On the Operational Semantics of a Coordination Language

Paolo Ciancarini
Università di Bologna, Italy
E-mail: cianca@cs.unibo.it

Keld K. Jensen
University of Aalborg, Denmark
E-mail: kondrup@iesd.auc.dk

Daniel Yankelevich
Universidad de Buenos Aires, Argentina
E-mail: dany@se.uba.ar

Abstract. Linda is a coordination language, because it has to be combined with a sequential language to give a full parallel programming formalism. Although Linda has been implemented on a variety of architectures, and in combination with several sequential languages, its formal semantics is relatively unexplored. In this paper we study and compare a number of operational semantics specifications for Linda: Plotkin's SOS, Milner's CCS, Petri Nets, and Berry and Boudol's Chemical Abstract Machine. We analyze these specifications, and show how they enlighten different abstract implementations.

1 Introduction

Languages for programming parallel systems have been defined and used for many years without significant changes in the basic communication primitives. Programming languages providing constructs for explicit parallelism are usually based on sequential processes and some set of synchronization and communication primitives. Two processes interact either through atomic operations on a shared resource (e.g., a semaphore or monitor), or by **send** and **receive** operations naming the peer process or an explicit channel over which the two processes communicate.

A third possibility is represented by the emerging class of parallel languages which conceptually are based on concurrent computations inside a shared data space of tuples. These languages are either imperative, like Linda [15], Associons [31], and Gamma [3], or logic-based, like Shared Prolog [5], Swarm [32], and LO [1]. The shared data space is either a multi-set of tuples (Linda, Associons, Gamma) or of logic terms (Shared Prolog, Swarm, LO). Furthermore, communication is based on some form of pattern matching, and is both associative and asynchronous.

Several common features of these languages suggest that their semantics can be formally defined and studied in a unified framework, aiming at clarifying which are the basic problems in implementing them. In this paper we explore some abstract models of concurrency that are interesting candidates for establishing such a theoretical framework useful to abstractly studying their implementation. As a specific example we concentrate on Linda; interestingly, Linda

[15] has been implemented on a variety of architectures, but its abstract semantics is comparatively less known. Some early and partial attempts by people in the Linda group can be found in [27, 21], where some semantics aspects relevant for its implementation are informally discussed. The Linda semantics is more formally addressed in two recent papers [17, 6], but certainly there is room for more investigations. In fact, in [17] a Linda extension is addressed (introducing first class tuple spaces), whereas in [6] a completely declarative specification of Linda in Z is given. We are here interested in an operational definition of Linda, that could be used in guiding the implementation of the language on different architectures.

The paper is structured as follows: in Section 2 we outline and formally specify Linda: we show a simple programming example and give an informal description of the language, giving also a formal syntax and semantic definition as it could be included in an ideal user manual. In section 3 we then introduce a simple "Linda Calculus" to have an abstract Linda syntax that is the basis of the "abstract implementations" given in the paper. In the next sections (4 and 5) we provide some abstract implementations based on SOS and CCS. An abstractly distributed implementation is given using Petri Nets in Section 6. In Section 7 we provide a Chemical Abstract Machine for Linda. In Section 8 a more abstract, "denotational", semantics is outlined, as a simple conceptual framework in which semantical equivalences among the different abstract implementations and between programs can be proved.

2 Linda

Linda was introduced in [15]. The basic work on its implementation was [7]. Several implementations were subsequently given by people in Yale and in other places, combining the basic model with different sequential languages and implementing it on several different architectures. For a discussion on the differences among these implementations, see [21].

Although Linda is widely known today, its formal definition is still lacking. We know only one main attempt, namely the one described in [6], that used a non-executable specification language, namely Z, to formalize Linda syntax and semantics.

Here we intend to follow a different path: we will define the language syntax using BNF, then we will describe formally the main data structures and domains, and a simple operational semantics, as could be given in an ideal "user manual" of the language. Thus, in this section we outline and formally specify the Linda concept independently from implementation; we use a formal notation to help users to understand what happens in the tuple space.

2.1 Coordination and Computation

To become a complete programming language, Linda has to be *embedded* in a sequential host language which provides data types and sequential control

primitives. The gross result is that the sequential language is extended with the four Linda primitives, but several important details have to be taken into account.

Linda (sequential) processes can perform operations on the tuple space. Seen from the tuple space, such processes are computations which, from time to time, request interaction through one of the four Linda primitives; the interaction is based on the form of a tuple. Conceptually a Linda process is simply a function, $p = \lambda t.e[t]$, which takes a tuple, t, as input and uses this to evaluate the process expression, $e[t]$. This expression either evaluates to a request for tuple space interaction and a new continuation for the process, $p \rightarrow op.(\lambda t.e[t])$, or simply into a typed value, $p \rightarrow v$, as the result of its computation.

Whenever a process issues a Linda operation, i.e., it becomes an expression of the form $op.(\lambda t.e[t])$, the tuple space carries out the operation on behalf of the process by creating the requested tuple (**eval** and **out**), or finding a match (**rd** and **in**). When the operation eventually completes, the tuple space invokes the continuation of the process with the resulting tuple as input, i.e., the tuple space enables the transition $op.(\lambda t.e[t]) \rightarrow (\lambda t.e[t])(t'')$ under the side-effect on the tuple space demanded by *op*. Here, the input tuple, t'', is the match found as a result of an **rd** or **in** operation, or some "empty" tuple for **eval** and **out** operations.

