A Logic Coordination Language based on the Chemical Metaphor

Paolo Ciancarini and Daniela Fogli and Mauro Gaspari

Technical Report UBLCS-96-12

July 1996

Department of Computer Science
University of Bologna
Piazza di Porta S. Donato, 5
40127 Bologna (Italy)
The University of Bologna Department of Computer Science Research Technical Reports are available in gzipped PostScript format via anonymous FTP from the area ftp.cs.unibo.it:/pub/TR/UBLCS or via WWW at URL http://www.cs.unibo.it/. Plain-text abstracts organized by year are available in the directory ABSTRACTS. All local authors can be reached via e-mail at the address last-name@cs.unibo.it. Questions and comments should be addressed to tr-admin@cs.unibo.it.

Recent Titles from the UBLCS Technical Report Series

94-21 Parallel Independent Grid Files Based on a Dynamic Declustering Method Using Multiple Error Correcting Codes, P. Ciaccia, November 1994.
95-13 Optimal Multi-Block Read Schedules for Partitioned Signature Files, P. Ciaccia, August 1995.
96-5 The Shape of Shade: a Coordination System, C. Castellani, P. Ciancarini, D. Rossi, March 1996.
96-7 Using Bayesian Belief Networks for the Automated Assessment of Students’ Knowledge of Geometry Problem Solving Procedures, M. Roccetti, P. Salomoni, March 1996.
96-9 Towards an Algebra of Actors, M. Gaspari, April 1996.
96-10 Mobile Petri Nets, A. Asperti, N. Busi, May 1996.
96-12 A Logic Coordination Language Based on the Chemical Metaphor, P. Ciancarini, D. Fogli, M. Gaspari, July 1996.
A Logic Coordination Language based on the Chemical Metaphor

Paolo Ciancarini and Daniela Fogli and Mauro Gaspari

Technical Report UBLCS-96-12

July 1996

Abstract

We describe Gammalog, a logic language whose semantics is based on the chemical metaphor typical of concurrent multiset rewriting. In fact, the language combines the ability of describing coordination by multiset transformation rules, as in GAMMA, with the execution model of logic programming in a strongly typed framework, as in Gödel. We present the design choices, the language syntax, and its formal semantics. Gammalog has a standard model theoretic semantics expressed in terms of multiset rewritings; soundness and completeness results are proved with respect to extended SLD resolution. We present a prototype implementation of a compiler based on the original Gödel implementation, which has been extended to support first class multisets. As an example of the expressive power of the language we provide a specification of a simple operating system.
1 Introduction

A distributed programming paradigm based on the concepts of generative coordination and communication in a shared dataspace has recently become popular [15]. Generative coordination and communication means that processes use no channel to communicate: they simply output tuples in a shared dataspace; processes which need input access the tuples associatively. Tuples are persistent, that is, messages survive to processes which originated them. Processes coordinate themselves uniquely using a very small set of primitives which access the shared dataspace. Some languages have been proposed which follow such a paradigm: for instance, Linda [5], and GAMMA [3].

The shared dataspace coordination model has been investigated in logic programming as well: Shared Prolog [4], μlog [20], and LO [1] are examples of logic languages based on shared dataspaces. Like the stream-based ones, these logic languages have no standard model-theoretic semantics, even if some of them offer a non standard declarative semantics. For instance, the model-theoretic semantics of μlog [20] is based on a notion of truth with respect to traces (i.e., truth depends on the sequence of communication events which may occur during the computation).

In this paper we present Gammalog, a logic language for which we provide a standard model theoretic semantics expressed in terms of multiset rewritings. We describe the language semantics and an implementation based on Gödel [19].

The coordination abstractions provided by GAMMA are thus made available in a logic programming computational framework. Another advantage of Gammalog is that it provides an executable version of GAMMA where coordination issues are separated from computation issues [5]. This is obtained by embedding GAMMA in Gödel, and distinguishing between coordination activities, expressed by multiset-rewriting rules, and sequential activities, expressed by (sequential) logic programs.

The paper is organized as follows: in Section 2 we shortly introduce GAMMA and discuss its features with respect to coordination; in Section 3 we introduce Gammalog, a logic programming paradigm based on multiset rewriting. In Section 4 we study the formal semantics of Gammalog and we propose a parallel operational semantics well suited for coordination. In Section 5 we describe the language Gammalog as an instance of Gammalog based on Gödel. In Section 6 we describe a prototype implementation, whereas in order to illustrate the expressive power of the language in Section 7 we show the specification of an operating system kernel. Finally, the last sections compare Gammalog to concurrent logic languages and describe the future developments of such a research.

2 The GAMMA coordination model

GAMMA program specifications can be described in terms of multiset transformations without introducing unnecessary sequentiality [3]. For instance, if we have to compute the maximum element of a set we could describe an abstract algorithm as follows:

\begin{verbatim}
while there are at least two elements in the set
  select two elements of the set, compare them and
  remove the smaller one
\end{verbatim}

The advantage of such a definition is that it does not fix the order of evaluation of the comparisons; thus, if the set contains \( n \) elements, and if \( n/2 \) processors are available, then \( n/2 \) comparisons and the subsequent replacements can be performed in parallel.

The basic data structure used in Gamma is the multiset. A multiset is just a bag containing items which are stored without any constraint or relationship among one another. The control structure associated with multisets is the \( \Gamma \) operator; its formal definition can be stated as follows [3]:

\[
\Gamma ((R, A))(M) = \\
\text{if } \forall x_1, \ldots, x_n \in M, \sim R(x_1, \ldots, x_n)
\]
then \( M \)
else let \( x_1, \ldots, x_n \in M \) be such that \( R(x_1, \ldots, x_n) \) in
\[
\Gamma((R, A))((M - \{ x_1, \ldots, x_n \}) + A(x_1, \ldots, x_n)).
\]
where \((R, A)\) is a pair of functions specifying the rewriting rule which can be applied on the multiset. \( R \) is a reaction condition, namely a boolean function which specifies if the rule is applicable. \( A \) is an action, namely a multiset rewriting, executed when the reaction succeeds. Operationally the \( \Gamma \) operator searches for a subset of \( M \), \( \{ x_1, \ldots, x_n \} \), such that \( R(x_1, \ldots, x_n) \) holds. When the reaction succeeds the elements satisfying it are removed from the multiset and the action \( A(x_1, \ldots, x_n) \) generates new elements to be inserted in the multiset. Otherwise, if no elements of \( M \) satisfy the reaction condition \(( \forall x_1, \ldots, x_n \in M, ~ R(x_1, \ldots, x_n) \) the \( \Gamma \) operator terminates and the result is \( M \).

What follows is a Gamma program which calculates the maximum element of a set.

\[
\text{max\_element}(s) = \Gamma((R, A))(s) \text{ where }
\]
\[
\begin{align*}
R(x, y) &= x \leq y \\
A(x, y) &= \{ y \}.
\end{align*}
\]

Such a program consists of one reaction only, which takes two numbers and discards the lower one.

The original GAMMA was based on a simple chemical metaphor of multiset transformation. Elements in the multiset react as chemical molecules originating new molecules according to the multiset transformation operator. Subsequently GAMMA was augmented with two operators to combine simple programs into complex ones in a “Gamma calculus” [17]. The operators are the sequential composition \( P \circ Q \) and the parallel composition \( P + Q \). For readability reasons in the rest of this paper we assume that the sequential composition operator is interpreted left to right, i.e. the program \( Q \) is executed with the multiset returned by \( P \) as an input only when the program \( P \) is terminated.\(^3\)

The sequential and the parallel operators build complex programs starting from simple Gamma programs. For instance, the program \( \text{Positive\_Integers} \) which computes the number of positive integers in a multiset can be expressed as the composition of three simple programs, as follows:

\[
\text{Positive\_Integers}(m) = (\text{Ones} + \text{Non\_neg}) \circ \text{Add}
\]

where \( \text{Add} \) is a program which returns the sum of elements of a multiset; it can be written as follows:

\[
\text{Add} = \Gamma((R, A))(s) \text{ where }
\]
\[
\begin{align*}
R(x, y) &= \text{true} \\
A(x, y) &= \{ x + y \}
\end{align*}
\]

\( \text{Ones} \) is the program which transforms to 1 all the positive integers in a multiset:

\[
\text{Ones} = \Gamma((R, A))(s) \text{ where }
\]
\[
\begin{align*}
R(x) &= x > 1 \\
A(x) &= \{ 1 \}
\end{align*}
\]

\( \text{Non\_neg} \) is a program which selects all the positive numbers of a multiset of integers. It can be defined in Gamma as follows:

\[
\text{Non\_neg} = \Gamma((R, A))(s) \text{ where }
\]
\[
\begin{align*}
R(x) &= x < 0 \\
A(x) &= \{ \}
\end{align*}
\]

This means that to compute the number of positive values in a multiset we can execute the first two Gamma programs in parallel on a shared multiset and when the two programs terminate we compute the number of ones in the multiset (using “Add”).

