DISTRIBUTED LOGIC PROGRAMMING

ANTONIO BROGI AND ROBERTO GORRIERI*

We present a model for distributed logic programming based on AND-parallelism and on explicit message-passing primitives. The communication mechanism is inspired by Milner's CCS [31]. First, a simple calculus of communicating sequential logic processes is defined to formally introduce this notion of communication in a logic programming setting. Then, the definition of the language is completed by extending the calculus with an alternative command, as well as with mechanisms for the dynamic creation of AND-parallel processes. The resulting language is a variant of Distributed Logic defined by Monteiro [33].

The second part of the paper focuses on the study of the semantics of the language. We define a model-theoretic semantics by providing the various goal composition operators and the communication primitives with a clear logical meaning. On the other hand, an operational semantics is given in terms of the distributed model of Petri nets. The latter characterization is shown to provide several insights on the programming language, such as the ability of capturing fairness and liveness properties.

The study of the semantics terminates with the proof of the equivalence (soundness and completeness) between the model-theoretic and the operational semantics.

1. INTRODUCTION

In the process interpretation [43] of Horn Clause Logic [2, 29], a goal \( \leftarrow A_1, A_2, \ldots, A_n \) is viewed as a net of \( n \) processes communicating via shared variables, where the clauses of the program correspond to alternative definitions of the processes. Concurrent computations exploit nondeterministic choices among several alternatives (clause selection phase), and a reconfiguration of the net is the

* Dipartimento di Matematica, Università di Bologna, P. za Porta S. Donato, 5, I-48127 Bologna, Italy.

Address Correspondence to Antonio Brogi, Dipartimento di Informatica, Università di Pisa, Corso Italia, 40, 56125 Pisa, Italy. E-mail: brogi@di.unipi.it.

Received December 1989; accepted October 1992.

THE JOURNAL OF LOGIC PROGRAMMING

© Elsevier Science Publishing Co., Inc., 1993
655 Avenue of the Americas, New York, NY 10010 0743-1066/93/$6.00
result of applying a clause to an atomic goal. However, Horn Clause Logic (HCL) is not a concurrent language by itself. The simple communication mechanism provided by shared variables is not expressive enough to model real programming problems, where communication and concurrency must be explicitly controlled.

Most of the concurrency-oriented extensions of logic languages restrict the possible modes of the unification mechanism by introducing a distinction between the producers and the consumers of the values of a logical variable. The resulting model exploits a restricted form of AND-parallelism, named STREAM-parallelism. The best known representatives of this class of languages are PARLOG [11], Concurrent Prolog [43, 44], and GHC [48]. These languages (as well as HCL) are given an operational semantics by means of a centralized abstract machine [26, 37] based on interleaving models [31], where concurrency is expressed as a mere combination of sequentiality and nondeterminism. In the process interpretation view, when aiming at defining a distributed semantics, that is a semantics based on a distributed abstract machine (e.g., Petri Nets [39]), the first nontrivial problem to be faced is how to model mutual exclusive accesses to shared variables in a decentralized environment. To exploit the implicit parallelism of STREAM-parallel languages, it is necessary to move towards the implementation level, possibly resorting to sophisticated compilative techniques, e.g., based on abstract interpretation [1].

Recently, the concurrent constraint (cc) paradigm has been proposed by Saraswat [41, 42] as a general framework for concurrent (logic) programming based on the key notion of constraint. A cc system consists of a concurrent execution of agents which add (tell) and check (ask) constraints on a shared set of variables (store). The notions of unifiers and substitutions are replaced in this context by the more handy notions of constraint and conjunction. From the point of view of the resulting model of computation, the shared store of constraints introduces in cc languages the same kind of problems that shared variables cause to concurrent logic languages. Nevertheless, some efforts have been devoted to define operational semantics for cc languages based on distributed models. We shall discuss the limits of such an approach in Section 8.

The main concern of this paper is to study a model, and a corresponding actual language, for distributed logic programming. Traditional concurrent languages are partitioned into two classes according to the possible existence of a shared memory as a communication medium. The same motivations that suggested Hoare to introduce CSP [25] as opposite to Brinch-Hansen's Concurrent Pascal [6] guided us to decide to firmly prefer a message-passing model of communication for concurrent logic programming. Tightly connected to this debate is the above observation concerning the fact that all of the parallelism in STREAM-parallel languages is only potential. On the contrary, in the message-passing perspective, the parallel composition operator really expresses the obvious expectation that two goals in parallel can be independently solved on separate machines.

We first define a simple calculus of communicating sequential logic processes to formally introduce a message-passing model of communication in a logic programming setting. The basic underlying idea is to define a calculus where communication is neatly separated from deduction, and where concurrent processes are independent of one another. To illustrate these ideas, we simply consider a calculus where a set of sequential logic processes is executed in parallel and exchanges information through message-passing primitives.
We then complete the definition of the language by extending the calculus with an alternative command and with mechanisms for the dynamic creation of AND-parallel processes. In the literature, some authors [14, 22] have defined concurrency-oriented extensions of logic programming by introducing explicit operators for controlling parallelism and concurrency. In particular, we have been inspired by extensions of HCL with explicit operators like fork and join [14, 33], and with interface predicates for sending/receiving messages [34, 36]. The language we consider turns out to be a variant of Monteiro's Distributed Logic [33, 34] and of its operational counterpart Delta Prolog [13, 36]. Although there are some minor differences with Monteiro's language, we will refer to the language we study as Distributed Logic.

The main features of Distributed Logic are the following. First, the classical comma "," operator of HCL is replaced by two distinct operators: the sequential ";;" and the parallel composition "||" of goals. Second, message-passing primitives allow parallel processes to communicate. Finally, an alternative command is provided to control global nondeterminism.

The second part of the paper is devoted to a detailed study of the semantics of Distributed Logic. Extra-logical primitives enable the control of concurrency, but at the same time, they decrease the logical flavor of concurrent logic languages. Phenomena such as synchronization, communication, deadlock, and process creation can be more advantageously modeled using techniques stemming from imperative and functional concurrent languages, emphasizing control rather than logic. In fact, denotational and interleaving operational semantics have been defined for concurrent logic languages [3, 4, 24, 40]. Orthogonally, only recently some remarkable results have been achieved in order to define declarative semantics for concurrent logic languages [5, 21, 27, 28, 30], even if a completely satisfactory general treatment of this issue has not been achieved yet.

A declarative semantics—both model theoretic and fixpoint—is defined here in a simple and direct way for Distributed Logic. Extra-logical primitives enable the control of concurrency, but at the same time, they decrease the logical flavor of concurrent logic languages. Phenomena such as synchronization, communication, deadlock, and process creation can be more advantageously modeled using techniques stemming from imperative and functional concurrent languages, emphasizing control rather than logic. In fact, denotational and interleaving operational semantics have been defined for concurrent logic languages [3, 4, 24, 40]. Orthogonally, only recently some remarkable results have been achieved in order to define declarative semantics for concurrent logic languages [5, 21, 27, 28, 30], even if a completely satisfactory general treatment of this issue has not been achieved yet.

A declarative semantics—both model theoretic and fixpoint—is defined here in a simple and direct way for Distributed Logic. We extend the declarative approach of [23], where interpretations containing (possibly) nonground atoms are considered. A variable in an atom of a Herbrand interpretation is considered universally quantified and the atom belongs to the interpretation if and only if there exists a refutation for it with a substitution which does not instantiate its variables. The approach is very adequate for the declarative description of the operational behavior of logic languages since it preserves a close correspondence between correct and computed answer substitutions. The possibility of declaratively denoting exactly the operationally computed substitution seems very adequate to model both aspects of the language. Another key problem in declaratively modeling Distributed Logic is how to assign a logical meaning to communication predicates. The truth of a communication predicate does not depend on the values of its arguments, but rather on the presence of a matching complementary communication. Therefore, following [34], we will extend the Herbrand base and associate to an atomic formula A a sequence of communications, which represents a sequence of not matched communication instructions to be executed in a refutation of A.

A distributed, net-oriented operational semantics for a concurrent logic language is also presented here. Differently from those semantics based on intrinsically sequential interleaving models, we describe a concurrent system as a set of sequential processes, possibly located in different places, which cooperate in accomplishing a task. Thus, neither a global state nor a global clock must be
assumed. The semantics is given by means of a very popular model: 1-safe Place/Transition Petri Nets [39]. A place represents a process state type, while a token in a place is an activated instance of the process in that state. A flow relation connects places and transitions, and transitions transform the processes of the places connected on its incoming arcs into the processes of the places on its outgoing arcs. The places are the sequential subgoals, while the transitions specify how a sequential subgoal can perform a reduction step or even how two subgoals can communicate. Our operational semantics suggests a naive implementation of a Distributed Logic program on several processors, where each processor has a copy of the program and solves a sequential subgoal by exchanging information with other subgoals.

The study of the semantics terminates with the proof of the equivalence (soundness and completeness results) between the model-theoretic and the operational semantics.

The plan of the paper follows. We introduce a calculus of sequential logic processes in Section 2, and provide it with a simple operational semantics. The complete definition of Distributed Logic is given in Section 3, along with some programming examples. In Section 4, we discuss the problems of modeling Distributed Logic, both in the declarative and in the operational semantics. Sections 5, 6, and 7 contain a detailed study of the semantics of the language, as well as the proof of the equivalence of the declarative and the operational semantics. Finally, we draw some conclusions and discuss related work in Section 8.

2. COMMUNICATING SEQUENTIAL LOGIC PROCESSES

We first study a simple model of communication for logic programs, based on message-passing primitives, inspired by CCS [31]. We consider standard logic programming, where a program is a finite set of definite Horn clauses. The only extension is that clauses may contain message-passing primitives (called event goals) in the body. The concurrent execution of logic processes is specified through the parallel composition of goals. Let us formally introduce this calculus of Communicating Sequential Logic Processes (CSLP).

**Notation.** For standard notations of logic programming, we refer to [2, 29]. The language alphabet is $\langle F, V, P \cup Q \rangle$, where $F$ is a family indexed on natural numbers of a data constructor ($F_0$ is the set of constants), $V$ represents the set of variable symbols, $P$ the set of all predicate symbols, and $Q$ the set of (unary, post fixed) communication predicate symbols. More precisely, the set of names of communication channels is denoted by a set of constants $CH$ (disjoint from $F_0$), and $Q$ is defined as follows $Q = \{ ?e, !e | e \in CH \}$. The set of terms Term is defined as usual [2, 29]. A term is a variable, a constant, or an n-adic function applied to n terms. According to [23], the Herbrand universe $U$ is $U_V$, that is, $T_{F(V)} = \left( \text{the free } F\text{-algebra on } V \text{ modulo variance} \right)$, whose elements are denoted by $U$ (possibly indexed). The Herbrand base $B$ is a proper subset of $B_V$, composed of all program-defined predicate symbols applied to terms. We assume the reader is familiar with other standard notations such as substitution, most general unifier, or ground term.
Definition 2.1. The syntax of a CSLP program is given by the following BNF-like grammar. Let $A$ be an atomic formula, $U$ a term, and $e$ a channel name $(e \in CH)$.

\[
\begin{align*}
\text{Program} &::= A \leftarrow \text{SeqG} | \text{Program} \cup \text{Program} \\
\text{SeqG} &::= \text{true} | A | U?e | U!e | \text{SeqG};\text{SeqG} \\
\text{Parallel goals and queries are defined as follows:} \\
\text{ParG} &::= \text{SeqG} | \text{ParG} \parallel \text{ParG} | (\text{ParG}) \setminus e \\
\text{Query} &::= (\text{ParG}) \setminus CH
\end{align*}
\]

A program is a clause $(A \leftarrow \text{SeqG})$ or the union of two programs $(\text{Program} \cup \text{Program})$. A sequential goal can be true, an atom $A$, an event goal, or the sequential composition of two sequential goals. There are two types of event goals with the form, respectively, $U?e$ and $U!e$, where $U$ is a term (the message), $e$ is the event name (the communication channel), and “!” and “?” specify the communication mode. In order to solve an event goal $U?e$, a complementary event goal $U'!e$ has to be simultaneously solved. Both goals solve if and only if the terms $U$ and $U'$ unify. Every communication is due to the simultaneous execution of two complementary event goals, and the exchange of values is bidirectional. Thus, only symmetric, synchronous communications are allowed, in a style which is very similar to CCS, where the communication channel $e$ takes the place of a port name. The sequential composition operator “;” forces a left-to-right order on the evaluation of goals. Roughly, in the sequential composition $A_1; A_2$, goal $A_2$ is executed after $A_1$. A parallel goal is defined by composing sequential goals via the “||” operator. The initial configuration of the system is determined by a ParG. This means that in CSLP, the net of processes is fixed once and for all. The operator “\$e” restricts the behavior of its argument by forbidding asynchronous steps of event goals over channel $e$. Notice that the restriction operator can be equivalently applied to sets of channels since

\[
\begin{align*}
\text{ParG} \setminus \{e\} &= \text{ParG} \setminus e \\
\text{ParG} \setminus (E_1 \cup E_2) &= (\text{ParG} \setminus E_1) \setminus E_2 = (\text{ParG} \setminus E_2) \setminus E_1
\end{align*}
\]

Queries are always restricted with respect to all of the communication channels and are a syntactic subset of parallel goals. The CSLP calculus is a subset of the language Distributed Logic, proposed by Montiero [33].

In CSP [25], a fundamental property of parallel processes is their ability to work with a local environment. Such a property is obviously also interesting in the case of logic processes. Actually, if two parallel logic processes do not share variables, they can be executed independently of one another. The exchange of messages may, however, make independent processes share some variables. For example, consider the following simple case:

\[
\leftarrow (X!e;p(X)) || (Y?e;q(Y))
\]

We observe that the processes $(X?e;p(X))$ and $(Y?e;q(Y))$ are initially independent since they do not share variables. However, the complementary event goals $X?e$ and $Y?e$ can be solved with the substitution $[X = Y]$, thus obtaining the new parallel goal:

\[
\leftarrow p(Y) || q(Y)
\]

where the processes $p(Y)$ and $q(Y)$ share the variable $Y$. 

This is the reason why Monteiro [33] introduced in Distributed Logic the constraint that messages must become ground after a communication has been performed. To solve an event goal $U\forall e$, there must be a complementary event goal $U'\forall e$ to be simultaneously solved such that

1. $\exists \theta = \text{mgu}(U, U')$, and
2. both $(U)\theta$ and $(U')\theta$ are ground terms.

It is easy to see that condition ii) ensures that communication does not affect the independence of parallel processes. It is worth noting that, despite the former constraint, parallel processes may share variables in the original proposal of Distributed Logic [13]. Variables shared by parallel goals are considered different instances, and the compatibility of the computed substitution is checked at the end of the derivation of the goals. The computation proceeds only if the substitutions are compatible.

Here, we prefer to consider a purely distributed model for CSLP where AND-parallel processes cannot share variables, leaving the general case of sharing to Sections 4–7. We define a syntactic condition of well-formedness on parallel goals, which guarantees the independence of AND-parallel processes. Let $\text{Vars}(G)$ denote the set of variables occurring in $G$.

Definition 2.2. The well-formedness of a parallel goal $G$—denoted by $\text{wf}(G)$—is defined inductively as follows:

1. $\text{wf}(\text{SeqG})$
2. $\text{wf}(G_1\parallel G_2)$ if $\text{wf}(G_1)$ and $\text{wf}(G_2)$ and $\text{Vars}(G_1) \cap \text{Vars}(G_2) = \emptyset$
3. $\text{wf}(G \setminus e)$ if $\text{wf}(G)$.

The condition of well-formedness can be easily checked for any parallel goal to be executed. We will show (Proposition 2.6) that the property of well-formedness of a parallel goal is an invariant with respect to any CSLP computation.

2.1. Simple Operational Semantics

We formally describe the operational behavior of CSLP programs through a simple interleaving model, namely, a transition system. Let us introduce the standard definition of a labeled transition system [26].

Definition 2.3. A labeled transition system is a triple $\langle \Gamma, \Lambda, \rightarrow \rangle$, where $\Gamma$ is a set of configurations, $\Lambda$ a set of labels, and $\rightarrow \subseteq \Gamma \times \Lambda \times \Gamma$ is the transition relation.