Thus, a Linda process is an autonomous process which sometimes requests the creation, copying, and removal of tuples. The behavior of a sequential Linda process, $p \in \mathit{Process}$, is expressed is defined by the following grammar:

$$p ::= op.e \mid t.e \mid e \mid v \tag{1}$$

The first production denotes a process which is able to perform a Linda operation, $op \in Op$, the second a process which has been provided with a tuple, $t \in \mathit{Tuple}$, as input; we shall formally specify these in the next section. The last two productions denote respectively local computation and termination with a final value: $e \in \Gamma_p$ is a configuration of the process which corresponds to some local computation.

We shall omit the details of the language in which local process computations are expressed and just assume some kind of continuation semantics. That is we shall assume an operational semantics for sequential Linda processes: $\mathit{Linda}_p = < \Gamma_p, \rightarrow_p, \mathcal{I}_p, \mathcal{T}_p >$. Here, $t.e \in \mathcal{I}_p$ is the set of "initial" states, $e \in \Gamma_p$ the set of intermediate states, while $v \in \mathcal{T}_p$ and $op.e \in \mathcal{T}_p$ are the set of "final" states. That is, the transition relation for Linda processes, $\rightarrow_p \subseteq \Gamma_p \times \Gamma_p$, defines the set of computations $\{t.e\} \rightarrow \{op.e,\ v\}$. Similarly, the Linda coordination language defines the set of coordinations $\{op.e\} \rightarrow \{t.e\}$ as defined by the semantics of Linda's tuple space.

As just described, this cycle of local computation and tuple space interaction is repeated until the process becomes a typed value; the evaluation of a Linda process is performed autonomously and independently with respect to the other activities in tuple space. Thus, seen from tuple space, a Linda process is abstractly a sequence of Linda operations terminated by the creation of

a tuple containing its final value, $p = op_1.op_2 \ldots op_n.v$. The tuple space does not know anything about the internal behavior (local computational model) of Linda processes, only that they are able to issue Linda operations.

2.2 Tuples and the Matching Rules

We need now to specify the concept of tuple and associative matching. The unit of interaction is the tuple, and the principle of coordination is associative addressing of tuples by pattern matching. Thus, we here define active and passive tuples and the data items which may occur as fields of these. Then we define the *match* relation between tuples, and the representation of Linda operations and tuple space configurations.

The tuple space is a multi-set of tuples (active or passive); a tuple is an ordered collection of fields. Each field is either a value-yielding Linda process or a typed value; the type name associated with each value denotes the underlying domain. No definition of the domains supported by the tuple space exists in the Linda literature; usually it is inferred from the host language in which Linda is embedded.

Let *Type* denote the type set supported by the tuple space, and let $\tau \in$ *Type* range over this set. Furthermore, let $\perp$ denote a formal (i.e., unbound) of any type; then the set of typed data values (actuals and formals) is defined as $Value = \bigcup_\tau \{a : \tau \mid a \in \mathcal{V}_\tau\} \cup \{\perp : \tau\}$; here $\mathcal{V}_{int} = Nat$, etc.

Given the definition of typed values, the set of passive Linda tuples is simply the union of all tuples of typed data values, $Tuple = \bigcup_{i>0} Value^i$ (where $Value^1 = Value$ and $Value^{i+1} = Value^i \times Value$). Similarly, if *Process* denotes the set of all Linda processes, the set of general Linda tuples (i.e., tuples in which active fields may occur) is defined as $Active = \bigcup_{i>0}(Value \cup Process)^i$. Throughout the paper we shall use the notation $\#t = n$ to denote the arity of a tuple, $t \in Value^n$, and t_i to denote the ith projection, i.e., the ith field of the tuple. Furthermore, we shall also use the notation $t'[i : p]$ to denote a tuple, t', where the ith field has the value p (to be a process).

Given the specification of tuples, we now define the *match* predicate, i.e., the associative addressing of tuples. Formally, it is a relation between pairs of (passive) tuples, $match : Tuple \times Tuple$, where the corresponding predicate yields *true* if two tuples are in the relation and *false* if not. Two tuples match if every corresponding pair of fields matches; two fields match if they have same type and exactly one is a formal or both have the same actual value. Thus, *match* is defined by the following relation (Linda does not allow either the nesting of tuples or the matching of variables, although the relation includes this for generality reasons):

$$\begin{aligned} &\forall \tau \in Type,\ a \in \mathcal{V}_\tau : \{(a : \tau, a : \tau),\ (a : \tau, \perp : \tau),\ (\perp : \tau, a : \tau)\} \in match \\ &\quad \forall s, t \in Value^n : \textstyle\bigwedge_{i=1}^{n}(s_i, t_i) \in match \ \Leftrightarrow\ (s, t) \in match \end{aligned} \tag{2}$$