The advantage of building programs which exploit these operators is that they allow one to design programs in a structured way, and provide a way to prove properties: for instance confluence and termination [17].

\(^3\) Instead, in [17] the result of \( P \circ Q \) is obtained by executing the programs from right to left.
Gamma can be also used to specify reactive systems. In this case the aim of the multiset rewriting rules is not to express computation as in the previous example but to express coordination.

As an example, consider the program in Figure 1 describing a system where \( n \) users share a common pool of resources [3]. Each user can be in one of three states: \( \text{passive} \), \( \text{waiting} \), and \( \text{busy} \). The multiset representing the state of the system contains free resources \( r_j \), busy users \( (u_i, r_j) \) where \( r_j \) is the allocated resource, waiting users \( (u_i, \text{waiting}) \) and passive users \( (u_i, \text{passive}) \). Transitions between states can occur as follows: \( \text{Passive} \rightarrow \text{Waiting} \rightarrow \text{Busy} \).

\[
\begin{align*}
\text{Resource allocation} &= \text{Res all 1} + \text{Res all 2} + \text{Res all 3}.
\text{Res all 1} &= \Gamma((R_1, A_1))(s) \text{ where } \\
R_1((u_i, \text{passive})) &= \text{True} \\
A_1((u_i, \text{waiting})) &= \{(u_i, \text{waiting})\}
\text{Res all 2} &= \Gamma((R_2, A_2))(s) \text{ where } \\
R_2((u_i, \text{waiting}), r_j) &= \text{True} \\
A_2((u_i, \text{waiting}), r_j) &= \{(u_i, r_j)\}
\text{Res all 3} &= \Gamma((R_3, A_3))(s) \text{ where } \\
R_3((u_i, r_j)) &= \text{True} \\
A_3((u_i, r_j)) &= \{(u_i, \text{passive}), r_j\}
\end{align*}
\]

Figure 1. The Gamma program for the resource allocation problem.

This example shows that Gamma does not care to specify the behaviour of the sequential processes that use the resources. This is because Gamma has not been designed as a language supporting the cooperation of sequential agents.

3 Gammalog

A Gamma program consists of a multiset transformation rule; its semantics can be modeled as a relationship on multisets, thus we can represent Gamma programs as predicates on multisets provided that we extend logic programming with this new data structure. A Gamma program then can be translated into a predicate taking two arguments: the first one represents the initial multiset and the second the final one.

3.1 Adding multisets in Logic Programming

In order to add multisets in pure logic programming we will exploit the approach presented in [11] to add sets. We will use double angle brackets \(<< \) and \(>>\) to denote multisets; we also represent partially known multisets in this way: \(<<x_1, \ldots, x_{n-1}, \text{rest}>>\) is a multiset containing some known elements \(x_1, \ldots, x_{n-1}\) whereas \(\text{rest}\) represents the rest of the multiset. The unification algorithm, which has to take into account the new equality defined on multisets, is non-deterministic since in general a unification between two multisets has more than one solution. For instance, \(<<x, y>>\) \(=<<1, 2>>\) returns two substitutions: \(\{x = 1, y = 2\}\) and \(\{x = 2, y = 1\}\) which are both correct. Formal aspects related to this issue are described in Section 4.1. In this section we suppose we have defined an extension of logic programming supporting first class multisets. The implementation of this extension is described in [7].

3.2 Syntax

Gammalog is a logic programming language extended with multisets plus the two connectives: \(\Gamma\) and \(\equiv\). The former connective allows one to define programs (Gamma clauses) following the Gamma style by specifying the multiset transformation reactions and actions; the latter provides a way to define new programs (definition clauses) exploiting the composition operators \(\circ\) and \(+\).

Gammalog clauses have the following syntax:

UBLCS-96-12
Gamma.Clause ::= S.Predicate(M_In, M_Out) Γ
          | L.Predicate & L.Predicate.
Def.Clause ::= G.Predicate ≡ Def.
Def ::= Def o Seq | Seq.
Seq ::= Seq + Par | Par.
Par ::= S.Predicate.

$M_{In}$, the first argument of $S$.Predicate, is a multiset containing the elements which have to be replaced by the elements in the multiset $M_{Out}$. $L$.Predicate stands for standard predicates in Logic Programming: the first one represents the reaction and takes $M_{In}$ as an argument; the second one is the action, which uses both $M_{In}$ and $M_{Out}$. The connective $\equiv$ allows one to define new predicates, starting from predicates defined with the $\Gamma$ connective ($S$.Predicates). Thus a recursive definition of a $G$.Predicate is not allowed. The operator $+$ has a higher precedence with respect to $\circ$ and parentheses are not allowed. This restriction is necessary to guarantee a correct operational semantics.

As discussed in [8], the semantics of Gamma programs having the form $(P \circ Q) + H$ presents compositional problems. Such a problem can be solved by adopting an alternative semantics also presented in [8]. Adopting such a new semantics we can prove that a Gamma program in the form $(P \circ Q) + H$ can be transformed into an equivalent one in the form $(P + H) \circ (Q + H)$ which can be implemented in Gammalog. Thus, if we assume the semantics presented in [8], Gammalog is also able to deal with general Gamma programs.

As an example, the Max_element program presented in Section 2 can be written in Gammalog as follows: (in the following we denote predicates names with identifiers beginning with upper case letters, while a variable begins with a lower case letter; this is also the choice of Gödel).

\[
\begin{align*}
\text{Max_element} & (<<x, y>>, <<z>>) \Gamma \\
R(x, y) & \land \\
A(x, y, z) . \\
R(x, y) & \leftarrow x \ll y . \\
A(x, y, z) & .
\end{align*}
\]

where the first clause is a Gamma clause and both $R(x, y)$ and $A(x, y, z)$ are predicates. Moreover, if we assume that the programs Ones, Non_neg and Add are defined by Gamma clauses, the program Positive_Integers which computes the number of positive values in a multiset can be defined by the following definition clause:

\[
\text{Positive_Integers} \equiv \text{Ones} + \text{Non_neg} \circ \text{Add}.
\]

**Definition 3.1 - Gammalog Program.** A Gammalog program $P$ is composed by a set of Gamma clauses $P^\Gamma$, a set of definition clauses $P^\equiv$ and a logic program $P^{LP}$. The logic program $P^{LP}$ can be partitioned into two subsets: $P^R$, which contains the predicates defining the reaction; and $P^A$ which contains the predicates defining the action.

**Definition 3.2 - Gammalog Query.** A Gammalog query $Q$ is a positive literal involving predicates defined in $P^\Gamma$ or in $P^\equiv$, where the first argument is a ground multiset.

The definition 3.2 introduces a limitation to restrict the set of possible goals of Gammalog. In fact, since the input multiset of a Gamma program is assumed to be always ground, a Gammalog goal can never flounder. This is an important restriction which is needed to guarantee the allowedness of Gammalog programs, as discussed in Section 4.2.