We usually write $\gamma \xrightarrow{\lambda} \gamma'$ whenever $\langle \gamma, \lambda, \gamma' \rangle \in \rightarrow$.

Transition systems are essentially (possibly infinite) state automata. Approximately ten years ago, Plotkin [37] devised a new, powerful technique to transform an operational definition of (i.e., an abstract machine for) a programming language into a deductive system. According to his approach, called Structural Operational Semantics (SOS), the states of the abstract machine are formulas of the language, while the transitions are defined by means of a set of axioms and inference rules in a syntax-driven way. Verifying whether a transition exists in the machine corresponds to proving that the transition is a theorem of the deductive system. Thanks
to this technique, it is possible to define very concise and intuitive operational semantics for programming languages. The resulting abstract machine is a labeled transition system.

The standard operational semantics for HCL [20] corresponds to a refutation procedure using resolution as the only inference rule. Nonetheless, Horn Clause Logic can be treated as a programming language, and thus Plotkin’s SOS can be applied as well (e.g., see [40]).

We first define the sets of labels and configurations for sequential and parallel goals. We do not include the program into the configurations, and just leave it as an implicit parameter since it never changes during the refutation of the goal.

**Definition 2.4.** The sets of labels $\Lambda$ and configurations $\Gamma$ for goals are

$$\Lambda = \{ U!e, U?e \mid U \in \text{Term and } e \in CH \}$$

$$\Gamma = \{ \langle G, \vartheta \rangle \mid G \in \text{ParG and } \text{wf}(G) \text{ and } \vartheta \in \text{Sub} \} \cup \{ \langle \Box, \vartheta \rangle \mid \vartheta \in \text{Sub} \}$$

where $\text{Sub}$ is the set of all possible substitutions.

We denote by $\varepsilon$ the empty label, and $\rightarrow \subseteq \Gamma \times (\Lambda \cup \{ \varepsilon \}) \times \Gamma$ is the transition relation.

Transitions are possibly labeled by communications. When a sequential goal communicates, it assumes the existence of a partner for the communication whose existence will be checked only at composition time, and thus we store the name, the communication mode (! or ?), and the message $U$ of a not yet matched event goal. We will omit the empty label $\varepsilon$ when representing transitions, that is, $\langle \gamma, \varepsilon, \gamma \rangle \in \rightarrow$ will be simply written as $\gamma \rightarrow \gamma'$. Symbol $\alpha$ will be used to denote any element of $\Lambda \cup \{ \varepsilon \}$. Therefore, $\rightarrow$ will be used to represent both labeled and unlabeled transitions. A configuration $\langle G, \vartheta \rangle$ stands for a goal $G$ to be reduced, where the substitution $\vartheta$ encodes the values computed so far for the variables used in the computation. The configuration $\langle \Box, \vartheta \rangle$ stands for the special stuck configuration “nothing to do.”

We introduce the function $ch$, which gives the channel name possibly contained in a label, and is defined as follows:

$$ch(\alpha) = \begin{cases} e & \text{if } \alpha = U!e \text{ or } \alpha = U?e \\ \varepsilon & \text{if } \alpha = \varepsilon \end{cases}$$

We are now in a position to give the set of transition rules modeling the behavior of CSLP programs.

**Definition 2.5.** The derivation relation over configurations $\rightarrow \subseteq \Gamma \times (\Lambda \cup \{ \varepsilon \}) \times \Gamma$ is defined as the least relation satisfying the following axioms and inference rules:

**True**

$$\langle \text{true}, \vartheta \rangle \rightarrow \langle \Box, \vartheta \rangle$$

**Atomic**

$$\exists A_2 \leftarrow G \in P \land \sigma = \text{mgu}(A_1, A_2)$$

$$\langle A_1, \vartheta \rangle \rightarrow \langle (G) \sigma , \vartheta \sigma \rangle$$
Event

\[
\langle U!e, \varnothing \rangle \xrightarrow{U!e} \langle \square, \varnothing \rangle
\]

\[
\langle U?e, \varnothing \rangle \xrightarrow{U?e} \langle \square, \varnothing \rangle
\]

Seq

\[
\langle G_1, \delta \rangle \xrightarrow{a} \langle \square, \varnothing \rangle
\]

\[
\langle (G_1; G_2), \delta \rangle \xrightarrow{a} \langle (G_2), \delta \rangle
\]

\[
\langle G_1, \delta \rangle \xrightarrow{a} \langle (G_1), \varnothing \rangle
\]

\[
\langle G_1; G_2, \delta \rangle \xrightarrow{a} \langle G_1^1; G_2, \varnothing \rangle
\]

Par

\[
\langle G_1, \delta \rangle \xrightarrow{a} \langle G_1', \varnothing \rangle
\]

\[
\langle G_1 || G_2, \delta \rangle \xrightarrow{a} \langle G_1' || G_2, \varnothing \rangle
\]

\[
\langle G_1, \delta \rangle \xrightarrow{a} \langle G_1', \varnothing \rangle
\]

\[
\langle G_2 || G_1, \delta \rangle \xrightarrow{a} \langle G_2 || G_1', \varnothing \rangle
\]

Sync

\[
\langle G_1, \delta \rangle \xrightarrow{U!e} \langle G_1', \delta \rangle \land \langle G_2, \delta \rangle \xrightarrow{U?e} \langle G_2', \delta \rangle \land \\
\sigma = mgu(U_1, U_2) \land (U_1 \sigma, U_2 \sigma) \text{ is ground}
\]

\[
\langle G_1 || G_2, \delta \rangle \xrightarrow{a} \langle (G_1' || G_2') \sigma, \delta \sigma \rangle
\]

\[
\langle G_1, \delta \rangle \xrightarrow{U!e} \langle G_1', \delta \rangle \land \langle G_2, \delta \rangle \xrightarrow{U?e} \langle G_2', \delta \rangle \land \\
\sigma = mgu(U_1, U_2) \land (U_1 \sigma, U_2 \sigma) \text{ is ground}
\]

\[
\langle G_2 || G_1, \delta \rangle \xrightarrow{a} \langle (G_2' || G_1') \sigma, \delta \sigma \rangle
\]

Join

\[
\langle \square || \square, \delta \rangle \xrightarrow{a} \langle \square, \varnothing \rangle
\]

Res

\[
\langle G, \delta \rangle \xrightarrow{a} \langle G', \delta \rangle \land ch(\alpha) \neq e
\]

\[
\langle G \setminus e, \delta \rangle \xrightarrow{a} \langle G' \setminus e, \varnothing \rangle
\]

Remarks. Rule \textbf{true} says that the formula true always leads to the stuck configuration “nothing to do” without changing the substitution computed so far. Rule \textbf{Atomic} describes the standard application of a clause to an atomic goal. Rule \textbf{Event} deals with event goals, and states that any event goal can be executed independently of the external environment. Rule \textbf{Seq} states that the first component of a sequential goal must be solved first, and that the computed substitution is passed to the second component. Rule \textbf{Par} states that if either of the components of a parallel goal can make a sequential step, then the parallel goal also can perform the same step by leaving the other component of the parallel goal idle during the
transition. Notice that the label of the transition is ignored. This implies that event goals can be executed independently, without any implicit synchronization. Moreover, the substitution computed by the active goal is not passed to the other goal. This relies on the well-formedness of parallel goals. Rule Sync defines the way processes synchronize via the synchronous execution of complementary matching event goals, while rule Join synthesizes successful branches of the derivation. Finally, rule Res deals with the restriction on parallel goals. Every transition labeled by a nonempty label (either U!e or U?e) is inhibited if the channel name is e. Notice that since queries are restricted with respect to the whole set of channels CH, communications cannot be exported outside, but must be solved within the goals composing the query.

We now show that the property of well-formedness (Definition 2.2) of a parallel goal is invariant during a computation. More precisely, all of the resolvents of any derivation starting from a well-formed goal are well-formed.

**Proposition 2.6.** For every well-formed goal \( G \), if \( \langle G, \varepsilon \rangle \rightarrow \langle G', \delta \rangle \), then \( G'\delta \) is a well-formed goal.

**Proof.** Immediate by structural induction following Definitions 2.5 and 2.2.

Notice that the above proposition leads to a simple but efficient implementation of the language, as parallel processes can be executed in parallel, independently of one another, possibly on a distributed architecture.

We can finally define the notion of successful derivation. In the following, for any substitution \( \delta \), we denote by \( \delta_G \) the restriction of \( \delta \) to the variables of \( G \).

**Definition 2.7.** Given a program \( P \), a goal \( G \) can be successfully derived in \( P \) with answer substitution \( \delta \) if and only if \( \exists \delta: \langle G, \varepsilon \rangle \rightarrow^* \langle \square, \delta \rangle \) and \( \delta = \delta_G \).

2.2. The Producers and Consumers Example

To illustrate the use of CSLP, we consider the well-known programming example of producers and consumers. Roughly, the problem is to specify a communication protocol between two kinds of processes, where some of them produce messages which are consumed by the others.

We first show how this problem can be solved in the concurrent constraint (cc) framework [41, 42]. According to the cc paradigm, a program consists of a concurrent execution of agents which add (tell) and check (ask) constraints on a shared set of variables (store), and whose behavior is described by a set of clauses. Two clauses suffice to specify the synchronization of producers and consumers:

(i) \( \text{producer}(X) \leftarrow \text{true}:X = [Y|Ys] \mid \text{produces}(Y), \text{producer}(Ys) \)

(ii) \( \text{consumer}(X) \leftarrow X = [Y|Ys]:\text{true} \mid \text{consumes}(Y), \text{consumer}(Ys) \)

Consider the initial goal

\( \leftarrow \text{producer}(Z), \text{consumer}(Z) \)

corresponding to the scenario with exactly one producer and one consumer. According to the cc paradigm, clause (i) defining the producer process can apply to the current goal by asserting (i.e., telling) the constraint \( X = [Y|Ys] \) to the store. If
the constraint is consistent with the current store, then the clause is applied and produces(Y), producer(Ys) is added to the current goal. On the other hand, the consumer process checks whether the constraint X = [Y|Ys] is entailed by the current store. If this is the case, the computation rewrites into consumes(Y), consumer(Ys). The definition of predicates produces and consumes is omitted, and corresponds to some (sequential) computation producing (respectively, consuming) a message.

Some remarks are worth making here. The cc paradigm supports an elegant programming style where synchronization steps are syntactically distinguished from deduction steps. Actually, the synchronization of the agents with the store are all encapsulated into the ask & tell part of a clause. However, at every step of the computation, that is, at each clause application, every process aimed at rewriting itself must synchronize with the shared store. This obviously introduces a centralization point in the resulting model of computation, which may generate a bottleneck, as in the case of concurrent logic languages. Moreover, in principle, there is no reason for requiring that every clause application must synchronize with the store. For example, the production of a message (produces(Y)) can be carried on by a process independently of the external environment (viz. the other processes). In other words, there is no reason for the process calculating produces(Y) to synchronize with the others since its execution does not depend on the external environment.

The producers/consumers problem can be programmed in CSLP in the following way. According to the specification of the problem, producers and consumers are expected to work concurrently. This implies that producer and consumer processes are to be composed by means of the parallel composition operator ("||"). Moreover, as we have previously observed, the producer and the consumer process do not need to share variables. The only exchange of values which takes place can be expressed by mean of explicit message-passing primitives.

(iii) producer ← produces(X); X!ch; producer
(iv) consumer ← Y?ch; consumes(Y); consumer

An initial goal of the form

← producer|consumer

corresponds to a configuration with one producer and one consumer. The intended meaning of clauses (iii) and (iv) is the following. The producer process produces a message X by means of some sequential computation produces(X), and then sends the message X on the channel ch (to a consumer process). Finally, the producer process restarts itself. On the other hand, the consumer process waits to receive a message on the channel ch, consumes it, and restarts itself.

It is worth observing that CSLP message-passing primitives are synchronous. This means that, according to clauses (iii) and (iv), neither the producer nor the consumer can proceed with the computation before the synchronization has been performed. However, as in the case of CSP [25], asynchronous communications can be specified by means of a buffer process.

To illustrate the operational behavior of CSLP, let us look at an example of
calculation of the operational semantics. Consider the parallel goal:

\( \left< \text{producer} \parallel \text{consumer}, \varepsilon \right> \)

\( \rightarrow \) (by rule Atomic and (iii))

\( \left< \text{produces}(X); X!\text{ch}; \text{producer} \right> || \text{consumer}, \varepsilon \)

\( \rightarrow \) (by rule Atomic and iv))

\( \left< \text{produces}(X); X!\text{ch}; \text{producer} \right> || \left< \text{Y}?!\text{ch}; \text{consumes}(Y); \text{consumer} \right>, \varepsilon \)

\( \rightarrow \) (by assuming that \( \left< \text{produces}(X), \varepsilon \right> \rightarrow \left< \square, \{X = \text{ml}\} \right> \))

\( \left< \text{ml}!\text{ch}; \text{producer} \right> || \left< \text{Y}?!\text{ch}; \text{consumes}(Y); \text{consumer} \right>, \{X = \text{ml}\} \)

\( \rightarrow \) (by rule Sync)

\( \left< \text{producer} \parallel \left< \text{consumes}(\text{ml}); \text{consumer} \right>, \{X = \text{ml}, Y = \text{ml}\} \right> \)

\( \rightarrow \) (by assuming that \( \left< \text{consumes}(\text{ml}), \{X = \text{ml}, Y = \text{ml}\} \right> \rightarrow \left< \square, \{X = \text{ml}, Y = \text{ml}\} \sigma \right> \)) \( \left< \text{producer} \parallel \text{consumer}, \{X = \text{ml}, Y = \text{ml}\} \sigma \right> \)

\( \rightarrow \)

... and so on.

3. A LANGUAGE FOR DISTRIBUTED LOGIC PROGRAMMING

The CSLP calculus has served for introducing the basic concepts of a distributed model for logic programming. However, CSLP lacks some of the basic mechanisms which are essential for any concurrent language. In this section, we extend the calculus with mechanisms for both the dynamic creation of processes and the explicit control of nondeterminism.

The CSLP calculus does not allow the dynamic creation of processes. This is a severe limitation for a concurrent language. Consider again the example of producers and consumers:

(iii) producer \( \leftarrow \) produces(X); X!ch; producer

(iv) consumer \( \leftarrow \) Y?ch; consumes(Y); consumer

As we have seen, the consumer process receives a message and consumes it. However, the consumer cannot receive any other message before it has completed the processing of the last received message. As far as a system consisting of only one producer and one consumer is concerned, the program works fine. However, such a behavior can induce a bottleneck in the computation as soon as, for example, there are more producers than consumers. In such a situation, there will most probably be several producers waiting for the consumer, while it would be very useful to have the possibility of dynamically creating new processes. Actually, the consumer process can be rewritten as follows:

(v) consumer \( \leftarrow \) Y?ch; (consumes(Y)||consumer)

In this way, the consumer, after receiving a message, can restart itself while launching in parallel a process which will consume the message. It is worth noting
that the dynamic creation of AND-parallel processes introduces the possibility of having nonindependent AND processes, even if the initial goal is well-formed. These kinds of problems have been discussed in [7]. Notice, however, that we are going to consider a variant of Distributed Logic in which AND-parallel goals are allowed to share variables according to [13].

Another serious limitation of the CSLP calculus is the lack of an alternative command to express nondeterministic choices depending on communications with other processes. Besides the internal nondeterminism when selecting one of the unifiable clauses for an atomic goal, another form of nondeterminism is considered in Distributed Logic. A choice goal represents a nondeterministic process where the selection between its two immediate constituent subgoals is driven by the external environment, as each one is guarded by an event goal. Proceeding with the example of producers and consumers, consider the case in which the consumer process can receive messages on two different channels. We expect the consumer process to be capable of receiving messages on either channel, and of receiving a message when there is at least one message on one of the channels. Such a behavior cannot be obtained by a CSLP program since the consumer process can only wait to receive message either on one channel or on the other. The introduction of a choice composition operator (denoted by "<>") allows us to tackle the problem:

\[
\begin{align*}
\text{(vi) consumer} & \leftarrow (Y?ch1; (\text{consumes(Y)} | \text{consumer}))) \\
& \diamond \\
& (Y?ch2; (\text{consumes(Y)} | \text{consumer}))
\end{align*}
\]

Notice that, differently from the previous case, the introduction of the choice composition operator does not affect the well-formedness of goals. It has not been included in the presentation of CSLP only for the sake of simplicity.