By definition, a Linda process is a value-yielding computation which, when instantiated on some input, performs a sequence of Linda operations and local computations before the process eventually terminates with its final value. Here, a Linda operation is simply one of the four Linda primitives with a tuple as argument, $Op = \{eval(t) \mid t \in Active\} \cup \{out(s), rd(s), in(s) \mid s \in Tuple\}$. It is the task of the process to evaluate the syntactic Linda operations to their semantic representation, computing by local computations a final value, $v \in Value$.

$$
\begin{array}{lll}
\text{Type names:} & Type & = \{int, char, \ldots\} \\
\text{Typed values:} & Value & = \bigcup_{\tau \in \text{Type}} \{a : \tau,\ \bot : \tau \mid a \in \mathcal{V}_\tau\} \\
\text{Passive tuples:} & Tuple & = \bigcup_{i>0} Value^i \\
\text{General tuples:} & Active & = \bigcup_{i>0} (Value \cup Process)^i \\
\text{Linda operations:} & Op & = \{eval(t) \mid t \in Active\} \\
 & & \cup\ \{out(s), rd(s), in(s) \mid s \in Tuple\} \\
\text{Linda processes:} & Process & ::= \Gamma_p, \text{ generated of the grammar 1} \\
\text{Tuple space:} & TS & = \biguplus_{t \in Active} \{\!|\ t : \#[t]\ |\!\}
\end{array}
$$

Fig. 1. Domain definitions for the tuple space

Finally, the configuration of the tuple space is as a multi-set of tuples: $TS = \biguplus_{t \in Active} \{\!|\ t : \#[t]\ |\!\}$, where $\#[\mathcal{D}] : \mathcal{D} \to Nat$ is an occurrence function for multi-sets. Here, $\uplus$ denotes multi-set union and $\{\!|\ t : i\ |\!\}$ denotes the multi-set with i occurrences of the element t (when i is 1 we may omit it). The multi-set union operator $\biguplus$ has the property $\{\!|\ t\ |\!\} \biguplus \{\!|\ t\ |\!\} = \{\!|\ t,\ t\ |\!\} = \{\!|\ t : 2\ |\!\} \neq \{\!|\ t\ |\!\}$. The usual set operators have the natural interpretation for multi-sets too; for example, the subset and power-set operators are defined as $A \subseteq C \Leftrightarrow \exists B : A \uplus B = C$ and $2^A = \{B \mid B \subseteq A\}$. We have summarized the domain definitions in Figure 1.

2.3 Operational Semantics of Tuple Space

In the previous subsection we defined the static configurations of the tuple space, i.e., its possible states. Here, we specify the dynamics of the tuple space, i.e., we provide an operational semantics which defines the possible state changes.

The dynamics of the tuple space is defined by the behavior of autonomous Linda processes, and by their interactions through the manipulation of tuples in the tuple space.

Operational Semantics:

Tuple space:

$$\mathit{Linda} = < \Gamma, \rightarrow >$$

where $\Gamma = TS$
$\rightarrow \subseteq \Gamma \times \Gamma$

Transition rules:

Process creation:

$$\forall t \in \mathit{Active} : \{\!|\ t'[i\colon \mathit{eval}(t).e]\ |\!\} \rightarrow \{\!|\ t'[i\colon e],\ t\ |\!\}$$

Tuple creation:

$$\forall t \in \mathit{Tuple} : \{\!|\ t'[i\colon \mathit{out}(t).p]\ |\!\} \rightarrow \{\!|\ t'[i\colon p],\ t\ |\!\}$$

Tuple copying:

$$\forall (s,t) \in \mathit{match} : \{\!|\ t'[i\colon \mathit{rd}(s).p],\ t\ |\!\} \rightarrow \{\!|\ t'[i\colon t.p],\ t\ |\!\}$$

Tuple removal:

$$\forall (s,t) \in \mathit{match} : \{\!|\ t'[i\colon \mathit{in}(s).p],\ t\ |\!\} \rightarrow \{\!|\ t'[i\colon t.p]\ |\!\}$$

Local transition:

$$\frac{p' \rightarrow_p p''}{\{\!|\ t'[i\colon p']\ |\!\} \rightarrow \{\!|\ t'[i\colon p'']\ |\!\}} \qquad \frac{ts' \rightarrow ts''}{ts \uplus ts' \rightarrow ts \uplus ts''}$$

Abbreviations:

Configurations:

$$p, p', p'' \in \mathit{Process},\ s, t \in \mathit{Tuple},\ t', t'' \in \mathit{Active},\ ts, ts', ts'' \in TS$$

Fig. 2. Operational semantics for Tuple Space

The behaviour of the tuple space is defined by a pair called *Linda* in Figure 2. It is defined in terms of the domain specifications, the *match* relation, and the operational semantics for Linda processes, Linda_p, as defined in the previous subsection: Γ is the set of tuple space configurations and $\rightarrow$ the transition relation for this transition system. As usual when parallelism is specified by an operational semantics, concurrency is described by an arbitrary interleaving of a set of atomic transitions performed by the acting processes [25].

The interleaving approximation does not provide true concurrency, but it

is sufficient as tuples are indivisible units which are manipulated by atomic operations. The **eval** and **out** operations are non-blocking output operations, while the **rd** and **in** operations are blocking input operations. The nonblocking vs. blocking features are modeled by the *match* relation: an output operation may always take place since there is no guard, while an input operation must find a matching tuple in the tuple space before the acting process may progress.