### 3.3 Translating Gammalog into Logic Programming

We define formally the function $\psi$ which, given a Gammalog program $P$, returns its translation $P^M$ into Logic Programming with multisets.
Definition 3.3  Let P be a Gammalog Program and c be a clause of P. The translation function \( \psi : P^T \cup P^= \cup P^{LP} \rightarrow P^M \) is defined as follows:

Case 1. P is a Program:

\[
\psi(\emptyset) = \emptyset \\
\psi(c \cup P) = \psi(c) \cup \psi(P)
\]

Case 2. c has the form \( P(\langle x_1, \ldots, x_n \rangle, \langle y_1, \ldots, y_m \rangle \Gamma) \)

\[
R(x_1, \ldots, x_n) & A(x_1, \ldots, x_n, y_1, \ldots, y_m).
\]

\[
\psi(c) = \{ P(m_1, m_3) \leftarrow \text{Step}_P(m_1, m_2) & P(m_2, m_3), \]
\[
\text{P}(m_1, m_2) \leftarrow \text{End}_P(m_1),
\]
\[
\text{Step}_P(\langle x_1, \ldots, x_n \rangle \Gamma, \langle y_1, \ldots, y_m \rangle \Gamma) =
\]
\[
R(x_1, \ldots, x_n) & A(x_1, \ldots, x_n, y_1, \ldots, y_m),
\]
\[
\text{End}_P(m) = \sim (m = \langle x_1, \ldots, x_n \rangle \Gamma & R(x_1, \ldots, x_n))\}.
\]

Case 3. c has the form \( H \equiv P + Q \)

\[
\psi(c) = \{ H(m_1, m_2) \leftarrow \text{P+Q}(m_1, m_2),
\]
\[
\text{P+Q}(m_1, m_3) \leftarrow \text{Step}_P+Q(m_1, m_2) & \text{P+Q}(m_2, m_3),
\]
\[
\text{P+Q}(m_1, m_1) \leftarrow \text{End}_P+Q(m_1),
\]
\[
\text{Step}_P+Q(m_1, m_2) \leftarrow \text{Step}_P(m_1, m_2),
\]
\[
\text{Step}_P+Q(m_1, m_2) \leftarrow \text{Step}_Q(m_1, m_2),
\]
\[
\text{End}_P+Q(m_1, m_1) \leftarrow \text{End}_P(m_1) \& \text{End}_Q(m_1)\}.
\]

Case 4. c has the form \( H \equiv P \circ Q \)

\[
\psi(c) = \{ H(m_1, m_3) \leftarrow \text{P}(m_1, m_2) \& \text{Q}(m_2, m_3)\}.
\]

Case 5. \( c \in P^{LP} \)

\[
\psi(c) = \{ c \}
\]

Let’s see the case 2: \( \text{Step}_P \) is a predicate which expresses one step of multiset transformation. It is associated to the pair \( (R, A) \) representing the multiset transformation rule in the source Gamma program. The reaction condition \( R \) is a logical predicate which takes \( x_1, \ldots, x_n \) as arguments; the action \( A \) needs also the additional arguments \( y_1, \ldots, y_m \) representing the elements that must be added to the multiset in place of \( x_1, \ldots, x_n \). We assume that if \( R(x_1, \ldots, x_n) \) succeeds also \( A(x_1, \ldots, x_n, y_1, \ldots, y_m) \) succeeds with ground \( y_1, \ldots, y_m \) according to the definition of \( \Gamma \) operator. The recursive structure of \( P \) allows one to make a sequence of multiset transformations until the termination condition is reached.

The translation of the parallel composition operator \( + \) (case 3) requires the definition of an auxiliary predicate \( ‘P+Q’ \) having the same recursive structure of the predicate defined in case 2, where the predicate \( ‘\text{Step}_P+Q’ \) express the transformation rules of the composing programs, and the predicate \( ‘\text{End}_P+Q’ \) is the joint termination condition expressed by the conjunction of the termination conditions of the two programs. These two predicates must be introduced to have a proper recursive definition because \( P \) and \( Q \) can be arbitrary programs. However, the syntactic limitations we have imposed to Gammalog guarantee that only programs which include the parallel composition operator can be composed in this way. Thus, since the parallel composition operator is associative, we can introduce the following optimized definition of \( ‘P+Q’ \) which will be used in the examples:

\[
‘P+Q’(m_1, m_3) \leftarrow \text{Step}_P(m_1, m_2) & ‘P+Q’(m_2, m_3),
\]
\[
‘P+Q’(m_1, m_3) \leftarrow \text{Step}_Q(m_1, m_2) & ‘P+Q’(m_2, m_3),
\]
\[
‘P+Q’(m_1, m_1) \leftarrow \text{End}_P(m_1) & \text{End}_Q(m_1).
\]

UBLCS-96-12 6
where, we can add more programs to the parallel composition by introducing new clauses for ‘P+Q’. For instance, if P is defined as ‘P1+P2’ we define a predicate ‘P1+P2+Q’ having four clauses which respectively represent Step P1, Step P2, Step Q and the final condition.

In case 4 the sequential composition operator δ states that the two Gamma programs are executed one after the other; so it can be translated into a conjunction where the multiset resulting from the execution of the first program is taken as an input of the second program. In this case we do not need to define a “Step” and an “End” program associated to the new predicate, because the syntax we have defined states that programs including sequential composition cannot be composed in parallel with other programs (see also Section 3.2).

We illustrate the translation technique with the following two examples.

**Example 3.4** The translation into logic programming with multisets of the Gammalog program Max_element presented above is:

```prolog
Max_element(m_1, m_3) ←
    Step Max_element (m_1, m_2) &
    Max_element (m_2, m_3).
Max_element(m_1, m_1) ← End Max_element (m_1).
Step Max_element (<< x, y | rest >>, << z | rest >>) ←
    R(x, y) &
    A(x, y, z).
End Max_element (m) ← ~ (m = << x, y | rest >> & R(x, y)).
R(x, y) ← x =< y.
A(x, y, z).
```

We remark that the predicate Max_element never fails.

**Example 3.5** The following logic program is the translation into logic programming with multisets of the complete Positive_Integers program.

```prolog
Positive_Integers (m_1, m_3) ←
    'Ones+Non_neg' (m_1, m_2) & Add (m_2, m_3).
'Ones+Non_neg' (m_1, m_3) ←
    Step Ones (m_1, m_2) & 'Ones+Non_neg' (m_2, m_3).
'Ones+Non_neg' (m_1, m_3) ←
    Step Non_neg (m_1, m_2) & 'Ones+Non_neg' (m_2, m_3).
'Ones+Non_neg' (m_1, m_1) ←
    End Ones (m_1) & End Non_neg (m_1).
Step Ones (<< x | rest >>, << y | rest >>) ←
    R_Ones (x) & A_Ones (x, y).
End Ones (m) ← ~ (m = << x | rest >> & R_Ones (x)).
R_Ones (x) ← x > 1.
A_Ones (x, 1).
Step Non_neg (<< x | rest >>, rest) ←
    R_Non_neg (x) & A_Non_neg.
End Non_neg (m) ← ~ (m = << x | rest >> & R_Non_neg (x)).
R_Non_neg (x) ← x < 0.
A_Non_neg.
Add (m_1, m_3) ← Step Add (m_1, m_2) & Add (m_2, m_3).
Add (m_1, m_1) ← End Add (m_1).
Step Add (<< x, y | rest >>, << z | rest >>) ←
    R_Add (x, y) & A_Add (x, y, z).
```

UBLCS-96-12 7
The different reactions of the clauses of “Ones+NonNeg” can be executed in OR parallel on the same multiset.

A Gamma program $G$ can be easily translated into a Gammalog program $P$ where the reaction is a logic predicate and the action function is transformed into the corresponding relation. To show that the logic program $\psi(P)$ is equivalent to the original Gamma program $G$, we introduce an abstract transition relation $\longrightarrow_{\text{SLD}_N\text{F}}$ defining the behaviour of a Gamma program as the sequence of multisets produced during the computation, according to the informal semantics presented in Section 2. A formal definition of the operational semantics of Gamma is presented in [16]. The following result holds.

**Theorem 3.6 Correctness of the translation**

*Given a Gamma program $G$ and the corresponding Gammalog program $P$, the logic program $\psi(P)$ is equivalent to the original Gamma program $G$.

**Proof:** Let $\longrightarrow_{\text{SLD}_N\text{F}}(\psi(P))$ be the abstract transition relation associated to the logic program $\psi(P)$ modeling the sequence of multisets produced during the computation. Basically this transition relation depends on the Step predicates defined in Section 3.3. We have to prove that $\longrightarrow_{\text{SLD}_N\text{F}}(\psi(P))$ and $\longrightarrow_{G}$ are equivalent: in the following we give a sketch of the proof.

First, we have to prove that for every multiset $m$, if $m \longrightarrow_{G} m'$, then $m \longrightarrow_{\text{SLD}_N\text{F}}(\psi(P)) m'$ and vice versa. This can be done by induction on the structure of the program. For instance, if $G$ is a simple program and $m \longrightarrow_{G} m'$, there are elements in $m$ which satisfy the reaction of $G$, thus the Step program generated in the case 2 of $\psi$ can be also executed on the same elements (because of multiset unification) and $m \longrightarrow_{\text{SLD}_N\text{F}}(\psi(P)) m'$.