In the next subsection, we introduce the formal syntax of Distributed Logic by extending the CSLP calculus with both parallel and choice composition operators.

3.1. Syntax

The abstract syntax of Distributed Logic (DL) is defined as an extension of HCL with some operators of goal composition which resemble those of more traditional concurrent languages such as CCS [31] and CSP [25]. In DL, a neat syntactical distinction is introduced between the sequential ";" and parallel composition "||" of goals. The operator for sequentialization ";" simply forces an order on goal derivations, while the operator for parallel composition states that the two subgoals are to be independently solved. The meaning of the HCL operator ";" cannot be immediately turned to either one. It is neither a sequential composition operator since the success set does not depend on the order in which atoms in a goal are to be reduced (i.e., the computation rule), nor is it a parallel operator since atoms cannot be independently derived in parallel because of shared variables. A nondeterministic choice operator and interface predicates for communication complete the set of the language primitives.

Definition 3.1. The abstract syntax of DL is given by the following BNF-like grammar. Let \(A\) be an atomic formula, \(U\) a term, and \(e\) a channel name.
P ::= A ← G | P ∪ P
G ::= true | A | Ge | Gc | G ; G | G || G | G \ e
Ge ::= U ? e | U ! e
Gc ::= Ge ; G | Ge \ Gc

We denote by

Pr the set of programs, ranged over by P
G the set of goals, ranged over by G
Ge the set of event goals, ranged over by Ge
Gc the set of choice goals, ranged over by Gc.

A goal G is a parallel goal iff it has one of the following forms:

1. G = G1 || G2
2. G = G' ; G'' where G' is a parallel goal
3. G = G' \ e where G' is a parallel goal.

Otherwise, G is a sequential goal.

Queries have the form

Query ::= G \ CH

Let us observe the differences with respect to the definition of the syntax of CSLP (Definition 2.1). A clause has the form A ← G instead that A ← SeqG, where G can be true, an atom A, an event goal Ge, a choice goal Gc, the sequential or the parallel composition of two goals, or the restriction of a goal. As in CSLP, queries are always restricted with respect to all communication channels.

Notice that in DL the condition of well-formedness on goals (Definition 2.2) is omitted. Thus, according to Monteiro's proposal, parallel goals may share variables. In other words, the purely distributed model of CSLP is a particular case of the DL model.

3.2. A Programming Example

We now give an example of a program to illustrate the expressive power of Distributed Logic. The interested reader can find several other examples in [13, 33, 34, 36].

Airline Reservation System. A standard benchmark for measuring the expressive power of concurrent languages is the so-called airline reservation system. Suppose that there are n travel agencies connected with an airline company on a computer network. The travel agencies accept reservations from customers via some user interface, and send requests of reservations to the airline company. After sending a request (on channel request), the agency process creates two AND processes. One process waits for the answer from the company (on channel answer), while the other one recalls the agency process which will be able to accept new requests from customers.

agency(X) ← read(R);
    msg(R,X)!request;
    (msg(Ans,X)?answer ; write(Ans))∥agency(X))
Notice that the definition of the agency process is parametric with respect to the name of the agency (X), so that the same code is used for arbitrarily many agencies.

The company manages a database containing all of the information concerning the flights (flight number, available seats, and so on). The company receives a request from an agency, processes it (by possibly modifying the database of the company), and sends the answer to the agency. Moreover, the company may also receive instructions to update the database from the administrator of the company (on channel internal). This behavior is modeled by using the alternative command as follows:

\[
\text{company(DB) } \leftarrow \text{msg(R,X)?request;}
\]
\[
\text{process(R,} \text{DB,} \text{NewDB,Ans);}
\]
\[
\text{(msg(Ans,X)!answer||company(NewDB))}
\]
\[
\diamond
\]
\[
\text{msg(Update)?internal;}
\]
\[
\text{update(DB,Update,NewDB);}
\]
\[
\text{company(NewDB)}
\]

Notice that, as in the case of the agencies, the company is capable of sending the answer to a request and processing in parallel new requests. The initial configuration of network consisting of the company together with \( n \) agencies and the administrator of the company is the following:

\[
\leftarrow \text{company(db)||agency(a1)||...||agency(an)||administrator}
\]

4. MODELING DISTRIBUTED LOGIC

This section is devoted to discussing the main problems of modeling Distributed Logic with respect to both the declarative and the operational semantics, and it can serve as a suitable introduction to Sections 5 and 6.

One of the distinguishing features of pure logic programming is its clear semantics. Logic programs can be characterized by model-theoretic, fixpoint, and operational semantics, which have been proved to be equivalent to one another [20]. In the rest of the paper, we focus on the study of the semantics of Distributed Logic and provide it with the three above-mentioned characterizations.

4.1. Declarative Semantics

A crucial issue in the definition of extensions of logic programming is whether or not the semantics of these extensions can be defined through simple extensions of the standard semantics given for the logic programming kernel. As far as the model-theory is concerned, the minimal Herbrand model of a program is usually taken as the intended meaning of a logic program [20]. When aiming at defining a model-theoretic semantics for some extension of logic programming, new syntactic constructs must be provided with a suitable logical meaning. As pointed out in [10], in the case of Distributed Logic, the main issues to be faced are different goal composition operators and event goals.
When logically interpreting a Horn program, an atomic formula A is true if it belongs to the minimal Herbrand model of the program. The unique goal composition operator is the conjunction "\(^\lor\)". The truth relation simply states that a conjunction of formulae \(A_1,\ldots,A_n\) is true if every \(A_i\) is true (i.e., if it belongs to the minimal model of the program). The situation is more complex in the case of Distributed Logic, where there are several goal composition operators (see Section 3). For instance, if we consider the choice operator, we expect that the goal \(G\lor G'\) will be true if either G or G' is true. Therefore, the truth relation must be suitably extended to model different goal composition operators.

The other relevant problem in defining a declarative semantics for Distributed Logic is to understand the logical meaning of communications (event goals). They are not normal user-defined predicates since they cannot be redefined in the program, that is, they cannot appear on the left-hand side of any clause. From this point of view, event goals resemble system predicates. However, they cannot be considered system predicates since the logical meaning of a communication predicate does not depend only on the values of its arguments, but also on the (possible) presence of a matching complementary event. Such predicates are not true or false by their own. On the other hand, their existence as syntactical entities strongly suggests including them into the Herbrand base. However, in this way, the Herbrand base would contain atoms which do not have a well-defined logical meaning, and it would be unclear what an interpretation should be. We therefore consider a proper subset of the Herbrand base, containing program-defined predicate atoms only, and relate the truth and the falsity of a goal with the history of the open communications (i.e., not yet matched event goals), following [34]. A goal can be true for some histories and false for others, so that event histories must be taken into account as part of the model. In our declarative semantics, event histories are modeled by finite sequences of events, called traces. To be more precise, consider the following program:

```
p ←
q ← u!e
r ← u?e
```

Intuitively, the minimal model of the program should be

\[
\{ \langle p, \lambda \rangle, \langle q, [u!e] \rangle, \langle r, [u!e] \rangle \} \]

where square brackets enclose traces and \(\lambda\) denotes the empty trace. The intended meaning of \(\langle q, [u?e] \rangle\) is that atom q is true with trace \([u?e]\), which represents the sequence of not matched communications of q. Differently from [22], only atomic goals are part of the model. The truth of composed goals is defined in terms of its atomic components. In the example, the goal \(q\lor r\) is true with the empty trace since it can be solved with a communication between q and r, as well as for the traces \([u?e, u!e]\) and \([u!e, u?e]\) since q and r may also choose not to communicate with each other, but rather with some external partner. Notice that, in order to describe other properties beyond the membership to the success set (e.g., causality), more complex histories should be considered (e.g., partially ordered multiset).

Since the purpose of the present work is to model the semantical properties of Distributed Logic, we are interested in establishing a tight relation between the operational behavior and the declarative semantics of a program. For Horn Clause Logic, the equivalence between these two different kinds of semantics has been
proved by considering ground atoms only. The correspondence is sound and complete: the minimal Herbrand model $M$ of a program $P$ coincides with its success set $[20]$. In any event, the classical notion of success set

$$SS = \{p(t) | p(t) \in HB \text{ and } \exists \theta \text{ such that } \leftarrow p(t) \xrightarrow{\theta} * \square \}$$

hides one of the fundamental aspects of logic programs: the ability to compute substitutions. A more adequate definition could be

$$SS' = \{(p(t), \theta') | p(t) \in B_v, \exists \theta \text{ such that } \leftarrow p(t) \xrightarrow{\theta} * \square \text{ and } \theta_{p(t)} = \theta'\}$$

but, in this case, the equivalence with the minimal model is lost.

In [23], a new interesting approach to the definition of declarative semantics based on interpretations containing (possibly) nonground atoms is introduced. That semantics fully characterizes the set $SS'$, thus establishing a tight correspondence between the operational and the declarative semantics. Any correct answer substitution (a substitution found via the declarative method) is also a computed answer substitution (operationally computed by refutation), and vice versa. We extend this approach to cope with the concurrency-oriented operators of Distributed Logic.

4.2. Operational Semantics

The study of the operational semantics is mainly devoted to substantiate the claim that Distributed Logic is a distributed logic language. We will present a distributed semantics for the language, defined in terms of Petri nets. Such a modelization allows us to study nontrivial computational properties of programming languages, such as fairness and liveness properties.

Part of the success of concurrent logic languages derives from the wide range of applications they proved to address. Operating systems [46] and other real-time applications cannot be modeled simply by means of the notion of success set because, in never-ending programs, the goal will never be refuted. Indeed, in such cases, interest has to be focused on the whole behavior of the concurrent system. Deadlock and fairness are typical examples of behavioral properties which refer to the history of the computational process to be represented. From an operational point of view, these properties can be better investigated within a distributed model (see [18] for a discussion about this topic), also stressing the distribution nature of DL. Let us introduce the general methodology for defining a distributed abstract machine for DL.

A powerful extension to Plotkin’s technique consists of substituting a distributed abstract machine, such as a Petri Net [39], for the target centralized labeled transition system. This interesting result is reported, for instance, in [15, 16, 18]. Other related papers are [17, 35, 47]. The basic ideas of the approach are the following.

1. Decompose syntactically a term $P$, representing a concurrent program, into a set of sequential components $[C_1, C_2, \ldots, C_n]$, which can be thought of as working concurrently. Sequential components denote the places of the Petri net.

2. Define net transitions following the SOS technique on (possibly several) sequential components so that they assume the form

$$\{C_1, C_2, \ldots, C_n\} \rightarrow \{D_1, D_2, \ldots, D_n\}.$$
Here, for the first time, this technique is extended for, and applied to, a concurrent logic language in order to fully express its parallelism via a distributed model.

5. DECLARATIVE SEMANTICS

We denote by $C$ the set of all ground event goals, that is,
\[ C = \{ u!e, u?e \mid \text{u is a ground term and } e \in CH \}. \]

ranged over by $c$, possibly indexed. Let $\overline{c}$ denote the complementary event of $c$ (i.e., if $c = u!e$, then $\overline{c} = u?e$, and if $c = u?e$, then $\overline{c} = u!e$).

We introduce the notion of trace of communications and define two operations of compositions on traces.

**Definition 5.1.** A trace is any finite sequence of elements from $C$. The concatenation of two traces is their juxtaposition. The empty trace $\lambda$ is the identity of trace concatenation (i.e., for any trace $t$: $t \cdot \lambda = \lambda t$). The merging $t_1 \cdot t_2$ of two traces $t_1$ and $t_2$ is the set of traces $t_1 \cdot t_2 = \{ tt \in t_1 \otimes t_2 \}$ where $\otimes$ is defined as follows:
\[
\begin{align*}
Ct_1 \otimes Ct_2 &= \{ c_1(t_1 \otimes c_2t_2) \} \cup \{ c_2(c_1t_1 \otimes t_2) \} \cup \{ t_1 \otimes t_2 \mid \lambda = \lambda \otimes t. \}
\end{align*}
\]

The restriction of a trace $t$ with respect to a communication channel $e$ is denoted by $t \setminus e$. The following two equations define the cases when the restriction operation does not affect a trace.

\[
\begin{align*}
(ct) \setminus e &= c(t \setminus e) \text{ if } (c = u!e' \text{ or } c = u?e') \text{ and } e \neq e' \\
\lambda \setminus e &= \lambda
\end{align*}
\]

The set of all traces is denoted by $Tr$.

Note that if $e$ does not occur in $t$, then $t \setminus e = t$.

Notice that while the concatenation of traces is a deterministic function, the merging of two traces is nondeterministic and produces a set of possible results. For example, consider the traces $t_1 = [u?e, u'?e']$ and $t_2 = [u!e]$. We have
\[
\begin{align*}
t_1t_2 &= [u?e, u'?e', u!e] \\
t_1 \cdot t_2 &= \{ [u?e, u'?e', u!e], [u?e, u!e, u'?e'], [u!e, u?e, u'?e'], [u'?e'] \}.
\end{align*}
\]

Roughly, trace concatenation corresponds to the sequential composition of processes, while trace merging corresponds to parallel composition, where parallel processes may communicate and close some of their open communications.

As previously mentioned, we consider the case in which parallel goals can share variables, and we introduce a notion of compatibility among substitutions. In DL, variables shared by parallel goals are treated as different variables, and the compatibility of the independently computed substitutions is checked at join time. We adopt the definition of parallel composition of substitutions introduced in [5].

**Definition 5.2.** A substitution $\sigma$ is less defined than $\delta$ ($\sigma \leq \delta$) if $\exists \delta: \sigma \delta = \delta$. Analogously, given two atoms $A, A'$, we set: $A \leq A'$ if $\exists \delta: A \delta = A'$.

The parallel composition $\delta_1 + \delta_2$ of two substitutions $\delta_1$ and $\delta_2$ is the minimal substitution $\delta$ such that $\delta_1, \delta_2 \leq \delta$, if any. Moreover, $\sigma$ is a substitution unifier of $\delta_1$ and $\delta_2$ if and only if $\delta_1 \sigma = \delta_2 \sigma$. 
In [19], it is shown that \( \vartheta_1 + \vartheta_2 = \text{mgu}(S(\vartheta_1) \cup S(\vartheta_2)) \), where \( S(\vartheta) \) is the set of equations \( x = t \) for each binding \( x/t \) in \( \vartheta \). Then
\[
\vartheta_1 + \vartheta_2 = \text{mgu}(S(\vartheta_1) \cup S(\vartheta_2)) = \vartheta_1\text{mgu}(S(\vartheta_2) \vartheta_1) = \vartheta_2\text{mgu}(S(\vartheta_1) \vartheta_2).
\]

Notice that if \( \vartheta_1 + \vartheta_2 \) is defined, then an \( SU \) \( \sigma \) for \( \vartheta_1 \) and \( \vartheta_2 \) always exists. In fact, the two sets \( S(\vartheta_1) \vartheta_2 \) and \( S(\vartheta_2) \vartheta_1 \) denote the same set of equations, where for each equation \( t = t' \) in the former set, \( t' = t \) appears in the latter, and vice versa. Up to variable renaming, a substitution unifier is unique. We will call it \( the \) \( SU \).

Example 5.3. Given two substitutions \( \vartheta_1 = \{X/2, Y/f(S, Z)\} \) and \( \vartheta_2 = \{X/W, Y/f(T, 2)\} \),
\[
\vartheta_1 + \vartheta_2 = \text{mgu}(\{X = 2, Y = f(S, Z)\} \cup \{X = W, Y = f(T, 2)\}) = \vartheta_1\text{mgu}(\{2 = W, f(S, Z) = f(T, 2)\}) = \vartheta_2\text{mgu}(\{W = 2, f(T, 2) = f(S, Z)\}) = \vartheta_1\sigma
\]
where the substitution unifier \( \sigma \) is \( [Z/2, S/T, W/2] \).

