We describe a first order applicative language for the specification of deterministic systems of communicating computing agents à la Kahn-MacQueen. Both the sequential and parallel interpreter we give are based on lazy evaluation, are demand driven and can handle infinite streams and non-terminating procedures. An equivalent least fixed-point semantics is then presented which neatly copes with the above features of the language. It is worth noting that computations in our logical based model can be considered as formal proofs, thus making formal reasoning about programs easier.

1. INTRODUCTION

The aim of this paper is to give a model of distributed computing based on an applicative first order logic language. The communicating computing agents we describe are essentially those modelled by Kahn and MacQueen /12, 13/, i.e. processes whose data are streams and which exchange information through one-way (single writer and multiple readers) channels. Since we want our language to be applicative, the agents we model are stateless, and the basic computation step consists in a reconfiguration of a system of computing agents (SCA).

The language we propose is a modification of PROLOG /14/. Any SCA is modelled by a set of atomic formulas, each corresponding to a single agent. Distributed programs are represented as sets of procedurally interpreted Horn clauses, each being a process definition. Channels which connect two or more agents are modelled by the presence of the same variable in the corresponding atomic formulas. In order to limit ourselves to statically defined one-way channels, we restrict PROLOG by statically distinguishing the input arguments of an atomic formula from its output arguments. This restriction to PROLOG limits its expressive power as far as problem solving is concerned, e.g. it forbids invertibility. On the other hand, PROLOG has been extended to cope with infinite streams, which require our language to have non-strict processes and lazy constructors. In spite of the above outlined modifications, the simple and clear logical concepts which underly PROLOG need to be only slightly modified to provide a fixed-point semantics of our language. Remarkably enough, the semantics is defined in terms of a least fixed-point construction, even in the presence of non-terminating processes.
Finally, let us note that our language can be seen as a proper extension of term rewriting systems, when Horn clauses are interpreted as rewrite rules extended to provide more than one output. Thus, we argue that relevant properties and proof techniques of term rewriting systems, such as Church-Rosser property and Knuth-Bendix completion algorithm, can be generalized and used here.

In Sections 2 and 3 we will introduce the syntax of the language we use to talk about agents and channels. Then, we will discuss the behaviour of SCA's in terms of lazy system transformations. Finally, we will define a fixed-point semantics, related to the model theoretic semantics of logic theories.

2. BASIC SYNTACTIC CONSTRUCTS

Our language is a many sorted first order language, which allows to express the behaviour of SCA's in terms of rewriting systems.

The language alphabet is \( \mathcal{A} = \{ S, C, D, V, F, R \} \), where:

- \( S \) is a set of identifiers. Given \( S \), we define a sort \( s \) which is:
  i) simple if \( s \in S \),
  ii) functional if \( s \in S \rightarrow S \),
  iii) relational if \( s \in S \rightarrow S^* \).

- \( C \) is a family of sets of constant symbols indexed by simple sorts.

- \( D \) is a family of sets of data constructor symbols indexed by functional sorts.

- \( V \) is a family of denumerable sets of variable symbols indexed by simple sorts.

- \( F \) is a family of sets of function symbols indexed by functional sorts.

- \( R \) is a family of sets of predicate symbols indexed by relational sorts.

The basic construct of the language is the atomic formula, which corresponds to the agent, the basic component of a SCA.

An atomic formula is:

i) a data atomic formula of the form \( d(t_1,\ldots,t_m)=v \), such that \( t_1,\ldots,t_m \) are data terms of sorts \( s_1,\ldots,s_m \), \( v \) is a variable symbol of sort \( s \), and \( d \in D \) has sort \( s_1 \times \cdots \times s_m \rightarrow s \), or

ii) a functional atomic formula of the form \( f(t_1,\ldots,t_m)=v \), such that \( t_1,\ldots,t_m \) are data terms of sorts \( s_1,\ldots,s_m \), \( v \) is a variable symbol of sort \( s \), and \( f \in F \) has sort \( s_1 \times \cdots \times s_m \rightarrow s \), or

iii) a relational atomic formula of the form \( r(t_1,\ldots,t_m; out, v_{m+1},\ldots,v_n) \), such that \( t_1,\ldots,t_m \) are data terms of sorts \( s_1,\ldots,s_m \), \( v_{m+1},\ldots,v_n \) are variable symbols of sorts \( s_{m+1},\ldots,s_n \), and \( r \in R \) has sort \( s_1 \times \cdots \times s_m \rightarrow s_{m+1} \times \cdots \times s_n \).

A data term of sort \( s (s \in S) \) is:

i) a constant symbol of sort \( s \),

ii) a variable symbol of sort \( s \),

iii) a data constructor application \( d(t_1,\ldots,t_m) \) such that \( t_1,\ldots,t_m \) are data terms of sorts \( s_1,\ldots,s_m \), and \( d \in D \) has sort \( s_1 \times \cdots \times s_m \rightarrow s \).

A system formula (s-formula) is either:

i) an atomic formula, or