Then, we have to prove that for all such that $m \not\longrightarrow_{G}$ then $m \not\longrightarrow_{\text{SLD}_N\text{F}}(\psi(P))$ and vice versa. This can be also proved by induction on the structure of the program. For instance, if $G = G_1 + G_2$ and $m \not\longrightarrow_{G}$, there are no elements in the multiset $m$ which satisfy the reaction of $G_1$ or the reaction of $G_2$, thus both the End programs must be true and $m \not\longrightarrow_{\text{SLD}_N\text{F}}(\psi(P))$ (i.e., the End predicate succeeds and $m$ is not transformed further).

---

4 **Semantics**

Two are the main aspects of the semantics of Gammalog that we analyse: first the extension of logic programming with multisets and the related equality axioms; second the proof of the soundness of Gammalog and the completeness of classes of Gammalog programs with respect to the extended SLD resolution.

In this section we use the standard notation of Lloyd [23], and we also suggest this book as a reference to a reader who is not familiar with logic programming. In particular, let $F$ be a set of functional symbols, $V$ a set of variables, $T$ the set of first order terms over $F \cup V$; $x, y, z$ will be generic variables ranging over $V$; $f$ and $g$ will range over $F$; $u, h, k, t, t_\ell$ will be terms; and $r$ and $s$ will be multiset terms. Finally, we assume that $A, A_i, B, B_i$ will range over literals.

4.1 **Extending logic programming with multisets**

The treatment of multisets in logic programming is similar to the treatment of finite sets presented in [12]. In particular, we need to define a set of equality axioms which express when two multisets are equal. Such a theory allows one to build a multiset unification algorithm which is non deterministic; thus, we also need to extend SLD-resolution and we need to prove soundness and completeness results for the extended logic programming language.

In this section we report the equality axioms for multisets, given this set of axioms it is not difficult to extend the results presented in [12] to logic programming with multisets, thus we report here the main results and we only provide the sketch of the proofs.
Let $\text{In}/3$ be the predicate which expresses multiset membership\(^4\): the first argument represents the number of occurrences of the second argument in the multiset in the third argument; and let $\text{Inc}_m(x, y)$ be the multiset constructor representing the multiset which is the union between $\text{In}/3$ and the multiset $y$. Let $\text{In}/3$ be the empty multiset. To have a suitable first-order multiset theory with equality, we add a set of axioms dealing with multisets to the standard equality axioms and the Clark freeness axioms \cite{2}, where the axiom $f(z_1, \ldots, z_n) = f(y_1, \ldots, y_n) \rightarrow z_1 = y_1, \ldots, z_n = y_n$ holds for all functors $f/n \neq \text{Inc}_m/2$.

**Definition 4.1 Multiset Axioms.**

(\textbf{Z}) $\text{In}(0, x, \text{In}/3) \rightarrow \text{In}(0, x, y) \wedge y = \text{In}(0, x, y)$

(\textbf{I}) $\text{In}(n+1, x, \text{Inc} \text{m}(y, z)) \leftrightarrow$

$(x \neq y \wedge \text{In}(n+1, x, z)) \lor (x = y \wedge \text{In}(n, x, z))$.

this rule specifies how the function $\text{Inc} \text{m}$ operates;

(\textbf{W}) $\text{In}(n, y, x) \rightarrow \exists z(\text{In}(0, y, z) \wedge x = \overline{\text{Inc} \text{m}(y, \text{Inc} \text{m}(y, \text{Inc} \text{m}(y, \text{Inc} \text{m}(y, z))})})$

(that is $y$ is inserted $n$ times in $x$); it is the “without” axiom which guarantees the existence of the multiset $z$ without any $y$;

(\textbf{L}) $\text{In}(n+1, y, x) \rightarrow \exists z(\text{In}(n, y, z) \wedge x = \text{Inc} \text{m}(y, z))$

“less” axiom to guarantee the existence of the multiset $x \setminus \text{In}/3$;

(\textbf{E}) $\text{Inc} \text{m}(x, v) = \text{Inc} \text{m}(y, w) \leftrightarrow$

$(x = y \lor v = w) \lor \exists z(\text{Inc} \text{m}(y, z) \wedge w = \text{Inc} \text{m}(x, z))$

“equality” axiom to establish when two multisets are equal;

(\textbf{R}) $\exists z(\text{In}(n, y, x) \rightarrow (\text{In}(m, z, x) \wedge \text{In}(0, y, z)))$

“regularity” axiom; it guarantees the membership does not generate loops.

(\textbf{U}) $f(z_1, \ldots, z_n) \neq \text{In}/3 \wedge \text{In}(0, x, f(z_1, \ldots, z_n))$

where $f/n \neq \text{In}/3$ and $f/n \neq \text{Inc} \text{m}/2$.

The axiom (I) corresponds to the axioms (W1) and (W2) in \cite{12}, while the axioms (W) and (L) correspond to the axiom (L), in the same paper.

Note that, as with sets, a special equality is required on multisets because the order of elements in a multiset is irrelevant; thus, the permutativity property holds. The axiom (E) guarantees this property: by applying it and (I) it is easy to prove that the following equality holds:

$\text{Inc} \text{m}(z, \text{Inc} \text{m}(y, x)) = \text{Inc} \text{m}(y, \text{Inc} \text{m}(z, x))$.

We define the immediate consequence operator and the least Herbrand model for logic programming with multisets.

First, we consider the ordinary Herbrand universe $H$ \cite{23} (generated by the collection $F$ of functors, where $\text{Null}/0 \in F$, and $\text{Inc} \text{m}/2 \in F$), and we take the smallest equivalence relation $\sim$ over $H$ that fulfills the above permutativity property. Then, we take a representative term from each equivalence class of $H/\sim$: a ground term $t_g$ is said to be canonical if either it is a constant or every one of its subterms is canonical and $t \leq u$ ($\leq$ can be defined antilexicographically) holds for every subterm of the form $\text{Inc} \text{m}(u, \text{Inc} \text{m}(t, s))$ of $t_g$.

Suppose to have $\tau$ to be the function that maps every ground term to its canonical representative. This function is trivially extended to apply to ground atoms and clauses, and sets thereof.

**Definition 4.2 - privileged interpretation domain.** The privileged interpretation domain of any given program $P$ is a set $H_P^M = \{ \tau(t) \mid t \in H_p \}$ where $H_p$ is the Herbrand universe associated to the program $P$.

The Herbrand base of $P$ is defined as $\tau(B_p)$ where $B_p$ is defined as usual \cite{23}. A (multiset) interpretation $I^M$ of $P$ is a subset of $\tau(B_p)$, and it is a model of $P$ if it satisfies $\text{ground}(P)$ which is the set of all the possible ground instances of the clauses of $P$.

---

\(^4\) This notation indicates that the predicate $\text{In}$ has arity $3$. 

UBLCS-96-12 9
Now, we can define the immediate consequence operator and the least Herbrand model for logic programming with multisets:

\[ T^M_\{P\} (I^M) = \{ A \mid A \vdash B_1 & \ldots & B_n \in \sigma(ground(P)) \} \]

and

\[ M^M_\{P\} = \bigcap_{M'^M_{modelofP}} M'^M . \]

In this context, we can easily prove the usual model-theoretic and fixpoint semantics results, for instance we have:

\[ M^M_\{P\} = T^M_\{P\} \uparrow \omega. \]

The development of an operational semantics for a logic programming language with multisets requires that the SLD procedure is modified in such a way that it includes multiset unification [13]. In the next section we provide a general description of the multiset unification algorithm which reflects the axiom (E).

4.1.1 Multiset Unification

Let E be a Herbrand system, i.e. a set of equations \{t_i = t'_i, i = 1..n\}, where the \(t_i\)'s are terms (variable or not). A Herbrand system is said to be in solved form if all equations have the form \(x_i = u_i\) (i = 1..m), and every variable \(x_i\) occurs exactly once in E. Such a system has the obvious solution \(\{x_i \leftarrow u_i, \ldots, x_n \leftarrow u_m\}\).