The tuple matched by an **rd** or **in** operation is provided as input to the process (the notation for this is $t.p$). The transition function for Linda processes, $\rightarrow_p$, may then be used to perform some local computations which map the process to a new process prefixed by a Linda operation, or its final value. This use of the transition function for processes is reflected in the first inference rule for *local transitions*: whenever a process may perform some local computation, the tuple space in which it resides as a field of an active tuple may as well. Especially when the process reaches its final state and becomes a value, $v \in \mathit{Value}$, the corresponding field of the active tuple (and possibly the entire tuple) becomes a passive one.

The second inference rule for *local transitions* specifies the overall evolution of the tuple space in terms of its components: *a transition is based on a partial view of the tuple space.* This important rule reflects the *local decision making* property of Linda: Linda operations may be performed without knowledge of the global state of the tuple space, i.e., in true parallel. This local decision making property is the source of Linda's time and space decoupling of processes [15]: the only coupling of Linda processes is through the tuples they create in, and withdraw from, the tuple space.

Given the transition rules in Figure 2 it is obvious that the semantics of **out** is a subset of the semantics of **eval**. The reason for having both is historical—initially, **eval** was not part of Linda (see e.g. [15])—and conceptual—process creation and (generative) communication are different operations. Also, the **rd** and **in** operations are much alike: they only differ in their effect on tuple space, while they are semantically identical locally in the issuing process.

Let us for a moment assume that every field of a tuple residing in the tuple space is an actual value, i.e., that no formals occur as argument to **eval** or **out** operations. In this case, the functionality of an **rd** operation may be implemented by an **in** operation followed by an **out** operation with the matched tuple as argument. However, the two cases are conceptually different as they specify different mutual exclusion properties for the addressed tuple (respectively read and write locking).

In addition, the specification given here only defines the functionality of tuple space; it does not say much about its interactive behavior—timing, fairness, scheduling, etc.

Finally, it should be pointed out that this operational semantics does not specify exactly what a Linda program is: *Linda* is not a completely specified transition system. The Linda concept extends a host language with the Linda primitives, i.e., it provides a model for parallelism through the notion of the tuple space. In principle, however, a Linda process may also interact with other pro-

cesses through the host language. That is, the interpretation of what constitutes a Linda program depends on the host language—notably its I/O model—and the environment in which the Linda system is used.

3 The Linda Calculus

The transition system *Linda* (Figure 2) can be considered a full specification of the Linda coordination language. This transition system is remarkable in two ways: first, it is defined independently of the "computational" host language in which Linda processes are expressed, and second, it is very simple because it includes six transition rules only. Thus, *Linda* may be taken as a proof that coordination and computation really can be seen as orthogonal concepts [9]. Furthermore, this specification of Linda may be seen as a prototypical example on what exactly a coordination language is.

However, though *Linda* specifies how Linda processes may coordinate with each other, Linda processes must of course also be able to perform local computations to be useful for concurrent programming. This ability is provided by embedding Linda in some sequential host language, e.g., C, FORTRAN or Prolog. Given that coordination and computation are orthogonal, there are several degrees of freedom, but also some constraints, when one has to extend a language with the Linda primitives. We shall only consider this problem briefly.

Linda is usually embedded into a sequential programming language by providing the Linda primitives as procedure-like abstractions (with side–effects). What has to be defined in this embedding is the syntactic category of the Linda operations, the order of evaluation of their arguments, and the notation (and representation) of formal and actual values, and process abstractions. Especially which expressions may be instantiated to Linda processes by the **eval** operation leaves some open design issues: Linda processes are evaluated in disjunct address spaces and thus cannot share any environment besides the tuple space.

A Linda embedding must define the above items, but each embedding can be designed independently. In addition, there is also the choice of the type system supported by the tuple space; this design issue is left to the Linda run-time system. Data objects (tuples) are communicated across the disjoint address spaces of the processes in the tuple space, and consequently only location independent data objects may be supported (i.e. pointers are ruled out). The type system supported by a Linda embedding is not necessarily the same as the one supported by the tuple space; existence of a bi-directional mapping between the two is enough. We shall not here go deeper into the problems of embedding Linda; we discuss it in details elsewhere (see [20]).

The specification of Linda given in the previous section is best thought of as a precise formulation of the intuitive understanding of the coordination language; it is neither particularly well-suited for reasoning about concurrency in Linda programs, nor for enlightening possible implementation approaches. We attack such a problem in this and the following sections by giving abstract implementations of Linda using some well-known models for concurrency.

When implementing Linda or reasoning about the concurrency in a Linda program, we prefer to abstract away the local behavior of processes. Thus, we here define a "Linda Calculus" which we shall use in the rest of the paper.

The host language in which Linda is embedded is symbolized by the e in the abstract syntax for Linda processes defined in (1). Besides local computations, a host language also provides for the internal flow of information in Linda processes: from the input tuple to its output through the Linda operations or value-yielding termination. In this paper we are only interested in modeling the coordination through the tuple space, and thus we shall ignore the internal propagation of tuples and local computations of processes. Similarly, we shall omit the aspects of Linda providing data values as bottom elements for the process domain, and only use **eval** for process creation.