5.1. Model-Theoretic Semantics

We introduce the definitions of interpretation domain, interpretation, and truth for a Distributed Logic program \( P \). Recall that (Section 2) the Herbrand base \( B \) is a proper subset of \( B_v \), consisting of all program-defined predicate symbols applied to terms.

Definition 5.4. The domain \( H \) of interpretations is defined as follows:
\[
H = \{\langle A, t \rangle | A \in B \text{ and } t \in Tr \}.
\]

An interpretation \( I \) is any subset of \( H \).

The definition of truth we are going to give is inherently compositional. The idea is to reduce the truth of a goal to the truth of its atomic components. For each goal constructor, we define a rule for determining the logical value inductively on its syntactical structure. The basis of the induction consists of checking whether a program-defined atom belongs to the model and taking the event goals just as they are. For the sake of simplicity, we will omit some minor details in Definition 5.5, and consider only ground communications. In Section 7, we will come back to this point and add the necessary details to the definition.

Definition 5.5. Let \( I \) be an interpretation for a program \( P \). Then
\[
\begin{align*}
&\text{A unit clause } A \leftarrow \text{ is true in } I \quad \text{iff} \quad \langle A, \lambda \rangle \in I \\
&\text{A clause } A \leftarrow G \text{ is true in } I \quad \text{iff} \quad \forall G' \forall \langle \vartheta, t \rangle \text{ s.t. } \langle \vartheta, t \rangle \in \text{s-mgu}_I(G, G') \quad \text{then } \langle A \vartheta, t \rangle \in I \\
&\text{A goal } G \text{ is true with trace } t \quad \text{iff} \quad \exists G' \text{ s.t. } \langle \vartheta, t \rangle \in \text{r-mgu}_I(G, G')
\end{align*}
\]
where relation \( \text{s-mgu}_I \) is defined as follows:
\[
\langle \vartheta, t \rangle \in \text{s-mgu}_I(A, A') \quad \text{iff} \quad \langle A', t \rangle \in I \quad \text{and} \quad \vartheta = \text{mgu}(A, A')
\]
• \( \langle e, (u!e) \rangle \in s\text{-}mgu_1(u!e, u?e) \) if \( u \) is a ground term

• \( \langle e, (u?e) \rangle \in s\text{-}mgu_1(u?e, u!e) \) if \( u \) is a ground term

• \( \langle \vartheta_1 \vartheta_2, t_1 t_2 \rangle \in s\text{-}mgu_1(G_1, G_2) \) if \( \langle \vartheta_1, t_1 \rangle \in s\text{-}mgu_1(G_1, G'_1) \) and \( \langle \vartheta_2, t_2 \rangle \in s\text{-}mgu_1(G_2, G'_2) \)

• \( \langle \vartheta, t \rangle \in s\text{-}mgu_1(G_1 \cup G_2, G'_1 \cup G'_2) \) if \( \langle \vartheta_1, t_1 \rangle \in s\text{-}mgu_1(G_1, G'_1) \) and \( \langle \vartheta_2, t_2 \rangle \in s\text{-}mgu_1(G_2, G'_2) \) and \( t = t_1 t_2 \) and \( \vartheta = \vartheta_1 + \vartheta_2 \)

• \( \langle \vartheta, t \rangle \in s\text{-}mgu_1(G \setminus e, G' \setminus e) \) if \( \langle \vartheta, t \rangle \in s\text{-}mgu_1(G, G') \) and \( t \setminus e = t \)

Relation \( r\text{-}mgu_1 \) is defined like \( s\text{-}mgu_1 \), except for the case of atomic goals, where

• \( \langle \vartheta, t \rangle \in r\text{-}mgu_1(A, A') \) if \( \langle A', t \rangle \in I \) and \( \vartheta = \text{mgu}(A, A') \) and \( A' \leq A \).

Relations \( s\text{-}mgu \) and \( r\text{-}mgu \) are parametric with respect to an interpretation \( I \). These relations compute a substitution \( \vartheta \) and a trace \( t \) which have an operational counterpart, as stated in Theorems 7.3 and 7.5: \( \vartheta \text{-}G \) is the computed answer substitution of a computation for goal \( G \) generating trace \( t \). Finally, notice that the definition of relation \( r\text{-}mgu \) slightly differs from \( s\text{-}mgu \) for the case of atoms since an atom \( A \) is true if and only if there is an atom \( A' \) in the interpretation which is less defined than \( A \). Intuitively, \( r\text{-}mgu \) formalizes the fact that if an atomic formula \( A' \) is true in an interpretation, then any instance of \( A' \) is true as well, while the converse is not true. For example, let \( a \) be a constant. If \( \langle p(X), t \rangle \in I \), then also \( \langle p(a), t \rangle \in I \) since \( p(X) \leq p(a) \). On the other hand, \( \langle p(a), t \rangle \in I \) does not obviously imply that \( \langle p(X), t \rangle \in I \).

Definition 5.6. An interpretation \( I \) of a program \( P \) is a model iff every clause of \( P \) is true in \( I \).

Definition 5.7. Let \( I, I' \) be interpretations, \( I \leq I' \) iff \( I \subseteq I' \) (in the set theoretical sense).

Proposition 5.8. Let \( I, J \) be interpretations such that \( I \leq J \).

If a goal \( G \) is true in \( I \) with trace \( t \), then \( G \) is true in \( J \) with \( t \).

Proof. Immediate, by observing that \( I \leq J \) means that \( J = I \cup I' \) for some (possibly empty) \( I' \). Now, if \( r\text{-}mgu_1(G, G') = \langle \vartheta, t \rangle \), then also \( r\text{-}mgu_1(G \cup I', G') = \langle \vartheta, t \rangle \) since \( I' \) can be ignored when checking the truth of \( G \).

Proposition 5.9. The class of interpretations is a complete lattice with respect to \( \leq \).

Proof. If \( L \) is a set of interpretations, then \( \text{glb}(L) = \cap L \) and \( \text{lub}(L) = \cup L \) since the partial ordering \( \leq \) is defined in terms of classical set inclusion.

Notice that \( \emptyset \) and \( H \) are the bottom and top elements of the complete lattice, respectively.
Proposition 5.10. If $L$ is a nonempty set of models of a program $P$, then $\text{glb}(L) = \cap L$ is a model of $P$.

PROOF. Consider a unit clause $A \leftarrow$ of $P$. Since for each model $I$ of $L$, $\langle A, \lambda \rangle \in I$, we have that $\langle A, \lambda \rangle \in \cap L$, and so $A \leftarrow$ is true in $\cap L$.

Consider now a clause $A \leftarrow G$. Assume that $s\text{-mgu}_I(G, G') = \langle \theta, t \rangle$, i.e., $s\text{-mgu}_I(G, G') = \langle \theta, t \rangle$, $\forall I \in \cap L$. Then $\langle A\theta, t \rangle$ belongs to every $I$, and then to $\cap L$. $\blacksquare$

Proposition 5.11. For any program $P$,

i) The class of models of $P$ is a complete lattice, and

ii) There exists a minimal model $M_P$ of $P$.

PROOF.

i) By Proposition 5.10, it is sufficient to show that for any set $L$, there exists $\text{lub}(L)$. Let $LS$ be the set of the upper bounds of $L$. If $LS$ is empty, then $\text{lub}(L) = H$. Otherwise, $\text{lub}(L) = \text{inf}(LS) = \cap LS$.

ii) Immediate by Proposition 5.10 since $H$ is a model of $P$. $\blacksquare$

Example 5.12. Consider the following simple program $P$:

\begin{align*}
p &\leftarrow u!e \\
q &\leftarrow u?e \\
r &\leftarrow p||q
\end{align*}

The minimal model of $P$ is $\{\langle p, [u!e] \rangle, \langle q, [u?e] \rangle, \langle r, \lambda \rangle, \langle r, [u!e, u?e] \rangle, \langle r, [u?e, u!e] \rangle\}$.

5.2. Fixpoint Semantics

The declarative semantics of a program is generally also given in terms of the least fixpoint of an associated continuous transformation [20]. In logic programming, the fixpoint of this transformation is used to prove the effectiveness of the minimal model.

We extend the definition of the standard $T_p$ operator for logic programs, and we prove that it is monotonic and continuous.

Definition 5.13. For any program $P$, the mapping $T^*_p: 2^H \rightarrow 2^H$ on the set of interpretation is defined as follows:

\[ T^*_p(I) = \{\langle A', t \rangle \mid \exists A \leftarrow G \text{ in } P \text{ and } \forall G': \langle \theta, t \rangle \in s\text{-mgu}_I(G, G') \text{ and } A' = A\theta \} \cup \{\langle A, \lambda \rangle \mid \exists A \leftarrow \text{ in } P\}. \]

Definition 5.14. The transformation $T^*_p$ is monotonic and continuous.

PROOF.

(Monotonicity) Straightforward by Proposition 5.8.

(Continuity) Let $K$ be a chain (i.e., a totally ordered set) of interpretations. We have to prove that $T^*_p(K) = \cup T^*_p(K)$.

$T^*_p(K) \supseteq \cup T^*_p(K)$ follows by monotonicity.
Let \( \langle A', t \rangle \in T^*_p(\bigcup K) \). Then \( \exists A \leftarrow G \) in \( P \), \( \langle \vartheta, t \rangle \in \text{s-mgu}_1(G, G') \), and \( A' = A \vartheta \). Following the definition of function \( \text{s-mgu} \), the proof that \( \text{s-mgu}_{\bigcup K}(G, G') = \langle \vartheta, t \rangle \) relies upon the fact that a set of pairs \( \langle A_i, t_i \rangle \) is contained into \( \bigcup K \). For every pair \( \langle A_i, t_i \rangle \), there exists a set \( I_i \in K \) such that \( \langle A_i, t_i \rangle \in I_i \). Let \( I = \max \{I_1, I_2, \ldots, I_n\} \). Note that \( I \in K \) and that \( \langle \vartheta, t \rangle \in \text{s-mgu}_1(G, G') \). This means that \( \langle A', t \rangle T^*_p(I) \), and thus \( T^*_p(I) \subseteq T^*_p(K) \).

The powers of the \( T^*_p \) operator are defined as usual [2, 29]:

\[
T^*_p \uparrow 0(\emptyset) = \emptyset \\
T^*_p \uparrow (n + 1)(\emptyset) = T^*_p(T^*_p \uparrow n(\emptyset)) \\
T^*_p \uparrow \omega(\emptyset) = \bigcup_{n \in \omega} T^*_p \uparrow n(\emptyset).
\]

**Proposition 5.15.** The transformation \( T^*_p \) has least fixpoint (\( \text{lfp}(T^*_p) \)) and \( \text{lfp}(T^*_p) = T^*_p \uparrow \omega(\emptyset) \).

**Proof.** Standard.

**Proposition 5.16.** An interpretation \( I \) of a program \( P \) is a model if and only if \( T^*_p(I) \subseteq I \) (i.e., \( T^*_p(I) \leq I \)).

**Proof.** \( I \) is a model

\[
\text{iff } \forall A \leftarrow G \text{ in } P: A \leftarrow G \text{ is true} \\
\text{iff } \forall A \leftarrow G \text{ in } P \forall G' \text{ such that } \text{s-mgu}_1(G, G') = \langle \vartheta, t \rangle, \text{ then } \langle A \vartheta, t \rangle \in I \\
\text{iff } \forall \langle A \vartheta, t \rangle \text{ such that } \exists A \leftarrow G \text{ in } P \text{ and } \text{s-mgu}_1(G, G') = \langle \vartheta, t \rangle, \text{ then } \\
\langle A \vartheta, t \rangle \in I \\
\text{iff } T^*_p(I) \subseteq I.
\]

Finally, we prove the equivalence of the model-theoretic and the fixpoint semantics.

**Theorem 5.17.** For any program \( P \), \( M_p = \text{lfp}(T^*_p) = T^*_p \uparrow \omega(\emptyset) \).

**Proof.** The least fixpoint of \( T^*_p \) is a model by Proposition 5.16, as

\[
\text{lfp}(T^*_p) = \min \{I | (T^*_p(1) = I)\}.
\]

Moreover, \( M_p = \min \{I | (T^*_p(1) \leq I)\} \) is a fixpoint of \( T^*_p \). Indeed, it is trivial to check that if \( I \) is a model, then \( T^*_p(I) \) is a model as well. Therefore, if the minimal model were not a fixpoint, then it would not be minimal. Thus,

\[
\min \{I | (T^*_p(1) = 1)\} = \min \{I | (T^*_p(1) \leq 1)\}.
\]

6. OPERATIONAL SEMANTICS

The operational semantics is defined by means of a hierarchy of two abstract machines. The lower level one, defined as a transition system, describes the evolution of sequential goals. Then, the evolution of the whole system is represented by a 1-safe Place/Transition net. Given a parallel goal, we single out its sequential subgoals, representing the places of the net. A marking of the 1-safe P/T net is represented by a set of places, while the net transitions, describing
process interactions, are defined by structural inductive inference rules. For the sake of homogeneity with the declarative semantics, we consider ground communications. However, the operational semantics can be easily extended to model possibly nonground communications. A detailed description of how to do this is reported in [9].

6.1. The Transition System for Sequential Goals

We first define the sets of labels and configurations for sequential goals. We introduce a special configuration \((\text{failure})\) to explicitly distinguish the cases of finite failure of a derivation from the cases of deadlock. When a goal is unable to make further progress because a unifiable clause is missing, the configuration \((\text{failure})\) is reached.

**Definition 6.1.** The sets of labels \(\Lambda_G\) and configurations \(\Gamma_G\) for sequential goals are

\[
\Lambda_G = \mathcal{C}
\]

\[
\Gamma_G = \{ \langle G, \theta \rangle \mid G \in G \text{ and } \theta \in \text{Sub} \} \cup \{ \langle \square, \theta \rangle \mid \theta \in \text{Sub} \} \cup \{ \langle \text{failure} \rangle \}
\]

The (possibly infinite) set of transitions is generated by means of an inference system defined à la Plotkin in a syntax-driven style. In the following definitions, we will use the notation

\[
\gamma \rightarrow \gamma_1/\ldots/\gamma_n
\]

as a shorthand for the \(n\) rules

\[
\gamma \rightarrow \gamma_1, \ldots, \gamma_1', \ldots, \gamma_n
\]

Let us now introduce the definition of the derivation relation for sequential goals. As one may expect, we shall use some of the inference rules which have been used for CSLP in Section 2. More precisely, rules True, Atomic, and Seq of Definition 2.5 actually model the behavior of sequential goals of DL as well. We only extend them to explicitly deal with the case of failure. Rule Event of Definition 2.5 is slightly adapted to consider only ground communications. Finally, a new rule Choice is introduced to model choice goals.

**Definition 6.2.** The sequential goal derivation relation over configurations, written as

\[
\gamma G \rightarrow G \gamma',
\]

is defined as the least relation satisfying the following inference rules in addition to True, Atomic, and Seq of Definition 2.5.