Definition 4.3 - Unification Algorithm. Given a Herbrand system E, the following multiset unification algorithm returns the system in solved form or reports a failure. The algorithm is non-deterministic; thus it may return more than one solution exploiting backtracking. The notation \(r \backslash t_i\) will denote the term obtained from a multiset term \(r\) by taking away its \(i\)-th element.

\begin{verbatim}
function unify(E: Herbrand system): Herbrand system;
begin
if E is in solved form then return E
else select arbitrarily an equation \(e\) in E so that:
\hspace{1cm} case \(e\) of
\hspace{2cm} 1) \(x = x\): return unify(E \ {\{e\}});
\hspace{2cm} 2) \(t = x\): t is not in \(V\): return unify((E \ {\{e\}}) \cup \{x = t\});
\hspace{2cm} 3) \(x = t\): t is not a multiset term and \(x\) occurs in \(t\): return unify((E \ {\{e\}}) \cup \{t = x\});
\hspace{2cm} 4) \(x = Inc_M(t_0, \ldots, Inc_M(t_n, t) \ldots)\) and \(x\) occurs in \(t_o\) or in \(t_0\) or \ldots or in \(t_n\): return unify((E \ {\{e\}}) \cup \{t_0 = x, \ldots, t_n = x\});
\hspace{2cm} 5) \(x = t\), \(x\) does not occur in \(t\), \(x\) occurs somewhere else in \(E\):
\hspace{1cm} return unify((E \ {\{e\}}) \sigma \cup \{e\}) where \(\sigma\) is the substitution \(\{x \leftarrow t\}\);  
\hspace{2cm} 6) \(f(t_1, \ldots, t_n) = g(t'_1, \ldots, t'_n), f \neq g\) or \(n \neq m\): fail
\hspace{2cm} 7) \(f(t_1, \ldots, t_n) = f(t'_1, \ldots, t'_n), f / n \neq Inc_M / 2\):
\hspace{1cm} return unify((E \ {\{e\}}) \cup \{t_1 = t'_1, \ldots, t_n = t'_n\});
\hspace{2cm} 8) \(r = s\), where \(r\) is \(Inc_M(t_0, \ldots, Inc_M(t_n, h) \ldots)\) and \(s\) is \(Inc_M(t'_0, \ldots, Inc_M(t'_n, k) \ldots)\) and \(h\) and \(k\) are terms with main functor \(\neq Inc_M / 2\);
\hspace{1cm} if \(h, k\) are not the same variable then
\hspace{1cm} 8.1) choose one among the following actions
\hspace{1cm} and backtrack if failure occurs:
\hspace{1cm} a) return unify((E \ {\{e\}}) \cup \{t_0 = t'_0, r \backslash t_0 = s \backslash t'_0\});
\hspace{1cm} b) return unify((E \ {\{e\}}) \cup \{r \backslash t_0 = Inc_M(t'_0, N), Inc_M(t_0, N) = s \backslash t'_0\}), N new variable;
\hspace{1cm} else \(h, k \in V, h = k = x\)
\hspace{1cm} 8.2) if \(n=m\) then
\end{verbatim}
choose i from 1..m and (backtrack if failure occurs)
return unify\((E \setminus \{e\}) \cup \{t_0 = t'_0, r \setminus t_0 = s \setminus t'_0\} ;
else fail.

This algorithm is similar to the set unification algorithm presented in [11]. The main difference
is the absence of cases 9.1(b), 9.1(c), 9.2(b), 9.2(c), 9.2(d) in the case 8 of our algorithm, because we
do not need to take into account the absorption property which does not hold for multiset terms.
The main consequence of this is that the proof of the following two theorems can be considered
as an instance of the proof of the analogous theorems on set unification [11]. For this reason we
only give here a sketch of the proofs; the interested reader can refer to [11, 12] for the complete
proofs.

**Theorem 4.4 - Termination of multiset unification.** If E is a Herbrand system, unify\((E)\) terminates.

**Proof:** To prove the termination of the algorithm we proceed by induction on the size of the
Herbrand system and the size of the unifying multisets in the Herbrand system. Intuitively, either
the size of the Herbrand system (case 5), or the length of the unifying multisets (cases 8.1(a), 8.1(b),
8.2) decreases at each recursive call so the algorithm always terminates. The complete proof is
similar to the proof of termination of the set unification algorithm presented in [12].

**Theorem 4.5 - Soundness and completeness of multiset unification.** Given a Herbrand system E,
the set of its solutions Sol\((E)\) produced by the multiset unification algorithm is sound and complete.

**Proof:** Let’s briefly examine action 8 of the algorithm. The case 8.2 says that two multisets with
the same tail (a variable or the empty multiset) unify if they contain the same elements modulo
permutativity. The combination of cases 8.1(a) and 8.1(b) guarantees the permutativity property
when the multisets have different unspecified tails. For instance, to unify \(<1,2|x>\) and
\(<1,3|y>\), first the rule 8.1(a) which calls recursively the unification between \(<2|x>\) and
\(<3|y>\) is applied. Then, the case 8.1(b) is chosen, and the unification is split in two branches:
x = <3|z> and <2|z> = y, which represent the only correct solution to the problem.

4.2 Soundness and Completeness of Gammalog

The semantics of Gammalog is given in term of its translation into logic programming with
multisets. The clauses of the logic program \(\psi(P)\) contain also negated literals, thus we need to
restrict Gammalog to a particular class of programs, those ones with non-recursive reactions, in
order to provide soundness and completeness results.

The dependency graph \(D_P\) for a program P is a directed graph with signed edges. The nodes
are the predicates occurring in P. There is a positive (resp. negative) edge \((r,q)\) if a clause in P
has the predicate r in its head and the predicate q in a positive (resp. negative) literal in its body.
P depends evenly (resp. oddly) on q if there is a path from p to q with an even (resp. odd) number
of negative edges. Given a general program P and a general goal Q, we recall here three basic
definitions from [2]:

**Definition 4.6 - Strictness.** We say that P is strict w.r.t. Q if no predicate occurring in Q depends both
evenly or oddly on a predicate defined in P.

**Definition 4.7 - Stratified Program.** A program P is stratified if no cycle with a negative edge exists in
its dependency graph.

**Definition 4.8 - Stratification.** Given a program P, a stratification of P is a partition \(P_1 \cup P_2 \cup \ldots \cup P_n\)
such that each \(P_i\) depends positively only on predicates defined in \(P_j, j \leq i\), and negatively only on
Stratified programs have the semi-strictness property [21]. A program $P$ is semi-strict if it does not contain predicates which depend on their negation.

SLDNF resolution has been proved sound w.r.t. two-valued semantics of program completion [10], while completeness holds for a semi-strict program $P$ and a goal $Q$ such that $P$ is strict and allowed w.r.t. $Q$ [21].

Allowedness is a condition which guarantees that $P$ and $Q$ do not flounder, i.e., there is a safe selection rule for literals which guarantees that only ground negated literals are selected exploiting SLDNF resolution. This condition holds in Gammalog since the multiset we give as an argument to negated literals is always ground, this is true because we have imposed a restriction to Gammalog queries, as presented in Section 3.2, the input multiset to a Gammalog program is always ground.

Lemma 4.9 - Stratification of Gammalog. A Gammalog program $P = P^T \cup P^e \cup P^LP$ is stratified if $P^LP$ is stratified.

Proof: If a stratification of $P$ exists then it can be proved stratified [2]. Thus we need to show that such a stratification exists for any Gammalog program which satisfies the hypothesis. Given a Gammalog program $P$ we define a stratification on the program $\psi(P)$ as follows. We proceed analysing the different cases of $\psi$. For each case we provide a stratification of the resulting program, given a clause $c$: in case 2, we have the stratification $p_1 = \phi$, $p_2 = \{P, StepP, EndLP\}$; in case 3 and 4, we do not introduce negation thus we only define a set $p_2 = \{H\}$ which includes all the clauses generated by the transformation and $p_1 = \phi$; in case 5, we put $p_1 = \{c\}$ and $p_2 = \phi$ for every $c \in P^LP$. Finally, considering case 1 we define the stratification of the whole program as follows: $P_1 = \bigcup_{c \in P} p_1$ and $P_2 = \bigcup_{c \in P} p_2$. 

Lemma 4.10 - Semi-strictness of Gammalog. A Gammalog program $P = P^T \cup P^e \cup P^LP$ is semi-strict.

Proof: The proof follows from the previous lemma and a well known result of [21].

The strictness condition is introduced to solve completeness problems of particular programs. For instance, given a logic program $\{p \leftarrow q; p \leftarrow q; q \leftarrow q\}$, $p$ is a logical consequence of the program, but no successful SLDNF derivation exists, because of the recursive definition of $q$.

A Gammalog program $P$ is not strict w.r.t. a Gammalog query $Q$, since a predicate occurring in $Q$, which represents a Gamma clause, depends both evenly and oddly on the reaction predicate. However, if we limit $P^R$ to non-recursive predicates, we can relax the strictness condition to prove completeness of Gammalog.

In the following we summarize the main results of the paper:

Theorem 4.11 - Soundness of Gammalog. Let $P$ be a Gammalog program and $G = (\leftarrow A_1 \& \ldots \& A_k)$ be a Gammalog goal. If $G$ has a refutation in $\psi(P)$ with computed answer $\theta = \sigma_1 \ldots \sigma_n$ this answer is correct for $\psi(P) \cup \{G\}$.

