathcal{S}_\mathcal{C})$ can naturally be described by means of a distributed model (see section 4), while interleaving models only have been defined so far for the family of concurrent logic languages.

Another related work is Shared Prolog (SP) [2]. An SP system is composed of a set of parallel agents (theories) which are Prolog programs extended by communication patterns ruling the communication and the synchronization via a shared blackboard. The blackboard is a set of unit clauses (facts). The synchronization of the agents with the blackboard consists of adding and/or deleting facts to/from the blackboard. The computational model of SP is similar to that of $\mathcal{L}(\mathcal{C}, \mathcal{S}_\mathcal{C})$. However, there are some major differences to be noticed. First, the synchronization model of $\mathcal{L}(\mathcal{C}, \mathcal{S}_\mathcal{C})$ is based on constraints rather than on assert and retract operations. Moreover, the dynamic evolution of the shared store of constraints is *monotonic* while the shared blackboard of SP is intrinsically *nonmonotonic* as long as retract operations are permitted. This difference is reflected by the underlying semantic model, when compositionality is taken into account.

Let us now briefly discuss some issues related to the language implementation. The dynamic configuration of a $\mathcal{L}(\mathcal{C}, \mathcal{S}_\mathcal{C})$ system consists of a parallel composition of a set of sequential processes and a store. Each sequential process can virtually be associated to a different processor capable of supporting the execution of $\mathcal{S}_\mathcal{C}$ programs. Another process is in charge of managing the store. Actually, it will be required to support the standard basic functions of any constraint system, that is to embed some evaluation function working on constraints.

In addition to that, there are communication primitives supporting the exchange of messages between the various processes. Notice that such primitives can be defined as standard inter-process primitives, not being affected by the particular nature of the $\mathcal{L}(\mathcal{C}, \mathcal{S}_\mathcal{C})$ language. There are three possible types of messages:

1. messages containing *ask* and *tell* constraints sent by some sequential process to the store manager as a request for starting an internal derivation,
2. messages sent by the store manager to a sequential process containing the authorization to start an internal derivation,
3. messages sent by a sequential process to the store manager.

It is worth noting that messages of type 2 contain - at least virtually - a copy of the current shared store to be passed to a sequential process. Indeed, in a real implementation it is not necessary to copy the whole store. It is sufficient that messages of type 2 contain only a copy of a subset of the store, roughly a suitable projection of the store on the variables occurring in the $Head$ of the agent.

Finally, the tail recursion of a process possibly generates several new processes. Correspondingly, the implementation has to provide some primitives for the dynamic creation of processes.

References

[1] R. Bahgat and S. Gregory. Pandora: Non-deterministic Parallel Logic Programming. In G. Levi and M. Martelli, editors, *Proc. Sixth Int'l Conf. on Logic Programming*, pages 471–486. The MIT Press, Cambridge, Mass., 1989.

[2] A. Brogi and P. Ciancarini. The concurrent language shared prolog. *ACM Transactions on Programming Languages and Systems*, 1(1), 1991. To appear.

[3] A. Brogi, R. Filippi, M. Gaspari, and F. Turini. An Expert System for Data Fusion based on a Blackboard Architecture. In *Proc. of Eight Int'l Workshop on Expert Systems and their Applications*, pages 147–166, Avignon, 1988.

[4] A. Brogi and R. Gorrieri. A Distributed, Net Oriented Semantics for Delta Prolog. In J. Diaz and F. Orejas, editors, *Proc. of TAPSOFT-CAAP'89, LNCS 351*, pages 637–654. The MIT Press, Cambridge, Mass., 1989.

[5] P. Degano and U. Montanari. Concurrent Histories: a Basis for Observing Distributed Systems. *Journal of Computer and System Science*, 34:442–461, 1987.

[6] R. Engelmore and T. Morgan. *Blackboard Systems.* Addison-Wesley, 1988.

[7] M. Gabbrielli and G. Levi. Unfolding and Fixpoint Semantics of Concurrent Constraint Programs. In H. Kirchner and W. Wechler, editors, *Proc. Second Int'l Conf. on Algebraic and Logic Programming*, volume 463 of *Lecture Notes in Computer Science*, pages 204–216. Springer-Verlag, Berlin, 1990.

[8] M. Gabbrielli and G. Levi. Modeling answer constraints in Constraint Logic Programs. In K. Furukawa, editor, *Proc. Eighth Int'l Conf. on Logic Programming.* The MIT Press, Cambridge, Mass., 1991. To appear.

[9] S. Haridi and S. Janson. Kernel Andorra Prolog and its computation model. In D.H.D. Warren and P. Szeredi, editors, *Proc. Seventh Int'l Conf. on Logic Programming*, pages 31–48. The MIT Press, Cambridge, Mass., 1990.

[10] J. Jaffar and J.-L. Lassez. Constraint Logic Programming. In *Proc. Fourteenth Annual ACM Symp. on Principles of Programming Languages*, pages 111–119. ACM, 1987.

[11] J. Jaffar and J.-L. Lassez. Constraint Logic Programming. Technical report, Department of Computer Science, Monash University, June 1986.

[12] G. Plotkin. A structured approach to operational semantics. Technical Report DAIMI FN-19, Computer Science Department, Aarhus University, 1981.

[13] V.A. Saraswat. GHC: operational semantics, problems and relationship with CP($\downarrow, |$). In *IEEE Int'l Symp. on Logic Programming*, pages 347–358. IEEE, 1987.

[14] V.A. Saraswat. *Concurrent Constraint Programming Languages.* PhD thesis, Carnegie-Mellon University, January 1989.

[15] V.A. Saraswat and M. Rinard. Concurrent constraint programming. In *Proc. Seventeenth Annual ACM Symp. on Principles of Programming Languages.* ACM, 1990.

[16] E.Y. Shapiro. The family of concurrent logic programming languages. *ACM Computing Surveys*, 21(3):412–510, 1989.