**Event**

\[
\begin{align*}
\langle U!e, \theta \rangle & \xrightarrow{U!e} \langle \square, \theta \rangle \\
\langle U?e, \theta \rangle & \xrightarrow{U?e} \langle \square, \theta \rangle
\end{align*}
\]

where

- \(U\) is a ground term
Choice

\[
\left\langle G_C, \delta \right\rangle \xrightarrow{AG} \left\langle \Box, \theta \right\rangle / \left\langle G, \theta \right\rangle
\]

\[
\left\langle G_C \triangleright G_C, \delta \right\rangle \xrightarrow{AG} \left\langle \Box, \theta \right\rangle / \left\langle G, \theta \right\rangle
\]

\[
\left\langle G_C, \delta \right\rangle \xrightarrow{AG} \left\langle \Box, \theta \right\rangle / \left\langle G, \theta \right\rangle
\]

\[
\left\langle G_C \triangleright G_C, \delta \right\rangle \xrightarrow{AG} \left\langle \Box, \theta \right\rangle / \left\langle G, \theta \right\rangle
\]

Failure Atomic

\[
\forall A_2 \leftarrow G \in P: \not \exists \sigma = \text{mgu}(A_1, A_2) \\
\quad \langle A_1, \theta \rangle \rightarrow \langle \text{failure} \rangle
\]

Failure Seq

\[
\langle G_1, \delta \rangle \rightarrow \langle \text{failure} \rangle
\]

\[
\langle (G_1; G_2), \delta \rangle \rightarrow \langle \text{failure} \rangle
\]

Failure Choice

\[
\langle G_C, \delta \rangle \rightarrow \langle \text{failure} \rangle \land \langle G_C, \delta \rangle \rightarrow \langle \text{failure} \rangle
\]

\[
\langle G_C \triangleright G_C, \delta \rangle \rightarrow \langle \text{failure} \rangle
\]

REMARKS. In order to solve an event goal \(U_1\), a complementary event goal \(U_2\) has to be simultaneously solved. Recall that we consider ground events only. Rule Event states that a sequential goal performs an open communication assuming the existence of its partner. The process interaction will be described later by rule Sync in the net for parallel goals (see Section 6.2.2). Rule Choice says that if a goal \(G_c\) can be reduced somehow, the choice goal discards the other alternative and follows the derivation of \(G_c\). The new rule Failure Atomic is added for the Atomic case, which states that if there is no clause unifying with an atomic goal \(A_1\), then the configuration \(\langle \text{failure} \rangle\) is reached. Rule Failure Seq states that if the derivation of the left goal leads to a failure, the whole goal fails, while Failure Choice says that a choice goal fails if both of its components fail. Finally, notice that all of the configurations whose left component is a parallel goal are stuck in the transition system for sequential goals.

6.2. The Petri Net for Parallel Goals

In net theory, a concurrent distributed system is described by using two kinds of objects: places and transitions. A place represents the state of a subpart of the system, so that the global state of a system is a set of places, that is, of local states. A transition is seen as an action which affects some places. A flow relation states how the state changes after the occurrence of a transition. A net \(N\) is a triple \(\langle S, T, F \rangle\), where

- \(S\) is the set of places
- \(T\) is the set of transitions
- \(S \cap T = \emptyset\)
\( F \subseteq (S \times T) \cup (T \times S) \) is the \textit{flow} relation.

Given a net \( N = (S, T, F) \), let \( x, y \in S \cup T \). The set \( \{y | x F x\} \), called the \textit{preset} of \( x \), is denoted by \( \cdot x \), while \( x \cdot \) denotes \( \{y | y F x\} \), the \textit{postset} of \( x \). Moreover, \( x \) is \textit{isolated} if \( \cdot x \cup x \cdot = \emptyset \). A net \( N = (S, T, F) \) is \textit{connected} if the following holds:

\[ \forall t \in T : t \neq \emptyset \text{ and } t \cdot \neq \emptyset \]

with the meaning that each transition has at least one preplace and at least one postplace.

A function \( M : S \to \mathbb{N} \), where \( \mathbb{N} \) is the set of natural numbers, is a \textit{marking} of the net \( N \). A quadruple \( \Sigma = (S, T, F, M_0) \) is a \textit{Place/Transition net} (P/T for short) iff \( (S, T, F) \) is a net and \( M_0 \) is a marking, called the initial marking. A transition \( t \in T \) is \textit{enabled} by a marking \( M \) iff \( \forall s \in \cdot t, M(s) \geq 1 \). Furthermore, the occurrence of a transition \( t \) \textit{produces} the marking \( M' \) (denoted by \( M[t > M'] \)):

\[
M'(s) = \begin{cases} 
M(s) & \text{if } s \not\in (\cdot t \cup t \cdot) \text{ or } s \in (\cdot t \cap t \cdot) \\
M(s) - 1 & \text{if } s \in \cdot t \setminus t \\
M(s) + 1 & \text{if } s \in t \cdot \setminus t
\end{cases}
\]

The \textit{sequential behavior} of a P/T net can be given by means of the so-called firing sequences. A finite or infinite sequence \( \xi \) of markings and transitions of \( \Sigma = (S, T, F, M_0) \) is called a \textit{firing sequence} of \( \Sigma \) iff \( \xi = M_0, t_1, M_1, t_2, M_2, \ldots \) and \( \forall i \geq 1 M_{i-1}[t_i > M_i] \).

A firing sequence \( \xi \) is \textit{terminal} iff \( \xi \) is finite and no transition \( t \in T \) is enabled by its last marking. \( [M_0] \) denotes the set of all the possible markings \( M \) reachable from \( M_0 \) by successive firings of enabled transitions, that is, all of the markings \( M \) occurring in any firing sequence \( \xi \). Let \textit{last}(\( \xi \)) denote the last marking of a finite firing sequence. Given a P/T net \( \Sigma = (S, T, F, M_0) \) and a reachable marking \( M \in [M_0] \), we call \( M \) an \textit{n-safe} marking (with \( n \in \mathbb{N} \)) iff \( \forall s \in S, M(s) \leq n \). The P/T net \( \Sigma \) is \textit{n-safe} iff \( \forall M \in [M_0], M \) is \textit{n-safe}. This means that a P/T net is safe if the number of tokens in any place has a finite bound. The nets we use for giving semantics to DL are 1-safe P/T nets.

\[ \text{6.2.1. Getting Places by Splitting Parallel Goals} \]

\textit{Definition 6.3}. We give the syntax of places:

\[
S ::= \langle G, \varnothing \rangle \langle G, \varnothing \rangle \langle \text{failure} \rangle \langle \text{id} S, G' \rangle \langle \text{id} S, G' \rangle \langle S \setminus e, G' \rangle
\]

where \( \varnothing \) is the empty goal, \( G \) is a sequential goal, \( \varnothing \) is a substitution, \( G' \) is a goal, and tags "id" and "id'" record the context in which a sequential component is set. The set of places \( S \) is ranged over by \( s \), and its subsets are named \( I, J \) (possibly indexed).

Intuitively, a place is a sequential goal, together with a substitution and an access path defining its location within the syntactical structure of the parallel goal. If the access path is not empty, the second component of the pair is a goal representing a continuation (what its ancestor has to do next).

Now, we describe how to map any pair \( \langle \text{goal}, \text{substitution} \rangle \) into a (finite) set of places. Notice that we do not need multisets since the decomposition function can never build two identical subgoals (places) from a goal. Therefore, such a set of places also represents a 1-valued marking.
Definition 6.4. Function \( \text{dec}: \Gamma_G \to \text{fin}(2^S) \) is defined by structural induction on goals.

Sequential Goal

\[
\text{dec}(\langle \square, \vartheta \rangle) = \{\langle \square, \vartheta \rangle\}
\]
\[
\text{dec}(\langle \text{failure} \rangle) = \{\langle \text{failure} \rangle\}
\]

\( G \) is a sequential goal implies \( \text{dec}(\langle G, \vartheta \rangle) = \{\langle G, \vartheta \rangle\} \)

The following rules apply only to parallel goals:

Sequential Composition

\[
\langle s, G' \rangle \in \text{dec}(\langle G, \vartheta \rangle) \quad \text{implies} \quad \langle s, G'; G'' \rangle \in \text{dec}(\langle G; G'', \vartheta \rangle)
\]
\[
\langle s, \square \rangle \in \text{dec}(\langle G, \vartheta \rangle) \quad \text{implies} \quad \langle s, G' \rangle \in \text{dec}(\langle G; G', \vartheta \rangle)
\]

Parallel Composition

\[
s \in \text{dec}(\langle G, \vartheta \rangle) \quad \text{implies} \quad \langle s|\text{id}, \square \rangle \in \text{dec}(\langle G|G', \vartheta \rangle)
\]
and \( \langle \text{id}|s, \square \rangle \in \text{dec}(\langle G'|G, \vartheta \rangle) \)

Restriction

\[
s \in \text{dec}(\langle G, \vartheta \rangle) \quad \text{implies} \quad \langle s|\text{id}, \square \rangle \in \text{dec}(\langle G|\text{id}, \vartheta \rangle)
\]
and \( \langle \text{id}|s, \square \rangle \in \text{dec}(\langle G', G, \vartheta \rangle) \)

The configurations for the empty goal, failure, and for a sequential goal are singleton sets of places. The Sequential Composition rule simply states that the decomposition recursively splits only the left component, and that the right component contributes to the continuation. Finally, the places out of a parallel goal are exactly those derived by its two components, enriched by tags "\text{id}" or "\text{idl}," and with the empty goal as continuation since nothing has to be done once the parallel goal is solved.

Example 6.5. Consider the parallel goal \(((G_1||G_2); G_3)||G_4); G_5\), where all of the \( G_i \) are sequential goals.

Given a substitution \( \vartheta \), we have

i) \( \text{dec}(\langle G_1, \vartheta \rangle) = \{\langle G_1, \vartheta \rangle\} \quad i = 1, 2, \ldots, 5; \)
by the Sequential Goal rule

ii) \( \text{dec}(\langle G_1||G_2, \vartheta \rangle) = \{\langle G_1, \vartheta \rangle|\text{id}, \square, \langle \text{id}|G_2, \vartheta \rangle, \square\} \)
by i) and the Parallel Composition rule

iii) \( \text{dec}(\langle G_1||G_2; G_3, \vartheta \rangle) = \{\langle G_1, \vartheta \rangle|\text{id}, G_3, \langle \text{id}|G_2, \vartheta \rangle, G_3\} \)
by ii) and the Sequential Composition rule

iv) \( \text{dec}(\langle G_1||G_2; G_3||G_4, \vartheta \rangle) \)
\[
= \{\langle G_1, \vartheta \rangle|\text{id}, G_3, \langle \text{id}|G_2, \vartheta \rangle, G_3, \langle \text{id}|G_4, \vartheta \rangle, \square\} \)
by iii) and the Parallel Composition rule

v) \( \text{dec}(\langle G_1||G_2; G_3||G_4; G_5, \vartheta \rangle) \)
\[
= \{\langle G_1, \vartheta \rangle|\text{id}, G_3, G_5, \langle \text{id}|G_2, \vartheta \rangle, G_3, \langle \text{id}|G_4, \vartheta \rangle, G_5, \langle \text{id}|G_5, \vartheta \rangle, \square\} \)
by iv) and the Sequential Composition rule.

Note that the \text{dec} function is injective, but not surjective since there are sets of places which are not obtainable via \text{dec}.

Now, we characterize the set of places corresponding to a possible marking reachable from a starting goal. Let us consider the empty goal \( \square \) and failure as
legal DL goals because we use them for representing a successfully or unsuccessfully terminated component process waiting for the termination of its brother. Given a set \( J \) of places, \( J \) is \textit{complete} iff there is a goal \( G \) and a substitution \( \vartheta \) such that \( J \) is \( \text{dec}(\langle G, \vartheta \rangle) \) up to substitutions, that is, the substitutions occurring in corresponding places can be different. Complete sets of places correspond to 1-valued reachable marking. The correspondence is "up to substitutions" because sequential components generated by a parallel goal work in separate environments, and thus their substitutions may change during the refutation process.

6.2.2. Deductive System for Net Transitions. The set of labels of the net transition relation is the same as that of the sequential transition relation.

\textbf{Definition 6.6.} The set \( \Lambda_D \) of labels is \( C \).

In the following definition, \( \mathcal{E} \) stands for a goal \( G \) or for the empty goal \( \Box \). Sometimes, the set \( \{\langle i, \mathcal{E} \rangle \mid i \in I\} \) is denoted by the pair \( \langle I, \mathcal{E} \rangle \).

\textbf{Definition 6.7.} The net transition relation \( I_1 \xrightarrow{\Lambda_D} I_2 \) is defined as the least relation satisfying the following axioms and inference rules.

\textbf{Join}

\[ \vartheta_1 \text{ and } \vartheta_2 \text{ are compatible} \]

\[ \{s_1, s_2\} \rightarrow \text{dec}(\langle \vartheta(\vartheta_1 + \vartheta_2), \vartheta_1 + \vartheta_2 \rangle) \]

\[ \text{where } s_1 = \langle \langle \Box, 1, \text{id}, \mathcal{E} \rangle \rangle \text{ and } s_2 = \langle \text{id} \langle \Box, \vartheta, 2 \rangle, \mathcal{E} \rangle \]

\[ s_1 = \langle \langle \text{failure} \rangle \langle \text{id}, \mathcal{E} \rangle \rangle \wedge s_2 = \langle \text{id} \langle \Box, \vartheta \rangle, \mathcal{E} \rangle \]

\[ \{s_1, s_2\} \rightarrow \{\langle \text{failure} \rangle\} \]

\[ s_1 = \langle \text{id} \langle \text{failure} \rangle, \mathcal{E} \rangle \wedge s_2 = \langle \langle \Box, \vartheta \rangle \text{id}, \mathcal{E} \rangle \]

\[ \{s_1, s_2\} \rightarrow \{\langle \text{failure} \rangle\} \]

\[ s_1 = \langle \langle \text{failure} \rangle \langle \text{id}, \mathcal{E} \rangle \rangle \wedge s_2 = \langle \text{id} \langle \text{failure} \rangle, \mathcal{E} \rangle \]

\[ \{s_1, s_2\} \rightarrow \{\langle \text{failure} \rangle\} \]

\textbf{Act}

\[ \langle G, \vartheta \rangle \xrightarrow{\alpha} \langle G', \vartheta' \rangle / \langle \Box, \vartheta' \rangle / \langle \text{failure} \rangle \]

\[ \{\langle G, \vartheta \rangle\} \xrightarrow{\alpha} \text{dec}(\langle G', \vartheta' \rangle) / \{\langle \Box, \vartheta' \rangle\} / \{\langle \text{failure} \rangle\} \]

\textbf{Async}

\[ I_1 \xrightarrow{\alpha} I_2 \]

\[ \langle I_1 \text{id}, \mathcal{E} \rangle \xrightarrow{\alpha} \langle I_2 \text{id}, \mathcal{E} \rangle \]

\[ I_1 \xrightarrow{\alpha} I_2 \]

\[ \langle \text{id} \langle I_1, \mathcal{E} \rangle \rangle \xrightarrow{\alpha} \langle \text{id} \langle I_2, \mathcal{E} \rangle \rangle \]

\textbf{Sync}

\[ I_1 \xrightarrow{\lambda_D} I_1' \wedge I_2 \xrightarrow{\lambda_D} I_2' \wedge \lambda_1 = \lambda_2 \]

\[ \langle I_1 \text{id} \cup \text{id} \langle I_2, \mathcal{E} \rangle \rangle \rightarrow \langle I_1' \text{id} \cup \text{id} \langle I_2', \mathcal{E} \rangle \rangle \]

\textbf{Res}

\[ I_1 \xrightarrow{\Lambda_D} I_1' \wedge \lambda_D \setminus e = \lambda_D \]

\[ \langle I_1 \setminus e, \mathcal{E} \rangle \rightarrow \langle I_1' \setminus e, \mathcal{E} \rangle \]
REMARKS. (Join) As soon as both components of a parallel goal terminate, the continuation of the parallel goal is enabled and its substitution is updated. If $\partial_1$ and $\partial_2$ are not compatible, then the set \{s_1, s_2\} represents a deadlocked state. If at least one of the components has failed, then the whole system fails.

(Act) This is an import rule for sequential goals. Everything a sequential goal can perform in the transition system for sequential goals can also be performed by the corresponding place in the net.

(Async) From the premise that a set J of places performs an action, we can infer that for any context, there is a suitable set of places which is able to perform that action.

(Sync) The communication mechanism is handshake. A communication takes place if and only if the message in the two event goals is the same ground term.

Note that the net transition relation is asynchronous. Actually, the transitions are independent of those sequential components (places) which are "concurrent" with the rewritten ones, but inactive. In other words, each transition is context-independent.

Property 6.8. If $I_1 \xrightarrow{\lambda_D} I_2$ is a transition, there exists a set of places $J$ such that $J \cap I_1 = \emptyset$ and $J \cup I_1$ is complete. Furthermore, for every such $J$, we also have that $J \cap I_2 = \emptyset$ and that $J \cup I_2$ is a complete set of places.