Given these simplifications, a process becomes a tuple space term which is created through the **eval** primitive, and which manipulates the contents of the tuple space through **out**, **rd**, and **in** operations until it terminates. Thus, we introduce the following Linda Calculus:

$$\mathrm{P} ::= \mathbf{eval}(\mathrm{P}).\mathrm{P} \mid \mathbf{out}(t).\mathrm{P} \mid \mathbf{rd}(t).\mathrm{P} \mid \mathbf{in}(t).\mathrm{P} \mid \mathrm{X} \mid \mathbf{rec}\mathrm{X}.\mathrm{P} \mid \mathrm{P}[]\mathrm{P} \mid \mathbf{end} \qquad (3)$$

The first four terms provide the Linda operations: only a single process is created by **eval** while the argument to the other operations is a tuple, $t \in \mathit{Tuple}$, as in the previous. We have provided three additional productions: two for modeling recursive process specifications, and one for modeling local nondeterministic choice. Finally, the **end** production denotes termination. In the tuple space we find both processes generated over this grammar and passive tuples, $t \in \mathit{Tuple}$.

A tuple space term, M, is then defined as either a process, a passive tuple, or a multi-set: $M ::= \mathrm{P} \mid t \mid M \oplus M$. Here, the $\oplus$ operator denotes multi-set construction, i.e., $p \oplus p = \{\!|\ p, p\ |\!\}$; we will overload the operator to denote multi-set union as well. A last remark: Linda processes are terms without free variables, i.e. each variable **X** is bound by a corresponding **recX** construct.

Many techniques for formal verification consider systems equipped with transition system semantics. The most popular approach to operational semantics is based on Plotkin's Structured Operational Semantics (SOS) [29]; another choice is the translation into another well-known, formal parallel language like Milner's CCS [23, 25]. A third option, more useful for a physically distributed analysis of a parallel language, are Petri Nets [28, 30].

The following sections in turn models our Linda Calculus using respectively SOS, CCS, and Petri Nets.

4 Abstract Implementations by Transitions Systems

Following Plotkin's SOS style, we give a set of rules defining the operational semantics of the Linda Calculus in a simple and intuitive way. This operational semantics of course highly resembles the Linda specification we gave in Section 2.

However, where *Linda* only specifies the partial coordination behavior specified for Linda processes, this transition system now specifies the full behavior of the (simplified) Linda Calculus. Again, most of the rules are axioms, with a unique rule for recursion; in the rules we use the *match* relation as defined by Equation 2. Thus, we define the semantics of the Linda Calculus by use of SOS as follows:

Definition: *ITS*[Linda] - *Interleaving Transition System for Linda*

$$
\begin{aligned}
\textit{eval}) \quad & M \oplus \mathbf{eval}(\mathrm{P}).\mathrm{P}' \rightarrow M \oplus \mathrm{P} \oplus \mathrm{P}' \\
\textit{out}) \quad & M \oplus \mathbf{out}(t).\mathrm{P} \rightarrow M \oplus \mathrm{P} \oplus t \\
\textit{rd}) \quad & M \oplus \mathbf{rd}(s).\mathrm{P} \oplus t \rightarrow M \oplus \mathrm{P}[t/s] \oplus t, \text{ if } (s,t) \in \textit{match} \\
\textit{in}) \quad & M \oplus \mathbf{in}(s).\mathrm{P} \oplus t \rightarrow M \oplus \mathrm{P}[t/s], \qquad \text{if } (s,t) \in \textit{match} \\
\textit{rec}) \quad & \frac{M \oplus \mathrm{P}[\mathbf{rec}\mathrm{X}.\mathrm{P}/\mathrm{X}] \rightarrow M \oplus \mathrm{P}'}{M \oplus \mathbf{rec}\mathrm{X}.\mathrm{P} \rightarrow M \oplus \mathrm{P}'} \\
\textit{leftch}) \quad & M \oplus \mathrm{P}[]\mathrm{P}' \rightarrow M \oplus \mathrm{P} \\
\textit{rightch}) \quad & M \oplus \mathrm{P}[]\mathrm{P}' \rightarrow M \oplus \mathrm{P}' \\
\textit{end}) \quad & M \oplus \mathbf{end} = M
\end{aligned}
$$

Here, $\mathrm{P}[t/s]$ denotes the substitution in P of all the formal parameters of s by the actual parameters of t, i.e., the syntactic replacement of the variables of s by the corresponding values of t. Similarly, $\mathrm{P}'[\mathbf{rec}\mathrm{X}.\mathrm{P}/\mathrm{X}]$ is the usual substitution of X in P' by **rec**X.P. We remark that the rule for recursion could have been replaced by an axiom, simply by introducing the "unfolding" of a term as a primitive axiom (i.e. $M \oplus \mathbf{rec}\mathrm{X}.\mathrm{P} \rightarrow M \oplus \mathrm{P}[\mathbf{rec}\mathrm{X}.\mathrm{P}/\mathrm{X}]$).