Proof: The soundness of Gammalog follows immediately from the soundness of logic programming with multisets and of SLDNF. This because a Gammalog derivation always corresponds to a SLDNF derivation.

Theorem 4.12 - Completeness of Gammalog. Let $P = P^T \cup P^e \cup P^R \cup P^A$ be a Gammalog program and $G$ be a Gammalog goal. Then, for each correct answer $\sigma$ for $\psi(P) \cup \{G\}$ $G$ has a refutation in $\psi(P)$ if $P^R$ is a non-recursive program and $P^A$ is a general program strict (w.r.t. action queries) and stratified.

Proof: The proof follows from lemmas 4.9 and 4.10 and from the well known result of [21]. Since the reaction condition is assumed to be always a non-recursive predicate, the strictness of a
4.3 A Parallel Operational semantics

The operational semantics of logic programming with multisets does not provide a satisfying execution model for Gammalog if one wants to build a parallel implementation of the language. This is because the $\Gamma$ operator is based on don’t care non-determinism, i.e., when a reaction condition commits the choice is not backtrackable. Moreover, adopting the standard operational semantics of Gamma is not a satisfying solution because our goal is to support coordination, as discussed in Section 2.

Thus, to give an operational definition of Gammalog to be used as a reference for a possible parallel implementation of the language, we define an operational semantics modeling its committed choice behaviour, where coordination issues are separated from computation issues.

The idea is to release the assumption that the execution of Gamma rewriting rules is atomic. More precisely, we provide an operational semantics were Gamma atomic rewriting rules are divided into two atomic steps: the reaction and the action. Actions are represented in Gammalog as standard logic predicates which compute the elements that must be added to the multiset (see also the example below, in Section 5). These predicates are sequential processes that can be executed in parallel and are coordinated by Gammalog rewriting rules.

We give a formal operational semantics of Gammalog based on the Plotkin Structural Operational Style (SOS) [25]. SOS computations are defined through a derivation relation over configurations (states), which is defined as the least relationship satisfying a set of axioms and rules.

A Gammalog program (see section 3) is composed by a set of rules, which specify the transformations of a multiset. We suppose that the activation of a program generates a set of processes corresponding to the rules, which operate on a shared data structure (the multiset) executing in parallel their reaction conditions and actions.

Let $M, M'$ be multisets, $x_1, \ldots, x_n, y_1, \ldots, y_m$ be terms of the multisets, $P$ be the program. The notation $P \parallel R$ and $P \parallel A$ indicate the reaction and the action of $P$ respectively.

Definition 4.13 - Gamma Configurations

$\Gamma = (\Gamma_{IDLE} \cup \Gamma_{ACTION} \cup T) \times M$

where

$\Gamma_{IDLE} = \{ P \mid \text{is a process associated to a Gammalog program}\}$

$\Gamma_{ACTION} = \{ P^A \mid \text{is a process evaluating the action of the program } P \}$

$T = \{ P_f \mid \text{is a process in a final state}\}$.

Definition 4.14 - Transition system for simple Gammalog programs. A transition system modeling a simple Gammalog program $P$ is a tuple $(\Gamma, \rightarrow_\gamma, T)$, where $\Gamma$ is a set of configurations, $\rightarrow_\gamma \subseteq \Gamma \times \Gamma$ is the smallest transition relation satisfying the following axioms and rules and $T$ is the set of terminal configurations.

step selection

$M = \ll{x_1, \ldots, x_n|M'}\rr$ and $P \parallel R(x_1, \ldots, x_n) \implies (P, M) \rightarrow_\gamma (P^A[x_1, \ldots, x_n], M')$

step execution

$(P^A[x_1, \ldots, x_n], M') \rightarrow_\gamma (P, \ll{y_1, \ldots, y_m|M'}\rr)$

where $P \parallel A(x_1, \ldots, x_n, y_1, \ldots, y_m)$. 
The main difference with the semantics of Gamma presented in [16] is that we separate the execution of the reaction and the action. According to our semantics the reaction and the action are two atomic operations. This allows one to use Gammalog as a coordination language where the sequential execution concerns the execution of action predicates. This make Gammalog suitable for coordination.

The semantics of a complex Gammalog program, i.e., a program built exploiting the sequential and parallel execution operators is defined in a hierarchical way in terms of the transition system described above.

**Definition 4.15** A transition system modeling a complex Gammalog program is a tuple $(\Gamma, \rightarrow, T)$ where $\rightarrow$ is the smallest transition relation satisfying the following rules:

**Sequential composition**

$$ (P \circ Q, M) \rightarrow (Q, M) $$

$$ (P, M) \rightarrow_{\gamma} (P', M') $$

$$ (P \circ Q, M) \rightarrow (P' \circ Q, M') $$

**Parallel composition**

$$ (P, M) \rightarrow_{\gamma} (P', M') $$

$$ (P + Q, M) \rightarrow (P' + Q, M') $$

$$ (Q + M') \rightarrow_{\gamma} (Q', M') $$

$$ (P + Q, M) \rightarrow (P + Q', M') $$

**Termination**

$$ (((P, M) \rightarrow_{\gamma} (P', M)) \text{ and } ((Q, M) \rightarrow_{\gamma} (Q', M))) $$

$$ (P + Q, M) \rightarrow ((P + Q), (M')) $$

The following result shows that, if a Gamma program terminates, the new operational semantics proposed for Gammalog is equivalent to the original operational semantics of Gamma. Let $\rightarrow_{Glog(P)}$ the abstract transition relation which describes multiset transformations of a Gammalog program $P$ with respect to the operational semantics above.

**Theorem 4.16** For any Gamma program $G$ let $P$ the corresponding Gammalog program. $\forall m, m'$, if $m \rightarrow_{Glog(P)} m'$ and $m' \not\rightarrow_{Glog(P)}$ then $m \rightarrow_{\Gamma(G)} m'$ and $m' \not\rightarrow_{\Gamma(G)}$ and vice versa.

**Proof:** We first prove that, given a Gammalog computation which terminates, an equivalent Gamma computation always exists.

Let $m, m_1, m_2, m_3, \ldots m_f$ be a sequence of multiset which is the result of a possible execution with respect to the new semantics, where $m_f$ is a final state (multiset). Let $\text{fair}(m_i)$ be a predicate which returns true if in the state $m_i$ there are not actions that have to be completed. A final state is always $\text{fair}$.

We prove by induction on the length of the computation that, $\forall m_i$ such that $\text{fair}(m_i)$, there is a Gamma computation which generates the multiset $m_i$.

- **Induction base:** The initial multiset is fair and is the same for Gammalog and Gamma.
- **Induction step:** let’s suppose that, for all the Gammalog computations of $n$ steps such that the resulting multiset $m_n$ is fair, there is a Gamma computation which generates $m_n$. Let $m_{n+i}$ be the first multiset after $m_n$ in the Gammalog computation such that $\text{fair}(m_{n+i})$ (this multiset always exists because the computation terminates). If $i < n$ then we can
apply the induction hypothesis starting from $m_n$ and thus a Gamma computation which generates $m_{n+i}$ exists; otherwise we show that, an equivalent Gammalog computation such that $m_{n+i-2}$ is fair always exists.

We consider the last Gammalog transition: $m_{n+i-1} \xrightarrow{G\log[P]} m_{n+i}$ which must be the execution of an action. We can move the corresponding reaction just before this transition without changing the final result (i.e., the multiset $m_n$), and now also fair ($m_{n+i-2}$) holds. This is always possible because the elements in the multiset which are removed from this reaction are not used in the reactions which follow. If $i - 2 < n$ then we can apply the induction hypothesis; otherwise we generate an equivalent Gammalog computation such that $m_{n+i-4}$ is fair, and so on.

Since fair ($m_f$) holds an equivalent Gamma computation which generates $m_f$ always exists.

The prove of vice versa is not difficult, in fact a computation with respect to the original Gamma semantics is also a computation with respect to the new semantics, where all the reactions are always followed by the corresponding actions.

5 Gammalöö: an instance of Gammalog

Gammalog is simply an abstract model; if we want to design a real coordination language several practical choices need to be made concerning its execution model. Our choice has been to embed Gammalog in the host language Gödel for a number of reasons: Gödel already supports sets; it is a well engineered and relatively efficient tool; it is freely accessible; it is implemented via translation to SICStus Prolog and the Gödel compiler (written in SICStus) is quite easy to manage and extend. Finally, as its authors say [19], “Gödel reduces the effort involved in providing a parallel implementation of the language and offers substantial scope for parallelism in such implementations”, we hope to simplify our efforts of building a parallel logic language based on Gamma.