PROOF. Immediate by induction on the structure of the proof of the transition. ■

6.2.3. DL-Systems. Given a DL program $P$, a goal $G$, and the empty substitution $\varepsilon$, we want to build the P/T net representing the distributed refutation of the initial goal $G$.

The set of places $S$, the set of transition $T$, the flow relation $F$, and the initial marking $M_0$ of the P/T net $\Sigma_{P,G} = \langle S, T, F, M_0 \rangle$, obtained by the program $P$ starting from the initial goal $G$, are defined as the least sets, relation, and function satisfying the following inference rules:

\[
\begin{align*}
\text{s} \in \text{dec}(\langle G, \varepsilon \rangle) \\
\text{s} \in S \text{ and } M_0(s) = 1
\end{align*}
\]

\[
\begin{align*}
I_1 \subseteq S \text{ and } I_1 \xrightarrow{\lambda_D} I_2
\end{align*}
\]

\[
\begin{align*}
( I_1 \xrightarrow{\lambda_D} I_2 ) \subseteq T \text{ and } I_2 \subseteq S \text{ and } \forall s \in ( I_2 \setminus \text{dec}(\langle G, \varepsilon \rangle)) : M_0(s) = 0
\end{align*}
\]

\[
\begin{align*}
I_1 \cap F( I_1 \xrightarrow{\lambda_D} I_2 ) \text{ and } ( I_1 \xrightarrow{\lambda_D} I_2 ) F I_2
\end{align*}
\]

Note that $\Sigma_{P,G}$ is indeed a P/T net since $\langle S, T, F \rangle$ is a net (it satisfies the condition $S \cap T = \emptyset$) and $M_0$ is a marking. The net is connected since $\forall t \in T : t \neq \emptyset$ and $t \neq \emptyset$ is satisfied (third rule). Finally, we are going to prove that $\Sigma_{P,G}$ is 1-safe.

Theorem 6.9. Given a program $P$ and a goal $G$, the P/T net $\Sigma_{P,G}$ is 1-safe.

PROOF. Since the P/T net $\Sigma_{P,G}$ is 1-safe if and only if $\forall M \in [M_0 > M]$ is 1-safe, the proof is by induction on the set $[M_0 > ]$. At each step of the induction, we also...
prove that the set \( P = \{ s \in S | M(s) = 1 \} \) is complete in order to inherit the results of Property 6.8. The initial marking \( M_0 \) is 1-safe by definition, and the set \( P_0 = \{ s \in S | M_0(s) = 1 \} = \text{dec}(G, \varepsilon) \) is complete. Let us suppose, by inductive hypothesis, that a reachable marking \( M \) is 1-safe, and that its corresponding set \( P \) is complete. Let \( t = I_1 \xrightarrow{A_D} I_2 \) be an \( M \)-enabled transition and \( M[t \rightarrow M'] \). We have to prove that the produced marking \( M' \) is 1-safe and that the corresponding set \( P' \) is complete. Let \( J = P \setminus I_1 \). By definition of the produced marking, \( M' \), on the set of places \( I_1 \setminus I_2 \) holds 0, on the set of places \( I_2 \setminus J \cup I_1 \cap I_2 \) holds 1, while for the set of places \( J \cap I_2 \), marking \( M' \) would hold 2. But by Property 6.8, we have that \( J \cap I_2 = \emptyset \)—thus, marking \( M' \) is 1-safe—and finally, that \( P' = J \cup I_2 \) is complete.

6.3. Observational Semantics

Now we have to decide what to observe from a DL-system \( \Sigma_{P, G} \). The final state operational semantics captures more than the set of all of the computed answer substitutions. In fact, the last marking of a terminal firing sequence \( \xi \) may represent

- The singleton \( \{ \langle \square, \varnothing \rangle \} \), i.e., computed answer substitution \( \varnothing \)
- The singleton \( \{ \langle \text{failure} \rangle \} \), i.e., a failure occurred since an atom goal cannot unify with any clause
- A nonsingleton set \( I \), which stands for a deadlock due to (possibly more than) one of the following reasons:
  - the nonexistence of a pair of compatible substitutions (termination of a parallel goal)
  - the inability to match terms in a communication
  - the possible partner for communication is terminated or failed.

Summing up, we can say that the final state operational semantics of a DL-system \( \Sigma_{P, G} \) is given below, where \( \xi \) represents a firing sequence:

\[
[\Sigma_{P, G}] = \{ \varnothing | \xi \text{ is terminal and } \text{last}(\xi) = \{\langle \square, \varnothing \rangle\}\} \cup \{\text{fail} | \xi \text{ is terminal and } \text{last}(\xi) = \{\langle \text{failure} \rangle\}\} \cup \{A | \xi \text{ is terminal and } \text{last}(\xi) | \geq 2\} \cup \{\& | \xi \text{ is infinite}\}.
\]

We could be dissatisfied for this final state semantics since it is not adequate for infinite computations (perpetual processes) in the sense that relevant features of behaviors are not dealt with. A better description of system \( \Sigma_{P, G} \) should take into account the causal relations among the computation steps, that is, every computation is observed as the partial ordering of steps it performs. This way, important properties, such as liveness and safety, can be easily observed.

As an example, let us consider the following DL program taken from [36]. A counter object is programmed, allowing commands for incrementing and checking the value of a counter:

```plaintext
counter(S) ← up(N) ? mail; U is S + N; counter(U)
◇equal_to(S) ? mail; counter(S)
producer ← prod(X); X! mail; producer
```

where prod is a predicate generating commands for the counter. An initial goal may be

\[ \text{counter(0)} \parallel \text{producer} \parallel \text{producer}. \]

It is trivial to observe that the computation of this DL system may not terminate while still having a precise meaning: an infinite sequence of communications and updatings of the counter value. The competition between the two producers may be solved by the counter serving only one of them from a certain time onwards: this is a typical example of unfair computation. A situation of partial deadlock may arise if a producer sends an equal-to command to the counter process with an argument more minor than the counter value: in this case, the producer which has sent the message is deadlocked, while the other producer and the counter may proceed.

A first attempt to express such properties in this framework may be the following. We look at a firing sequence simply as the sequence of the sets of places with one token. Given a reachable marking \( M \), a firing sequence \( \xi = M_0 t_1 M_1 t_2 M_2 t_3 M_3 \ldots \) is observed as the sequence \( P_0 P_1 P_2 P_3 \ldots \) of places denoting the markings.

Among the various notion of fairness, we just consider here a few of them. The first is called \textit{global fairness} in [18]. It states that a transition \( t = I_1 \rightarrow I_2 \) which is always enabled will eventually be fired. This property is expressed by the following formula:

\[ \neg (\exists i \in \mathbb{N}, \exists t \in T : t \subseteq \bigcap_{j \geq i} P_j). \]

A second, more demanding notion of fairness (\textit{local fairness} [18]) concerns the fact that a place which can always (infinitely often, respectively) be involved in a firing, possibly within different transitions, will eventually be consumed. We represent this fact, respectively, as

\[ \neg (\exists i \in \mathbb{N}, \exists s \in S : s \in \bigcap_{j \geq i} P_j \text{ and } \forall j \geq i \exists t \in T : t \subseteq P_j \text{ and } s \subseteq t) \]

and

\[ \neg (\exists i \in \mathbb{N}, \exists s \in S : s \in \bigcap_{j \geq i} P_j \text{ and } \exists \text{ infinitely many } j, \]

\[ j \geq i, \exists t \in T : t \subseteq P_j, s \subseteq t \]

We can also write formulas stating that a firing sequence is free of some kind of \textit{deadlock}. For instance, a sequence \( P_0 P_1 P_2 P_3 \ldots \), satisfying the following formula:

\[ \exists i \in \mathbb{N}, \exists s \in S : s \in \bigcap_{j \geq i} P_j \text{ and } \text{goal}(s) = g_e / g_e, g / g_c \diamond g_c \]

shows a deadlock due to a missing matching event goal, where \( \text{goal} : S \rightarrow G \) is the function that yields the sequential goal represented by a place. Notice that a sequential process of this kind could have partners that may never perform the complementary event goal. Thus, a sequence with a partial deadlock may be locally fair as well.

Another manner to define the operational semantics of a DP system \( \Sigma_{P,G} \) consists of considering its processes [39], that is, the pairs \( \pi = \langle N, \rho \rangle \), where \( N \) is an occurrence net and \( \rho : B \cup E \rightarrow S \cup T \) a labeling of \( N \).

For finite processes, we have that \( \text{Max}(N) \) may be either \( \langle \emptyset, \emptyset \rangle \), or \( \langle \text{failure} \rangle \), or even a nonsingleton set of places. For infinite processes, \( \text{Max}(N) \) denotes the set
of places that do not fire from a certain point onwards. If a transition \( t \) is such that \( \cdot t \subseteq \text{Max}(N) \), then \( \langle N, \rho \rangle \) is not globally fair. On the contrary, we can say nothing about local fairness. Even if a place \( s \) belongs to \( \text{Max}(N) \), we do not know if there has always (infinitely often) been some transition \( t \), which may be time-dependent, such that \( s \in \cdot t \).

The discussion above can be considered a first possible approach to the problem of describing properties of perpetual processes. Further research would be mandatory to better investigate this subject.

7. DECLARATIVE AND OPERATIONAL SEMANTICS COINCIDE

We show that the declarative semantics defined in Section 5 and the operational semantics presented in Section 6 coincide. The proofs of soundness and completeness theorems conclude our study of the semantics of Distributed Logic.

As pointed out in Section 5, the definition of relation \( s\text{-mgu}(G, G') \) (Definition 5.5) is not completely appropriate when aiming at proving the equivalence between the declarative and the operational semantics. A more strict definition is necessary, considering additional requirements on the separation of the environments of goals. Roughly, such requirements are simply satisfied by choosing suitable variants of the goals. The definition of relations \( s\text{-mgu} \) and \( r\text{-mgu} \) is refined as follows.

- \( \langle \emptyset, t \rangle \in s\text{-mgu}(A, A') \) if \( \text{Vars}(A) \cap \text{Vars}(A') = \emptyset \) and \( \langle A', t \rangle \in I \) and \( \emptyset = \text{mgu}(A, A') \)
- \( \langle e, [u!e] \rangle \in s\text{-mgu}(u!e, u!e) \) if \( u \) is a ground term
- \( \langle e, [u?e] \rangle \in s\text{-mgu}(u?e, u?e) \) if \( u \) is a ground term
- \( \langle \emptyset_1, \emptyset_2, t_1 t_2 \rangle \in s\text{-mgu}((G_1 ; G_2), (G'_1 ; G'_2)) \) if \( \text{Vars}(G_1 ; G_2) \cap \text{Vars}(G'_1 ; G'_2) = \emptyset \) and \( \text{Vars}(G'_1) \cap \text{Vars}(G'_2) = \emptyset \) and \( \langle \emptyset_1, t_1 \rangle \in s\text{-mgu}(G_1, G'_1) \) and \( \langle \emptyset_2, t_2 \rangle \in s\text{-mgu}(G_2, G'_2) \)
- \( \langle \emptyset, t \rangle \in s\text{-mgu}((G_1 \diamond G_2), (G'_1 \diamond G'_2)) \) if \( \text{Vars}(G_1 \diamond G_2) \cap \text{Vars}(G'_1 \diamond G'_2) = \emptyset \) and \( \text{Vars}(G'_1) \cap \text{Vars}(G'_2) = \emptyset \) and \( \langle \emptyset_1, t \rangle \in s\text{-mgu}(G_1, G'_1) \) and \( G_2 = G'_2 \) with \( \emptyset_2 = \text{mgu}(G_2, G'_2) \) and \( G_1 = G'_1 \) and \( \emptyset = \emptyset_1 + \emptyset_2 \)
- \( \langle \emptyset, t \rangle \in s\text{-mgu}((G_1 \diamond G_2), (G'_1 \diamond G'_2)) \) if \( \text{Vars}(G_1 \diamond G_2) \cap \text{Vars}(G'_1 \diamond G'_2) = \emptyset \) and \( \text{Vars}(G'_1) \cap \text{Vars}(G'_2) = \emptyset \) and \( \langle \emptyset_2, t \rangle \in s\text{-mgu}(G_2, G'_2) \) and \( G_1 = G'_1 \) with \( \emptyset_1 = \text{mgu}(G_1, G'_1) \) and \( G_2 = G'_2 \) and \( \emptyset = \emptyset_1 + \emptyset_2 \)
- \( \langle \emptyset_1, t_1 \rangle \in s\text{-mgu}(G_1, G'_1) \) if \( \text{Vars}(G_1) \cap \text{Vars}(G'_1) = \emptyset \) and \( \text{Vars}(G'_1) \cap \text{Vars}(G'_2) = \emptyset \) and \( \langle \emptyset_1, t_1 \rangle \in s\text{-mgu}(G_1, G'_1) \) and \( \langle \emptyset_2, t_2 \rangle \in s\text{-mgu}(G_2, G'_2) \) and \( t \in t_1 \cdot t_2 \) and \( \emptyset = \emptyset_1 + \emptyset_2 \)
• \((\vartheta, t) \in s\text{-}mgu_1((G \\not\vdash e), (G' \not\vdash e))\) if \(\operatorname{Vars}(G) \cap \operatorname{Vars}(G') = \emptyset\)
  and \((\vartheta, t) \in s\text{-}mgu_1(G, G')\) and \(t \setminus e = t\)

Relation \(r\text{-}mgu_1\) is defined like \(s\text{-}mgu_1\), except for the case of atomic goals, where

• \((\vartheta, t) \in r\text{-}mgu_1(A, A')\) if \(\operatorname{Vars}(A) \cap \operatorname{Vars}(A') = \emptyset\) and \((\vartheta', t) \in r\text{-}mgu_1(A, A')\) and \((\vartheta', t) \in r\text{-}mgu_1(A, A')\) and \(A \not\vdash e = A\).

Notice that, if \(s(r)\text{-}mgu(A, G') = (\vartheta, t)\), then \(\vartheta\) is exactly the mgu between \(A\) and \(G'\). All of the cases are trivial, except for parallel, sequential, and nondeterministic composition:

\[
\operatorname{mgu}((a_1 \parallel a_2), (a_1 \parallel a_2)) = \operatorname{mgu}((a_1, a_2), (a_2, a_1)) = \vartheta_1 \parallel \vartheta_2,
\]

and thanks to a result in [19], \(\operatorname{mgu}((G_1, G_2), (G'_1, G'_2)) = \operatorname{mgu}((G_1, G'_1), (G_2, G'_2)) = \vartheta_1 \parallel \vartheta_2\).

For the sake of brevity and simplicity, given a DL-system \(P\), \(G\), a successful firing sequence \(\xi = \operatorname{dec}((G, e)) \stackrel{A}{\rightarrow} \{D, \vartheta\}\), the sequence of markings and net transitions in \(\xi\) with initial substitution \(\sigma\), that is,

\[
\xi \sigma = \operatorname{dec}((G \sigma, e)) \stackrel{A}{\rightarrow} \{D, \sigma \vartheta\}
\]

is also derivable.

**Proof.** The proof is by induction on the length of the firing sequence and on the structure of the derivation of the net transitions. The only rule which depends on the substitution in the place is \(\text{Join}\), where the compatibility of substitutions is checked. If \(\vartheta_1\) and \(\vartheta_2\) are compatible with substitution unifier \(\delta\), then \(\sigma \vartheta_1\) and \(\sigma \vartheta_2\) are also compatible with the same substitution unifier \(\delta\) since \((\vartheta_1, \vartheta_2) \oplus \delta = \sigma(\vartheta_1) \oplus \delta = \sigma(\vartheta_2) \oplus \delta\).

**Lemma 7.2.** Let \(\Sigma_{P, G}\) a DL-system, and let \(M_P\) be the minimal model of \(P\). If \((\vartheta, t) \in s\text{-}mgu_{M_P}(G' \vartheta, G'')\), \(\operatorname{Vars}(G') \cap \operatorname{Vars}(G'') = \emptyset\), and \(\operatorname{mgu}(G', G'') = \gamma\), we have that \((\gamma, t) \in s\text{-}mgu_{M_P}(G', G'')\).