Example: Evaluation under *ITS*

To illustrate the execution of a Linda program under *ITS*, let P be the Linda process **eval**(Q).**rd**(a).**out**(b).**end** where Q is the process **out**(a).**in**(b).**end**. If P is instantiated in an empty tuple space it may evolve as follows:

$$
\begin{aligned}
\textit{ITS}[\![\mathrm{P}]\!] &= \mathbf{eval}(\mathrm{Q}).\mathbf{rd}(a).\mathbf{out}(b).\mathbf{end} && (\textit{eval}) \\
&\rightarrow \mathbf{out}(a).\mathbf{in}(b).\mathbf{end} \oplus \mathbf{rd}(a).\mathbf{out}(b).\mathbf{end} && (\textit{out}) \\
&\rightarrow a \oplus \mathbf{in}(b).\mathbf{end} \oplus \mathbf{rd}(a).\mathbf{out}(b).\mathbf{end} && (\textit{rd}) \\
&\rightarrow a \oplus \mathbf{in}(b).\mathbf{end} \oplus \mathbf{out}(b).\mathbf{end} && (\textit{out}) \\
&\rightarrow a \oplus \mathbf{in}(b).\mathbf{end} \oplus b \oplus \mathbf{end} && (\textit{in}) \\
&\rightarrow a \oplus \mathbf{end} \oplus \mathbf{end} && (\textit{end}) \\
&= a
\end{aligned}
$$

We have to make some comments comparing this abstract implementation with the semantics of the Linda specification given in Section 2. The ITS transition system is a particular implementation of the Linda specification which has two attractive properties: it is simple, and it is property preserving. Thus, it is sufficient to prove the correctness of an implementation in terms of ITS as opposed to (the more complicated) specification. ITS defines a full Linda language providing a model for the set of possible concurrent computations. Contrariwise, the Linda specification does not provide any relation between the coordination of agents and their local computations. This especially has the drawback of not expressing anything about the causal relationship between process actions, something which ITS captures by its simple computational model (sequence, recursion, and nondeterministic choice).

In practice the interleaving semantics defined by the transition system *ITS* suggests a centralized implementation on a monoprocessor, because the transition system has a unique thread of control. We note that this does not mean that the system is deterministic, but that the transitions modify global states. At each tick of the clock, only one action can occur, and the change of state is global: from a global state to another global state. Implicitly, it is supposed that an observer that can look at the behavior of the whole system does exist, and that the time of the system is the time of this global observer.

In order to have a multithread implementation, we should extend this transition system to a *multistep transition system*, i.e., we should permit a set of processes to act in each transition. Following the ideas of SCCS [24], we add rules to express the maximum parallelism specified by a Linda program as follows:

Definition: $MTS[\![\text{Linda}]\!]$ – *Multistep Transition System*

ITS augmented with the inference rule

$$par)\ \frac{M_1 \rightarrow M_1' \quad M_2 \rightarrow M_2'}{M_1 \oplus M_2 \rightarrow M_1' \oplus M_2'}$$

In this new transition system we have more transitions than in *ITS*, because all the interleaving transitions are kept; a transition now comprises a multi-set of Linda operations. This semantics has the advantage that there may be as many control threads as processes specified in the program. However, the idea of a global state is still there, or equivalently the concept of a global clock: all the actions are synchronous, i.e., they happens in the same tick of the clock. This concept is clearly unrealistic in a distributed implementation, but it could be useful for an implementation that exploits a SIMD architecture, e.g., an array processor.

Distribution and asynchrony are not captured by an operational semantics like *ITS* and *MTS*. Here we define distribution as the inability to "observe" the entire system simultaneously, and not just as a set of distinct places. Also, problems about the granularity of atomic operations arise. In practice, several operations (with duration greater than zero) can be overlapped in time without having a common state in which they all are enabled or completed.

5 Translating Linda into CCS

An interesting way of defining the semantics of a language consists of translating the programs into another language with well-known semantics and properties. It is widely accepted that Milner's CCS [25] is a well-defined language for studying concurrency. So, it is natural to study the translation from Linda to CCS; this translation may also be seen as an implementation. The version of CCS we use here is the one introduced in [23], i.e., pure CCS with unbounded nondeterminism.

CCS is a formal language for specifying parallel systems which is based on synchronization of agents through complementary named actions. The basic agent is Nil which cannot perform any actions; a derived CCS agent may then be an agent prefixed with an action, $\alpha \in Act$, a nondeterministic choice between CCS agents, or a parallel composition of agents. Two CCS agent may synchronize on a complementary action $(\alpha, \overline{\alpha})$ and perform the τ action (silent move), or each agent may act on their own through a visible action. We may describe CCS by the following abstract syntax:

$$A ::= Nil \mid \alpha.A \mid A + A \mid (A|A) \mid X \mid recX.A \mid A \setminus S \mid A[\mathcal{F}]$$

The last two terms are respectively restriction and renaming: restriction disables the set of actions (and their complementary actions) in $S \subseteq Act$, and renaming uses $\mathcal{F}$ to rename the visible actions of A. The silent move τ cannot be restricted away, and the renaming function $\mathcal{F} : Act \rightarrow Act$ must satisfy the properties $\mathcal{F}(\overline{\alpha}) = \overline{\mathcal{F}(\alpha)}$ and $\mathcal{F}(\alpha) = \tau \Leftrightarrow \alpha = \tau$.