The new language we have designed is named Gammalöö; it is an extension of Gödel with the $\Gamma$ operator plus facilities which allow one to define programs exploiting sequential and parallel composition. For instance, the module in Figure 2 defines the program Resource Allocation which implements the same Gamma program presented in Section 2.

The connective $<=$ denotes Gamma clauses and the connective $<=>$ the definition clauses. Since Gammalöö is strongly typed in the same way as Gödel all the predicates appearing in a program must be declared in a given module. For instance, the declarations needed for Res_all_1, Res_all_2, Res_all_3 are:

PREDICATE Res_all_1,Res_all_2,Res_all_3: Multiset(Elem)*Multiset(Elem).

where Elem is a base type for multisets.

Simple Gammalöö programs are declared as ordinary Gödel predicates. On the other side, the predicate Resource Allocation being defined with the definition operator needs a particular declaration using the keyword GPREDICATE (which stands for Gamma Predicate) this is needed to inform the compiler that the predicate is a Gamma predicate. During the translation, this particular declaration allows the compiler to add the appropriate arguments to the Gamma predicate and to generate object code corresponding to the rules described in section 3.

Note that the action predicate A3 is defined by the predicate Use resource: this represents the sequential program which is coordinated by the above multiset-rewriting rules.

6 Implementation

The implementation of Gammalöö has been realized extending the Bristol version of Gödel. A Gammalöö program is first translated into Gödel and then it is translated into Prolog by the Gödel compiler.

In order to have adequate runtime support for Gammalöö we extended Gödel with first class multisets providing an extended unification algorithm. Then, we extended the Gödel compiler to deal with Gammalöö operators.
The extension of Gödel with multisets is based on an extended unification algorithm presented in Section 4.1.

Gödel's scanner and parser have been modified to recognize the new statements. The composition operators \( \lor \) and \( \land \) have the same precedence of "\( \lnot \)" (OR) and "\&" (AND) respectively. The Gödel's code generator has been modified as well in order to provide the translation of the new statements.

A particular treatment has been designed to compile clauses containing the definition connective \( \langle=\rangle \) with the operators \( \lor \) and \( + \) in their bodies: we compile the first operator as a conjunction, while the second operator requires the definition of an auxiliary predicate to handle the intrinsic nondeterminism. A parallel combination of Gammalög simple programs has the form: \( \text{Prog} \langle=\rangle \ P + Q \) and it is compiled in Gödel generating the following clauses:

\[
\begin{align*}
\text{Prog}(m_1, m_2) \leftarrow & \quad \text{'}P + Q^\prime(m_1, m_2). \\
\text{'}P + Q^\prime(m_1, m_3) \leftarrow & \quad \text{Step}_P(m_1, m_2) \mid \text{'}P + Q^\prime(m_2, m_3). \\
\text{'}P + Q^\prime(m_1, m_3) \leftarrow & \quad \text{Step}_Q(m_1, m_2) \mid \text{'}P + Q^\prime(m_2, m_3). \\
\text{'}P + Q^\prime(m_1, m_1) \leftarrow & \quad \text{End}_P(m_1) \land \text{End}_Q(m_1).
\end{align*}
\]

The bar commit ('\') between reaction conditions and actions in \( \text{Step}_P \) clauses is the Gödel pruning operator. The commit has the declarative meaning of a conjunction, and the following procedural meaning: only one solution is found for a formula in its scope (on the left of the
commit), all the other branches arising from the other clauses of the same predicate which contain a commit are pruned. The order in which the statements are tried is not specified, so that the meaning of ‘|’ is close to meaning of the commit of the concurrent logic programming languages.

7 Specifying an Operating System

A good example to test the expressiveness of a language for specifying and designing reactive systems is the specification of an operating system. Some recent papers where this example has been used are [18, 30]; such an example has also been widely used in logic programming, eg. in [22, 14, 6].

The following Gammalög program

We want specify the kernel of a simple operating system, managing a set of processes running in parallel, with provisions for file and process management, and support for communication.

Each process has a unique process-id. Processes run programs which, for the sake of simplicity, include only operating system primitives, like for instance operations for sending or receiving a message, forking a process, opening a file, and so on. We make the assumption that each program has a name. A process can be either Busy or Waiting. In the first case the process is active, in the second case the process is waiting for a message and is suspended.

Let a, b, ... range over process-id, f, f’ file-id, and P, P’ program names. The system primitives we specify are:

- `Send(a, b, m)` process a sends process b a message m;
- `Receive(b, mess)` is a blocking receive primitive; if there is a message for the current process b is instantiated to the process-id of the sender and mess to the contents of the message;
- `Fork(a, P)` creates a new process with a new process-id running the program P;
- `Exec(P)` the current process executes a new program P, the process-id is not changed;
- `Create(f)` creates a new file f;
- `Open(f)` opens the file f;
- `Close(f)` closes the file f;
- `Read(f, data, n)` reads data, which has length n, from the file f;
- `Write(f, data, n)` writes data, which has length n into the file f.

The specification we give consists of a hierarchy of modules according to the style of programming in Gödel. Every module usually includes an export part, containing the symbols which can be imported by other modules, and a local part, containing the symbols known only locally.

The main module is System (Figure 3), which does not have an export part, whereas imports from modules Messages, Process, and Files predicates to manage messages, processes, and files, respectively. The module System contains the predicate Sys, which is defined by a parallel composition of three imported predicates: Communication, Proc, and Man_files. Sys is a GAMMA predicate, thus it takes a multiset as an argument, which contains the “state” of the operating system, i.e., process ids, file descriptors, pending messages, and so on.

The structure of the multiset is defined in the module Declarations, which is described below. The constants P1, P2, ..., Pn represent names of programs.

The predicate Program represents a set of executable programs, each program has an associated name and consists of a sequence of system primitives.

The module Declarations (Figure 4) is imported by all the modules defining operating system primitives. Such a module consists of the export part only; it imports the module Multisets which contains basic declarations and predicates to deal with multisets. The module Declarations includes all the base types: Elem, to denote an element of the multiset; Mess, to denote a message; Prog, which is the type of program names. Such a module includes as well the declarations of the functions representing processes which are Busy and Waiting. The term `Busy(a, p)` denotes a busy process and the term `Waiting(a, b, p, mess)` denotes a process a which is waiting for a message from the process b; in both cases p is the program (a list of system primitives) associated to the process a; the variable mess, which also appears in p, will be bound when the message is received, see the module Messages defined below.
MODULE System.
IMPORT Messages, Processes, Files.
CONSTANTS P1, P2, ..., Pn : Prog. %n different program names are available.
PREDICATE Program : Prog * List(Elem).
%Program-id * List of statements of a program.
GPREDICATE Sys. % This GAMMA predicate represents the main program.
Sys ⊢ Communication + Proc + Man_files.
% This predicate represents programs that can be executed.
Program(P1, "body of P1").
Program(P2, "body of P2").
... Program(Pn, "body of Pn").

Figure 3. Main program: module System

EXPORT Declarations.
IMPORT Multisets.
BASE Elem, Mess, Prog.
FUNCTION Busy: Integer * List(Elem) → Elem; % to denote busy process.
% to denote a process waiting for a message.

Figure 4. Module Declarations

The module Messages in Figure 5 includes both an export part and a local part. The export part defines the predicate Communication which is a parallel combination of the predicates Send_mess, Receive_mess and Deliver_mess. In the local part of Messages there are the definitions of functions Send(a, b, m), Receive(a, m) and Message(a, b, m). The last one represents pending messages in the multiset. The first argument of Send and Message is the sender of message m, while the second one is the receiver process-id. The Receive primitive takes two variables as arguments, the first will be instantiated to the process-id of the sender and the second to the contents of the message.

Finally, such a module includes the declarations of the GAMMA clauses Send_mess, Receive_mess and Deliver_mess, which implement asynchronous message passing. Send_mess deals with the Send primitive, it pops the statement Send(a, b, mess) from the program (i.e., it increments the program counter) and inserts the message into the multiset of pending messages. Receive_mess deals with the Receive primitive, it pops the statement Receive(a, mess) from the program and it transforms the state of the process into waiting. Deliver_mess delivers a pending message to the destination.