**Proof.** Immediate, by structural induction on goal \(G'\).

We are now in the position of proving the soundness theorem.

**Theorem 7.3 (Soundness).** Given a DL-system \(\Sigma_{P, G}\), assume \(\operatorname{dec}((G, e)) \stackrel{A}{\rightarrow} \{\varnothing, \vartheta\}\). Then \(\exists G'\) such that \((\vartheta, t) \in s\text{-}mgu_{M_P}(G, G')\) and \(\vartheta|_A = \vartheta'|_A\).

**Proof.** The proof is by induction on the length of the computation and on the proof of its first transition. The induction is proved by means of inference rules, where \(\text{Unit}\) and \(\text{Derive}\) induce on the length of the computation in the case of atomic goals, and the other rules on the syntactical structure of the goal \(G\), thus obtaining one (in the case of choice goal) or two simpler computations.

\((\text{Unit})\)

\(A'\leftarrow\) is a variant (sharing no variables with \(A\)) of a unit clause in \(P\)

implies

\[
\operatorname{dec}((A, e)) \rightarrow \{\varnothing, \vartheta\} \Rightarrow (\vartheta', \lambda) \in s\text{-}mgu(A, A') \text{ and } \vartheta|_A = \vartheta'|_A
\]

The (one-step) computation can be derived if \(\operatorname{mgu}(A, A') = \vartheta\), thanks to a proof that involves \(\text{Atomic}\) and \(\text{Act}\) inference rules. On the other hand, since \(M_P\) is a
model of \(P\), the unit clause \(A' \leftarrow \) is true. This means that \(\langle A', \lambda \rangle \in M_p\), and thus \(\langle \delta', \lambda \rangle \in \text{s-mgu}(A, A')\).

(Derive)

\[
\text{dec}(\langle G\delta, \varepsilon \rangle) \xrightarrow{\lambda} \{\langle \Box, \psi \rangle\} \Rightarrow \langle \psi', t \rangle \in \text{s-mgu}(G\delta, G') \quad \text{and} \quad \psi'_{G\delta} = \psi'_{G'}
\]

and

\(A' \leftarrow G\) is a variant (sharing no variables with \(A\)) of a clause in \(P\) imply

\[
\text{dec}(\langle A, \varepsilon \rangle) \Rightarrow \text{dec}(\langle G\delta, \delta \rangle) \xrightarrow{\lambda} \{\langle \Box, \delta \psi \rangle\}
\]

\[
\Rightarrow \langle \delta', t \rangle \in \text{s-mgu}(A, A''') \quad \text{and} \quad \delta|_A = \delta|_A
\]

The step \(\text{dec}(\langle A, \varepsilon \rangle) \Rightarrow \text{dec}(\langle G\delta, \delta \rangle)\) can be derived by applying clause \(A' \leftarrow G\) if \(\text{mgu}(A, A') = \delta\). By Lemma 7.1, \(\text{dec}(\langle G\delta, \delta \rangle) \xrightarrow{\lambda} \{\langle \Box, \delta \psi \rangle\}\) is also derivable. Thus, \(\delta = \delta\psi, \delta = \delta|_A \cup \delta|_A\) (since \(\delta\) is an \(\text{mgu}\) between atoms which do not share variables) and \(\delta|_A = \delta|_A\psi|_{G\delta}A\). By inductive hypothesis, we know that \(\psi' = \text{mgu}(G\delta, G') = \psi'_{G\delta} \cup \psi'_{G'}, \text{ and } \psi'_{G\delta} = \psi'_{G\delta}\). It is worth observing that \(G'\) and \(G\delta\), \(G'\) and \(G\), and \(G'\) and \(A\), respectively, do not share variables by construction. We have to prove that

a) \(\exists A'', \text{ sharing no variables with } A, \text{ such that } \langle \delta', t \rangle \in \text{s-mgu}(A, A'')\)

b) \(\delta|_A = \delta|_A\).

We observe that

- \(\delta(\psi'_{G\delta}) \cup \psi'_{G'}\) is a unifier of \(G\) and \(G'\), and thus \(\exists \text{mgu}(G, G') = \gamma = \gamma_{G} \cup \gamma_{G'}\),

  - by Lemma 7.2, there exists \(\langle \gamma, t \rangle \in \text{s-mgu}(G, G')\),

- \(\gamma \leq \delta(\psi'_{G\delta}) \cup \psi'_{G'}, \text{ and therefore } \gamma_{G} \leq (\delta|_A\psi'_{G\delta}),\)

- \(A'\gamma_{G} \leq A'(\delta|_A\psi'_{G\delta}) = A(\delta|_A\psi'_{G\delta}).\)

a) Since clause \(A' \leftarrow G\) is true in \(M_p\) and \(\langle \gamma, t \rangle \in \text{s-mgu}(G, G')\) with \(\gamma = \gamma_{G} \cup \gamma_{G'}, \text{ then } \langle A'\gamma_{G}, t \rangle \in M_p\). The atom \(A''\) we are looking for is exactly \(A'\gamma_{G}\. Since \(A'' \leq A'(\delta|_A\psi'_{G\delta}), \text{ let } \chi\) be the least substitution such that \(A''\chi = A(\delta|_A\psi'_{G\delta}), \text{ i.e., } \delta'' = \chi \cup (\delta|_A\psi'_{G\delta})|_A\) is the most general unifier of \(A\) and \(A''\) such that \(\langle \delta', t \rangle \in \text{s-mgu}(A, A'')\).

b) \(\delta|_A = (\delta|_A\psi'_{G\delta})|_A = (\delta|_A\psi'_{G\delta}) = \delta|_A\) by inductive hypothesis.

(\text{Event})

\(U\) is a ground term implies

\[
\langle [U!e, \varepsilon] \rangle \xrightarrow{U!e} \langle [\Box, \varepsilon] \rangle \Rightarrow \langle \varepsilon, [U!e] \rangle \in \text{s-mgu}(U!e, U!e)
\]

and

\[
\langle [U?e, \varepsilon] \rangle \xrightarrow{U?e} \langle [\Box, \varepsilon] \rangle \Rightarrow \langle \varepsilon, [U?e] \rangle \in \text{s-mgu}(U?e, U?e)
\]

(\text{Seq})

\[
\text{dec}(\langle G_1, \varepsilon \rangle) \xrightarrow{\lambda} \{\langle \Box, \delta_1 \rangle\}
\]

\[
\Rightarrow \langle \delta_1', t_1 \rangle \in \text{s-mgu}(G_1, G_1) \text{ and } \delta|_{G_1} = \delta|_{G_1}\]
and 
\[
\text{dec}(\langle G_2 \theta_1, e \rangle) \overset{t_2}{\rightarrow} \{\langle \square, \theta \rangle\}
\]
\[
\Rightarrow \langle \theta', t_2 \rangle \in \text{s-mgu}(G_2 \theta_1, G_2') \quad \text{and} \quad \theta'_{G_2 \theta_1} = \theta'_{G_2}
\]

imply
\[
\text{dec}(\langle G_1; G_2, e \rangle) \overset{\langle \square, \theta \rangle}{\rightarrow} \{\langle \square, \theta \rangle\}
\]
\[
\Rightarrow \langle \theta', t_1 t_2 \rangle \in \text{s-mgu}(G_1; G_2, G_1'; G_2') \quad \text{and} \quad \theta'_{G_1; G_2} = \theta'_{G_1; G_2}
\]

Given the two computations in the premise, \(\text{dec}(\langle G_1; G_2, e \rangle) \overset{\langle \square, \theta \rangle}{\rightarrow} \{\langle \square, \theta \rangle\}\) can be derived. In fact, from the first one, we have \(\text{dec}(\langle G_1; G_2, e \rangle) \overset{t_1 t_2}{\rightarrow} \{\langle \square, \theta \rangle\}\) is also derivable, and thus \(\theta = \theta_1 \theta_2\).

We have to prove that

a) \(\exists \theta': \langle \theta', t_1 t_2 \rangle \in \text{s-mgu}(G_1; G_2, G_1'; G_2')\)

b) \(\theta'_{G_1; G_2} = \theta'_{G_1; G_2}\).

Since new, refreshed clauses (variables) are used in the refutation process, and these variables can be chosen so that they do not appear in \(G_i\) and \(G'_i\) (\(i = 1, 2\)), we have that

i) \(\theta'_{G_i} = e\), and therefore, by inductive hypothesis, \(\langle \theta'_1, t_2 \rangle \in \text{s-mgu}(G_2 \theta_1, G_2'; G_1)\) and

ii) \(\theta'_{G_1; G_2} = (\theta'_{G_1; G_2'}, \theta'_{G_2; G_1'})_{G_1; G_2}\).

Moreover, since we choose \(G'_1\) and \(G'_2\) such that they do not share variables neither with \(G_1\) and \(G_2\), respectively, nor mutually,

iii) \((\theta'_1; \theta'_2)_{G_1; G_2} = (\theta'_1; \theta'_2)_{G_1; G_2'}\)

iv) \(\theta'_1; \theta'_2 = e = \theta'_1; \theta'_2\).

Therefore,

a) by i), iv), and definition of s-mgu, \(\langle \theta', t_1 t_2 \rangle \in \text{s-mgu}(G_1; G_2, G_1'; G_2')\) and

b) \(\theta'_{G_1; G_2} = \theta'_1; \theta'_2\).

(Choice)
\[
\text{dec}(\langle G, e \rangle) \overset{t}{\rightarrow} \{\langle \square, \theta \rangle\} \Rightarrow \langle \theta', t \rangle \in \text{s-mgu}(G, G') \quad \text{and} \quad \theta'_{G} = \theta'_{G'}
\]

and
\[
G' = G^* \text{ with } \sigma = \text{mgu}(G^*, G^*) \quad \text{and} \quad \text{Vars}(G^*) \cap \text{Vars}(G') = 0 \quad \text{imply}
\]
\[
\text{dec}(\langle G \land G^*, e \rangle) \overset{t}{\rightarrow} \{\langle \square, \theta \rangle\}
\]
\[
\Rightarrow \langle \theta^*, t \rangle \in \text{s-mgu}(G \land G^*, G^* \land G^*) \quad \text{and} \quad \theta'_{G \land G^*} = \theta'_{G^* \land G}.
\]

and
\[
\text{dec}(\langle G^* \land G, e \rangle) \overset{t}{\rightarrow} \{\langle \square, \theta \rangle\}
\]
\[
\Rightarrow \langle \theta^*, t \rangle \in \text{s-mgu}(G^* \land G, G^* \land G') \quad \text{and} \quad \theta'_{G \land G^*} = \theta'_{G^* \land G}.
\]
Given the computation $\text{dec}(\langle G, e \rangle) \xrightarrow{\lambda_1} I_1 \ldots I_n \xrightarrow{\lambda_n} \{ \square, \vartheta \}$ in the premise, the computation $\text{dec}(\langle G \triangleleft G'' ', e \rangle) \xrightarrow{\lambda_1} I_1 \ldots I_n \xrightarrow{\lambda_n} \{ \square, \vartheta \}$ can also be derived. In fact, the first transition is derived by a proof involving the Choice rule. Notice that the induction on the structure of the goal in this case implies a simplification of the proof of the first transition in the computation. We have to prove that

a) $\langle \vartheta '', t \rangle \in \text{s-mgu}(G \triangleleft G'', G' \triangleleft G')$ and
b) $\vartheta''_{|G \triangleleft G''} = \vartheta''_{|G \triangleleft G'}$

A goal $G^*$ satisfying the properties in the premise can be easily found. Therefore, a) is trivially true with $\vartheta'' = \sigma + \vartheta'$. Moreover, $\sigma_{|G \triangleleft G'} = \varepsilon$ as $G'' = G * \sigma$, and $\vartheta'_{|G \triangleleft G'} = \vartheta'_{|G}$. Thus,

$$\vartheta''_{|G \triangleleft G''} = (\sigma + \vartheta')_{|G \triangleleft G'} = (\sigma_{|G \triangleleft G'} + \vartheta'_{|G \triangleleft G'})_{|G \triangleleft G''} = (\sigma + \vartheta'_{|G})_{|G \triangleleft G''} = \vartheta'_{|G \triangleleft G''}$$

(Parallel)

$$\text{dec}(\langle G_1, e \rangle) \xrightarrow{t_1} \{ \langle \square, \vartheta_1 \rangle \} \Rightarrow \langle \vartheta_1', t_1 \rangle \in \text{s-mgu}(G_1, G'_1) \text{ and } \vartheta_{|G_1} = \vartheta'_{|G_1}$$

and

$$\text{dec}(\langle G_2, e \rangle) \xrightarrow{t_1} \{ \langle \square, \vartheta_2 \rangle \} \Rightarrow \langle \vartheta_2', t_2 \rangle \in \text{s-mgu}(G_2, G'_2) \text{ and } \vartheta_{|G_2} = \vartheta'_{|G_2}$$

and $t = t_1 \cdot t_2$ and $\vartheta_1 + \vartheta_2 = \vartheta$ and $\vartheta_1' + \vartheta_2' = \vartheta'$ imply

$$\text{dec}(\langle G_1 \parallel G_2, e \rangle) \xrightarrow{t_1} \{ \langle \square, \vartheta \rangle \}$$

$$\Rightarrow \langle \vartheta', t \rangle \in \text{s-mgu}(G_1 \parallel G_2, G'_1 \parallel G'_2) \text{ and } \vartheta_{|G_1 \parallel G_2} = \vartheta'_{|G_1 \parallel G_2}$$

First of all, let us discuss how to derive the computation $\text{dec}(\langle G_1 \parallel G_2, e \rangle) \xrightarrow{t_1} \{ \langle \square, \vartheta \rangle \}$. Let us take any step $I \xrightarrow{\lambda} I'$ of the first computation in the premise. By Async, $I \text{id} \xrightarrow{\lambda} I' \text{id}$ is also derivable. By substituting every step $I \xrightarrow{\lambda} I'$ with $I \text{id} \xrightarrow{\lambda} I' \text{id}$, we obtain a new subcomputation called right-context idle computation, symmetrically, for the second computation in the premise. The computation of the consequent is obtained by interleaving and/or synchronizing the steps of the two context idle computations according to the merging of the two traces $t_1$ and $t_2$. Finally, according to the rule Join, a further step has to be added: the substitution unifier step.

Declaratively, $\langle \vartheta', t \rangle \in \text{s-mgu}(G_1 \parallel G_2, G'_1 \parallel G'_2)$ by inductive hypothesis since the additional requirements about the separation of the environment can be easily satisfied by choosing suitable variants of the involved goal components. Note that

$$\vartheta_{|G_1 \parallel G_2} = (\vartheta_1 + \vartheta_2)_{|G_1 \parallel G_2} = (\vartheta_{|G_1} + \vartheta'_{|G_2})_{|G_1 \parallel G_2} = (\vartheta_1 + \vartheta'_{|G_1})_{|G_1 \parallel G_2} = \vartheta'_{|G_1 \parallel G_2} = \vartheta'_{|G_1 \parallel G_2}$$

where the equalities hold due to the extra conditions about separation.

(Res)

$$\text{dec}(\langle G, e \rangle) \xrightarrow{t_1} \{ \langle \square, \psi \rangle \} \Rightarrow \langle \psi', t \rangle \in \text{s-mgu}(G, G') \text{ and } \psi_{|G} = \psi'_{|G}$$

and

$$t \setminus e = t$$
imply
\[ \text{dec}(\langle G \setminus e, \varepsilon \rangle) \xrightarrow{1} \{\langle \Box, \vartheta \rangle\} \]
\[ \Rightarrow \langle \vartheta', t \rangle \in \text{s-mgu}(G \setminus e, G' \setminus e) \text{ and } \vartheta_{|G \setminus e} = \vartheta'_{|G \setminus e} \]
The proof is straightforward. Actually, since \( t \setminus e = t \) and \( \langle \psi', t \rangle \in \text{s-mgu}(G, G') \), then \( \langle \psi', t \rangle \in \text{s-mgu}(G \setminus e, G' \setminus e) \).