Besides an operational semantics for the evolution of agents, CCS also has a formalism for reasoning about agents. This formalism comprises a set of laws for process equivalences (e.g. $A|Nil = A$), and bisimulation (strong and weak) for determining equivalent (visible) behavior of two agents.

5.1 Pure CCS implementation

In order to define the translation of our Linda calculus into CCS, an encoding of Linda operations as synchronizations (actions of CCS agents) is necessary. If the set of typed values is assumed to be finite, this encoding may be done in various ways; it is not important for the translation, but may be seen as an implementation issue (i.e. how to make an encoding of one data structure in a simpler one). Here, we assume a function, *code*, which maps Linda operations into CCS actions. Moreover, we will use the notation $\sum_{t:predicate} P(t)$ to denote the nondeterministic composition of the CCS agents generated over P from the t's which satisfies the predicate.

Definition: $CCS[\![\text{Linda}]\!]$ – *translation of Linda into CCS*

$$code : \{rd, in\} \times Tuple \rightarrow Act$$

$$\begin{aligned}
tuple)\ & A[\![t]\!] && = recX.(code(rd,t).X + code(in,t).Nil) \\
eval)\ & A[\![\mathbf{eval}(\mathrm{P}).\mathrm{P}']\!] && = \tau.(A[\![\mathrm{P}]\!] \mid A[\![\mathrm{P}']\!]) \\
out)\ & A[\![\mathbf{out}(t).\mathrm{P}]\!] && = \tau.(A[\![t]\!] \mid A[\![\mathrm{P}]\!]) \\
rd)\ & A[\![\mathbf{rd}(s).\mathrm{P}]\!] && = \textstyle\sum_{t:\mathrm{match}(s,t)} \overline{code(rd,t)}.A[\![\mathrm{P}]\!] \\
in)\ & A[\![\mathbf{in}(s).\mathrm{P}]\!] && = \textstyle\sum_{t:\mathrm{match}(s,t)} \overline{code(in,t)}.A[\![\mathrm{P}]\!] \\
var)\ & A[\![\mathrm{X}]\!] && = X \\
rec)\ & A[\![\mathbf{rec}\mathrm{X}.\mathrm{P}]\!] && = recX.A[\![P]\!] \\
ch)\ & A[\![P_1 [] P_2]\!] && = (\tau.A[\![P_1]\!] + \tau.A[\![P_2]\!]) \\
end)\ & A[\![\mathbf{end}]\!] && = Nil \\
ts)\ & A[\![M_1 \oplus M_2]\!] && = A[\![M_1]\!] \mid A[\![M_2]\!]
\end{aligned}$$

$$CCS[\![\mathrm{Linda}]\!] = A[\![Linda]\!] \setminus \{code(rd,t), code(in,t) \mid t \in Tuple\}$$

The intuitive reading of the translation defined by $CCS[\![\mathrm{Linda}]\!]$ is as follows: every Linda process and every tuple residing in the tuple space is modeled as a CCS agent. Such a modeling of tuples as processes is forced since pure CCS only has processes and synchronization actions, and not passive data objects. The CCS agent corresponding to a tuple t may be involved in a number of **rd** operations before it participates in an **in** operation and becomes the Nil agent, i.e., it is removed from the tuple space.

Processes and tuples are created by **eval** and **out** operations, respectively. The translation models this by the parallel composition of (the translation of) the created process or tuple and the continuation of the acting process. The parallel composition is prefixed by the τ action which models the creation of the process or tuple: it does not become part of tuple space until after the τ transition. Similarly, the nondeterminism in matching is reflected by the translation of the **rd** and **in** operations: they are represented by a nondeterministic composition of a set of CCS agents—one for each possible tuple that can be matched by the operation.

Example: Evaluation under *CCS*

Again, let P = **eval**(Q).**rd**(a).**out**(b).**end** and Q = **out**(a).**in**(b).**end**. For simplicity, in this example we only model matching by the case of identical tuples; thus, we let $F = \{code(rd,a), code(in,a), code(rd,b), code(rd,b)\}$ be the actions which are restricted away. The execution of a Linda program under *CCS*

may then be illustrated by the execution of P in an initially empty tuple space:

$$
\begin{aligned}
CCS[\![\mathrm{P}]\!] &= \tau.(A[\![\mathrm{Q}]\!] \mid A[\![\mathbf{rd}(a).\mathbf{out}(b).\mathbf{end}]\!] \setminus F && (eval)\\
&\xrightarrow{\tau} \tau.(A[\![a]\!] \mid A[\![\mathbf{in}(b).\mathbf{end}]\!]) \mid A[\![\mathbf{rd}(a).\mathbf{out}(b).\mathbf{end}]\!] \setminus F && (out)\\
&\xrightarrow{\tau} (code(rd,a).A[\![a]\!] + code(in,a).Nil) \mid A[\![\mathbf{in}(b).\mathbf{end}]\!] \mid && (rd)\\
&\quad \overline{code(rd,a)}.A[\![\mathbf{out}(b).\mathbf{end}]\!] \setminus F\\
&\xrightarrow{\tau} A[\![a]\!] \mid A[\![\mathbf{in}(b).\mathbf{end}]\!] \mid \tau.(A[\![b]\!] \mid Nil) \setminus F && (out)\\
&\xrightarrow{\tau} A[\![a]\!] \mid \overline{code(in,b)}.Nil \mid && (in)\\
&\quad (code(rd,b).A[\![b]\!] + code(in,b).Nil) \mid Nil \setminus F\\
&\xrightarrow{\tau} A[\![a]\!] \mid Nil \mid Nil \mid Nil \setminus F && (\mathrm{CCS})\\
&= A[\![a]\!] \setminus F
\end{aligned}
$$