The module Processes in Figure 6 includes two parts. The export part contains the definition of the GAMMA predicate Proc which handles the operations of forking, executing and terminating a process. The local part of Processes contains the declarations of functions Fork(process-id) and Exec(program-name). The GAMMA clauses in this module are: Fork_pr to fork a busy process; Exec_pr to exec a new program without changing the process identifier as the exec procedure of Unix; Term to terminate a process.

The operating system includes a file manager (Figure 7) as well. The module Files exports the GAMMA predicate Man_files, which is a parallel combination of simple predicates capable of creating, opening, closing, reading, and writing files. The implementation of these predicates is in the local part of Files together with the declarations of new symbols. The local declarations include: the base type Status, denotes if a file is open or closed; the constants Op and Cl, which stand for open file or closed file respectively; the functions Create(file-id), Open(file-id), Close(file-id), Read(file-id, data, n) and

UBLCS-96-12 18
GPREDICATE Communication.
% This GAMMA predicate handles the operations of
% sending and receiving messages.

Communication <$> Send mess + Receive mess + Deliver mess.$

Figure 5. Module Messages

Write(file-id, data, n), representing primitive operations. The function File(process-id, file-id, data, status, lookup) represents a file descriptor where data is the contents of the file, a list, and lookup is the current position in the list. File descriptor are initialized by the system primitive Create described below.

The module includes the declarations and the definitions of the GAMMA predicates Open_file, Crea_file, Close_file dealing with requests of opening, creating or closing files. Crea_file creates an empty file descriptor: data is the empty list and lookup = 0. Open_file sets the status of a file descriptor to Op if the file was closed. Close_file sets the status of a file descriptor to Cl.

Finally, the module include the declarations and the definitions of low level primitives to perform input/output operations. Read_file reads a buffer of data from a file, and Write_file writes a block of data into a file. Let’s note that Read_file and Write_file call Sys_read and Sys_write respectively; they are external procedures written in C.

8 Comparison

The linguistic evolution of parallel logic languages shows two different trends, one including languages based on dataflow-like streams, and one, more recent, including languages based on multiset rewriting. In the first set of languages entities with state (logic processes) are encapsulated
inside logic atoms; languages in the second set define an object as a multisets manipulated by “reactive” transformations.

The family of stream-based languages derived from Concurrent Prolog [28] is still under development, but after the Saraswat’s work [27], that introduced the theoretically appealing notion of Concurrent Constraint Logic Languages, it seems currently more geared toward the design of constraint-based systems than toward object-based systems. Moreover, the restriction of Concurrent Prolog to its flat subset constrains the language toward a synchronous style of programming, according to [29].

Contrariwise, a language based on multiset rewriting, like Gammalog, allows to avoid the semantic intricacies derived from “synchronous” communications based on streams. Activities that take place inside a multiset are totally asynchronous, and this assumption simplifies the design of programs.

Also from a pragmatic point of view a multiset is more manageable than a dynamic set of streams. For instance, suppose we have to use Concurrent Prolog in the design of a multiuser, distributed programming environment. The users should be seen as objects outputting streams of asynchronous requests; the description of their possible interactions become rapidly unmanageable and difficult to keep under control.

Contrariwise, the description of their coordination based in Gammalog is much simpler, because abstractly there is a unique communication channel. The users in such a case are simply producers of tuples in the multiset, and the system “reacts” whenever a coordination rule can be applied.

These languages are also well suited to be used as specification languages. In fact, Swarm [26]
is formalism based on a logic tuple space with a well-developed programming logic in the Unity style [24]. Swarm shows how a language based on transformations of multisets of tuples can be used for reasoning about properties of parallel systems.

Finally, the use of a formal semantics for describing the language adds a important tool since it helps in clarifying several implementation issues, and moreover can inspire the language designers, for instance suggesting new operators for enhancing coordination.

9 Conclusion and Future Work

We presented Gammalog: an integration between logic programming and the parallel language Gamma. We show that to achieve a full integration between the two paradigms, proving Gammalog complete and sound, we need to express reaction conditions as non-recursive logic programs. The implementation we describe is based on Gödel and in our knowledge is the first executable version of Gamma on a general purpose hardware. The language can be used as a specification language to explore the power of multiset rewriting in a logic programming framework.

Future work will be focused on the design of a parallel implementation of Gammalog; we are studying three sources of parallelism: multiset unification, “+” execution and parallel execution of sequential Prolog processes representing action predicates.

- Multiset unification can be performed in parallel in order to execute the same reaction condition on disjunct subsets of the initial multiset.
- The combination of two or more programs using “+” makes concurrent the execution of the steps of these programs.
- A Gammalog program can be translated into a Shared Prolog program, where actions are translated into sequential Prolog programs coordinated by rewriting rules as in Shared Prolog [9].

Acknowledgments

First and foremost, we would like to thank António Porto for interesting discussions and detailed comments on the contents of this work, that have contributed to improve the presentation and to fix some problems of a preliminary version of this paper. Moreover, we thank Paola Bruscoli, Eugenio Omodeo, Gianfranco Rossi, and Gianluigi Zavattaro for their remarks. Partial support for this work was provided by the Commission of European Communities under ESPRIT Programme Basic Research Project 9102 (COORDINATION).

References


EXPORT Files.
GPREDICATE Man_files. % This GAMMA predicate is the file manager.

Man_files = Crea_file + Open_file + Close_file + Read_file + Write_file.

LOCAL Files.
IMPORT Declarations, Strings.
BASE Status.
CONSTANT Cl, Op : Status. % Cl = Close; Op = Open
FUNCTION Open : Integer
        /!
        Elem;
Close : Integer
        /!
        Elem;
File : Integer*Integer*List(String)*Status*Integer
        /!
        Elem.
Create : Integer
        /!
        Elem;
Read, Write : Integer * List(String) * Integer
        /!
        Elem;

PREDICATE Crea_file, Open_file, Close_file, Read_file, Write_file : Multiset(Elem) * Multiset(Elem).
PREDICATE Open_file R1, Open_file R2 : Elem * Elem;
Open_file A1, Open_file A2 : Elem * Elem * Elem * Elem;
Crea R : Elem;
Close R, Read R, Write R : Elem * Elem;
Crea A, Close A : Elem * Elem;
Read A, Write A : Elem * Elem * Elem.

Open_file(<<x, y | rest>>), <<z, w | rest>>) = Open_file R1(x, y) | Open_file A1(x, y, z, w).
Open_file R1(Busy(a, [Open(fid)[p]]), File(a, fid, data, Cl, l)).
Open_file A1(Busy(a, [Open(fid)[p]]), File(a, fid, data, Cl, l), Busy(a, p), File(a, fid, data, Op, l)).

Open_file(<<x, y | rest>>), <<z, w | rest>>) = Open_file R2(x, y) | Open_file A2(x, y, z, w).
Open_file R2(Busy(a, [Open(fid)[p]]), File(a, fid, dati, Op, l)).
Open_file A2(Busy(a, [Open(fid)[p]]), y, Busy(a, p), y).

Crea_file(<<x | rest>>, <<y, z |rest>>) = Crea R(x) | Crea A(x, y, z).
Crea R(Busy(a, [Create(fid)[p]])).
Crea A(Busy(a, [Create(fid)[p]]), Busy(a, p), File(a, fid, [], Cl, 0)).

Close_file(<<x,y |rest>>, <<z |rest>>) = Close R(x, y) | Close A(x, y, z).
Close R(Busy(a, [Close(fid) | p]), File(a, fid, [], Op, l)).
Close A(Busy(a, [Close(fid) | p]), File(a, fid, [], Op, l), File(a, fid, [], Cl, l)).

Read_file(<<x, y |rest>>, <<z, w | rest>>) = Read R(x, y) | Read A(x, y, z, w).
Read R(Busy(a, [Read(fid,d,n)[p]]), File(a,fid,data,Op,l)).
Read A(Busy(a, [Read(fid,d,n)[p]]), File(a,fid,data,Op,l), Busy(a, p), File(a, fid, data, Op, newl)) =
        Sys_read(fid, l, d, n, newl).

Write_file(<<x,y | rest>>, <<z,w | rest>>) = Write R(x, y) | Write A(x, y, z, w).
Write R(Busy(a, [Write(fid,d,n)[p]]), File(a, fid, data, Op, l)).
Write A(Busy(a, [Write(fid,d,n)[p]]), File(a, fid, data, Op, l), Busy(a, p), File(a, fid, newdata, Op, newl)) =
        Sys_write(fid, l, d, n, newdata, newl).

Figure 7. Module Files