In order to prove the completeness part, we need the following lemma.

**Lemma 7.4.** Given two goals \( G \) and \( G' \) such that \( \text{Vars}(G) \cap \text{Vars}(G') = \emptyset \) and a substitution \( \sigma \), if there exists \( \text{mgu}(G, G' \sigma) \), then there also exists \( \text{mgu}(G, G') \).

**PROOF.** Immediate. 

**Theorem 7.5 (Completeness).** Given a DL-system \( \Sigma_{P, G} \), let us assume that \( \exists G' \) such that \( \langle \vartheta', t \rangle \in \text{s-mgu}_{M_p}(G, G') \). Then there exists a computation
\[ \text{dec}(\langle G, \varepsilon \rangle) \xrightarrow{1} \{\langle \Box, \vartheta \rangle\} \]
\[ \text{such that } \vartheta_{|G} = \vartheta'_{|G}. \]

**PROOF.** The proof is by structural induction on \( G \), and for the case of an atomic goal \( A \), the induction relies on the proof of the membership of \( A \) to the minimal model (see rule Derive below). Some rules are not justified since they can be proved by simply reverting to the arguments of the corresponding rules in Theorem 7.3.

(\text{Unit})
\[ A' \leftarrow \text{is a variant (sharing no variables with } A \text{) of a unit clause in } P \]
implies
\[ \langle \vartheta', \lambda \rangle \in \text{s-mgu}(A, A') \Rightarrow \text{dec}(\langle A, \varepsilon \rangle) \xrightarrow{1} \{\langle \Box, \vartheta \rangle\} \text{ and } \vartheta'_{|A} = \vartheta_{|A} \]
Since the unit clause \( A' \leftarrow \) is true in \( M_p, \langle A', \lambda \rangle \in M_p \). The fact that \( \text{mgu}(A, A') = \vartheta' \) is implies by \( \langle \vartheta', \lambda \rangle \in \text{s-mgu}(A, A') \). The (single step) computation can be derived iff \( \text{mgu}(A, A') = \vartheta \), thanks to a proof that involves rules \text{Atomic} and \text{Act}.

(\text{Derive})
\[ \langle \psi', t \rangle \in \text{s-mgu}(G \delta, G') \Rightarrow \text{dec}(\langle G \delta, \varepsilon \rangle) \xrightarrow{1} \{\langle \Box, \psi \rangle\} \text{ and } \psi_{|G \delta} = \psi'_{|G \delta} \]
and
\[ A' \leftarrow G \text{ is a variant (sharing no variables with } A \text{) of a clause in } P \]
and
\[ \langle A'', t \rangle \in M_p \text{ because of } \langle \gamma, t \rangle \in \text{s-mgu}(G, G') \text{ and } A'' = A' \gamma \]
and
\[ \text{if } \exists \text{mgu}(A, A') \text{ then } \text{mgu}(A, A') = \delta \]
imply
\[ \langle \vartheta', t \rangle \in \text{s-mgu}(A, A'') \Rightarrow \langle A, \varepsilon \rangle \Rightarrow \text{dec}(\langle G \delta, \delta \rangle) \]
\[ \xrightarrow{1} \{\langle \Box, \vartheta \rangle\} \text{ and } \vartheta'_{|A} = \vartheta'_{|A} \]
In order to have \( \langle \vartheta', t \rangle \in \text{s-mgu}(A, A'') \), \( \text{mgu}(A, A'') \) has to be equal to \( \vartheta' \). Thanks to Lemma 7.4, \( \text{mgu}(A, A') \) is defined, and let us assume it to be \( \delta \). Thus,
operationally, the first step \( (A, \varepsilon) \to \text{dec}(\langle G\delta, \delta \rangle) \) can be derived, and by the inductive hypothesis and Lemma 7.1, \( \text{dec}(\langle G\delta, \delta \rangle) \xrightarrow{1} \{ \langle \square, \theta \rangle \} \) is also derivable. Therefore, \( \theta = \delta \psi, \) and \( \theta_A = (\delta_A \psi_{G\delta})_A \). By the inductive hypothesis, \( \psi' = \text{mgu}(G\delta, G') = \psi'_{G\delta} \cup \psi'_{G'G}, \) and \( \psi'_{G\delta} = \psi'_{G\delta} \), where \( G' \) and \( G\delta \) (and \( G \)) do not share variables mutually.

We note that

- \( \delta(\psi'_{G\delta}) \cup \psi'_{G'} \) is a unifier of \( G \) and \( G' \). Thus, \( \text{mgu}(G, G') = \gamma \leq \delta(\psi'_{G\delta}) \cup \psi'_{G'} \), and therefore \( \gamma_{|G} \leq (\delta_{|A} \psi'_{G\delta}), \)

- \( A'' = A' \gamma_{|G} \leq A'(\delta_{|A} \psi'_{G\delta}) = A(\delta_{|A} \psi'_{G\delta}), \) and let \( \chi \) be the least substitution such that \( A'(\gamma_{|G}) = A'' = A(\delta_{|A} \psi'_{G\delta}). \)

Therefore, \( \theta' = \chi \cup (\delta_{|A} \psi'_{G\delta}) \) is the most general unifier of \( A \) and \( A'' \), and then \( \theta'_{|A} = (\delta_{|A} \psi'_{G\delta})_A = (\delta_{|A} \psi'_{G\delta})_A = \theta_A. \)

(Event)

\( \langle \varepsilon, [\text{U}!e] \rangle \in \text{s-mgu}(\text{U}!e, \text{U}!e) \Rightarrow \{ \langle \text{U}!e, \varepsilon \rangle \} \xrightarrow{\text{U}!e} \{ \langle \square, \varepsilon \rangle \} \)

and

\( \langle \varepsilon, [\text{U}?e] \rangle \in \text{s-mgu}(\text{U}?e, \text{U}?e) \Rightarrow \{ \langle \text{U}?e, \varepsilon \rangle \} \xrightarrow{\text{U}?e} \{ \langle \square, \varepsilon \rangle \} \)

(Seq)

\( \langle \theta'_1, t_1 \rangle \in \text{s-mgu}(G_1, G'_1) \Rightarrow \text{dec}(\langle G_1, \varepsilon \rangle) \xrightarrow{1} \{ \langle \square, \theta_1 \rangle \} \) and \( \theta_{G_1} = \theta'_{G_1} \)

and

\( \langle \theta'_2, t_2 \rangle \in \text{s-mgu}(G_2, \theta_1, G'_2) \Rightarrow \text{dec}(\langle G_2, \theta_1, \varepsilon \rangle) \xrightarrow{2} \{ \langle \square, \theta_2 \rangle \} \) and \( \theta_{G_2;\theta_1} = \theta'_{G_2;\theta_1} \)

and

\( \langle \theta', t_1 t_2 \rangle \in \text{s-mgu}(G_1;G_2, G'_1;G'_2) \Rightarrow \text{dec}(\langle G_1;G_2, \varepsilon \rangle) \xrightarrow{1+2} \{ \langle \square, \theta \rangle \} \) and \( \theta_{G_1;G_2} = \theta'_{G_1;G_2} \)

(Choice)

\( \langle \theta', t \rangle \in \text{s-mgu}(G, G') \Rightarrow \text{dec}(\langle G, \varepsilon \rangle) \xrightarrow{1} \{ \langle \square, \theta \rangle \} \) and \( \theta_{G} = \theta'_{G} \)

and

\( G^{*} = G'' \) with \( \sigma = \text{mgu}(G^{*}, G'') \), \( G'' = G^{*}\sigma \), and \( \text{Vars}(G^{*}) \cap \text{Vars}(G') = \emptyset \)

imply

\( \langle \theta'', t \rangle \in \text{s-mgu}(G \odot G'', G' \odot G^{*}) \Rightarrow \text{dec}(\langle G \odot G'', \varepsilon \rangle) \xrightarrow{1} \{ \langle \square, \theta \rangle \} \) and \( \theta_{G \odot G} = \theta''_{G \odot G} \)

and

\( \langle \theta'', t \rangle \in \text{s-mgu}(G'' \odot G, G^{*} \odot G') \Rightarrow \text{dec}(\langle G'' \odot G, \varepsilon \rangle) \xrightarrow{1} \{ \langle \square, \theta \rangle \} \)

and \( \theta_{G \odot G} = \theta''_{G \odot G} \).
(Parallel)
\[ \langle \vartheta', t_1 \rangle \in \text{s-mgu}(G_1, G'_1) \Rightarrow \text{dec}(\langle G_1, \vartheta \rangle) \xrightarrow{\text{1}\text{1}} \{\langle \varnothing, \vartheta_1 \rangle\} \text{ and } \vartheta_{1|G_1} = \vartheta'_{1|G_2} \]
and
\[ \langle \vartheta_2', t_2 \rangle \in \text{s-mgu}(G_2, G'_2) \Rightarrow \text{dec}(\langle G_2, \vartheta \rangle) \xrightarrow{\text{1}\text{1}} \{\langle \varnothing, \vartheta_2 \rangle\} \text{ and } \vartheta_{2|G_2} = \vartheta'_{2|G_2} \]
and \( t = t_1 \cdot t_2 \) and \( \vartheta_1 + \vartheta_2 = \vartheta \) and \( \vartheta_1' + \vartheta_2' = \vartheta' \)

imply
\[ \langle \vartheta', t \rangle \in \text{s-mgu}(G_1||G_2, G'_1||G'_2), \]
\[ \Rightarrow \text{dec}(\langle G_1||G_2, \vartheta \rangle) \xrightarrow{\text{1}\text{t}} \{\langle \varnothing, \vartheta \rangle\} \text{ and } \vartheta_{G_1||G_2} = \vartheta'_{G_1||G_2} \]

(Res)
\[ \langle \psi', t \rangle \in \text{s-mgu}(G, G') \Rightarrow \text{dec}(\langle G, \vartheta \rangle) \xrightarrow{\text{1}\text{t}} \{\langle \varnothing, \psi \rangle\} \text{ and } \psi_{|G} = \psi'_{|G} \]

and
\[ t \setminus e = t \]

imply
\[ \langle \vartheta', t \rangle \in \text{s-mgu}(G \setminus e, G' \setminus e) \Rightarrow \text{dec}(\langle G \setminus e, \vartheta \rangle) \xrightarrow{\text{1}\text{t}} \{\langle \varnothing, \vartheta \rangle\} \]
\[ \text{ and } \vartheta_{G \setminus e} = \vartheta'_{G \setminus e} \]

Looking at the set of rules we have introduced in the proofs of the soundness and completeness theorems, we realize that our declarative semantics can be seen as a sort of more abstract operational semantics. The Unit rule states that all of the instances of an atom in a unit clause are true and refutable with the same correct answer substitution. The other rules allow us to compute correct answer substitutions in a much more clear, compact, abstract, and compositional way. When deriving \( \langle \vartheta', t \rangle \in \text{s-mgu}(G, G') \), we know that \( G \) has at least a refutation with computed answer substitution \( \vartheta \) such that \( \vartheta_{|G} = \vartheta'_{|G} \), but not all of the details about such a refutation are contained in its proof: the order of independent steps in a parallel goal is left unspecified. In a sense, we can say that if the proof for \( \langle \vartheta', t \rangle \in \text{s-mgu}(G, G') \) corresponds to the computation \( \xi \) such that \( \text{last}(\xi) = \{\langle \varnothing, \vartheta \rangle\} \), then it also corresponds to any other computation \( \xi' \) such that its partial ordering is the same as \( \xi \), but with a different temporal (or generation) ordering.

8. CONCLUSIONS AND RELATED WORK

The definition of Distributed Logic given in [36] includes the constraint that messages must become ground after communication is performed. To solve an event goal \( U?e \), there must be a complementary event goal \( U'!e \) to be simultaneously solved such that

i) \( \exists \vartheta = \text{mgu}(U, U') \), and
ii) both \( (U)\vartheta \) and \( (U')\vartheta \) are ground terms.

In the study of the semantics of DL, we have considered ground communications only so that constraint ii) is always satisfied. However, the operational semantics presented in Section 6 can be easily extended to model possibly non-ground communications. A detailed description of how to do this is reported in [9].
Precisely, the premise of the Event inference rule is withdrawn, while additional constraints concerning unification and groundization are added to the Sync rule. On the other hand, it is not easy to provide nonground communications with a declarative meaning. When composing two goals in parallel, each of them true with some trace, we check whether they can communicate a substitution when considering nonground communication terms. Correspondingly, the definition of trace merging (Definition 5.1) must be changed to model such an operational behavior. Summing up, a model-theoretic treatment of nonground communications introduces very operational denotations into interpretations.

We now briefly survey some interesting related works.

A comparison with the semantics defined by Monteiro is mandatory. We have borrowed from [33] the idea of relating the truth of a goal with the history of its (open) communications. However, some deep differences are quite evident. First, our declarative semantic is based on the interpretations containing possibly nonground atoms, while Monteiro uses models defined in terms of ground atoms. Second, we focus on the study of the operational semantics, which is defined in terms of the distributed model of Petri nets. We have also presented in Section 2 the basic distributed model of computation underlying DL through the definition of a suitable subset of DL (CSLP).

In [12], an algebraic approach to the semantics of concurrent logic languages is presented. Following the same intuition we have presented here, a process semantics is shown to be superior to the usual final state semantics. Moreover, the authors prove in a categorical framework that the declarative and operational semantics naturally coincide.

In [32], a truly concurrent description of the cc paradigm is presented. However, such a semantics of cc languages does not offer any hint for an actual implementation of the languages. In the cc paradigm, parallel processes share a store of constraints, as STREAM-parallel processes share variables. Whatever the description of the operational semantics is, the compatibility of constraints (respectively, variables) independently computed by different processes must be checked. Notice that this is not the case for DL, and for CSLP, where parallel processes can work in parallel.

Shapiro surveyed in [45] the family of concurrent logic programming languages, with particular emphasis on the class of languages imposing constraints on the unification mechanism so that a distinction between the producer(s) and the consumer(s) of the values of a logical variable is introduced (STREAM-languages). In contrast, little attention is paid to the orthogonal line followed in the present paper. Shapiro complains that DL (and its programmative counterpart Delta Prolog): “is not a logic programming language in the sense that a successful computation corresponds to a proof of a goal statement,” and also that: “the role of the communication primitives of Delta Prolog in the declarative reading of programs is unclear.” We think that the presented semantics completely overcomes these criticisms.

However, we do not want to assess the superiority of one approach over the other (in fact, Delta Prolog also has some drawbacks, e.g., it is not a “reactive” system since it can backtrack on communication), but to demonstrate the interest the Distributed Logic model can have. We think it could be interesting to better study the Delta Prolog model of computation, maybe trying to integrate it with the STREAM-parallel approach.
A first possible approach has been defined in [8], where a somewhat mixed model of computation has been proposed. The basic idea is to define a committed-choice language where the communication mechanism is realized by a shared blackboard which represents the only centralization point of the model. Sequential computations can be derived in parallel, and the corresponding parallel processes synchronize via the shared blackboard.

A communication model for distributed logic programming has been studied in [7]. As for the case of CSLP, parallel processes do not share variables, while process synchronization is ruled by multiple-headed clauses.

Finally, we would like to mention another current research direction in concurrent logic programming, whose main concern is to exploit the inherent concurrency of Prolog programs by implementing Prolog on distributed systems, rather than extending the language with new primitives for managing concurrency and synchronization (as we have done here). The interested reader may refer to [38] and to the references therein.

The work of A. Brogi was supported in part by "CNR Progetto Finalizzato Sistemi Informatici e Calcolo Parallelo" under Grant 92.01564.PF69. The work of R. Gorrieri was supported in part by Esprit BRA 6360 "Broadcast" and in part by MURST 40%. We would like to thank L. Monteiro for valuable discussions and suggestions. Thanks also to A. Corradini, P. Degano, and C. Palamidessi. The pertinent comments of one the referees have contributed to improving the overall content of the paper.

REFERENCES


43. Shapiro, E., A Subset of Concurrent Prolog and its Interpreter, in: [45].