A tuple space configuration is represented by the term itself; multi-set construction ($\oplus$) is reflected in the parallel composition ($\mid$) of CCS. A communication takes place when one agent consumes a tuple from the tuple space, in which case the CCS translation of a Linda process synchronizes with the translation of a matching tuple.

Correctness of the translation For the implementation of Linda in (pure) CCS we may of course want to establish the correctness of the translation. The correctness relation between the Linda specification and its translation is that of strong bisimulation, i.e., a process must evolve under *ITS* and *CCS* by similar actions through similar states. Intuitively, the idea is that two agents are in a bisimulation relationship if and only if (*iff*) they can do the same transitions and the states they reach after an action are still in the bisimulation relation (see e.g. [24]).

Definition: Strong bisimulation

Given two transition systems, T_1 and T_2, and a relation $\mathcal{R} \subseteq T_1 \times T_2$ between states of T_1 and T_2, $\mathcal{R}$ is a bisimulation if it has the following property:

$$
\begin{aligned}
&(P_1, P_2) \in \mathcal{R} \ \textit{iff} \ \forall \alpha : \\
&\quad P_1 \xrightarrow{\alpha} P_1' \Rightarrow \exists P_2' : P_2 \xrightarrow{\alpha} P_2' \wedge (P_1', P_2') \in R. \\
&\quad P_2 \xrightarrow{\alpha} P_2' \Rightarrow \exists P_1' : P_1 \xrightarrow{\alpha} P_1' \wedge (P_1', P_2') \in R.
\end{aligned} \tag{4}
$$

Given two states $s_1 \in T_1$ and $s_2 \in T_2$, s_1 is in strong bisimulation with s_2 *iff* there exists a strong bisimulation $\mathcal{R}$ such that $(s_1, s_2) \in \mathcal{R}$.

Given this definition it should be clear that the examples illustrating the execution of the Linda process **eval**(Q).**rd**(a).**out**(b).**end** under *ITS* is in strong bisimulation with its execution under *CCS*: the execution of the program evolves via corresponding transitions through similar states. By inspection it is seen that

the execution under *CCS* generally results in a number of *Nil* terms that is larger than the number of **end** terms in the corresponding execution under ITS, which, however, is unimportant as *Nil* processes cannot act.

Theorem: For every P $\in$ *Process* there exists a strong bisimulation between its execution under *ITS* and *CCS*

> **Outline of proof**: Transitions under *ITS* are not labeled, whereas all possible transitions under *CCS* are τ actions. Thus, we define the function relating transitions in *ITS* with transitions in *CCS* as $r(\rightarrow) = \stackrel{\tau}{\rightarrow}$. Given this, let $\mathcal{R}$ be defined as follows:
>
> $$\bigcup_{M}\{(M, A[\![M]\!])\} \in \mathcal{R}, \quad \text{and} \quad \forall (M, A) \in \mathcal{R} : (M, A|Nil),\ (M \oplus \mathbf{end}, A) \in \mathcal{R} \tag{5}$$
>
> The proof now follows from structural induction in the syntactic representation of P. □

The transition systems obtained by applying the operational rules are clearly related, but it is still an open issue which is a "good" notion of observer for such languages. In addition, the question of how to compare languages based on the occurrence of actions (as CCS and CSP, where all equivalences are defined in terms of sets of traces, trees of actions, etc.) with a language based on multi-set rewritings is still open. – Bisimulation may not be a good idea: it is mainly based on observable actions, and there are none in *ITS* and *CCS*. See the last section for a discussion on these topics.

CCS is a well studied language, and the result that a Linda process and its translation are bisimilar tells us that the transition systems defined are very similar. The translation that we developed can help in proving properties of the underlying transition system of the language, simply by looking at the properties of the CCS process.

5.2 Value–passing CCS implementation

In the translation of Linda into *CCS*, everything—notably matching of tuples—is modeled via processes and their synchronization. This is because pure CCS is meant for modeling synchronization of processes, but not for how they cooperate on some computation. Since Linda matching is highly data dependent, it is interesting to consider its translation into value–passing CCS [25]. Value–passing CCS is an extension of CCS where actions may be parameterized. The convention is that $\alpha(v)$ is an output operation and that $\overline{\alpha(x)}$ is the corresponding input operation which binds the value of v to the variable x. This translation will be much closer to an (abstract) implementation for a message based architecture; especially it will identify where (i.e. by which specific process) the *match* predicate is computed.

Definition: $CCS_v[\![\text{Linda}]\!]$ – *Translation of Linda into value–passing CCS*

$$
\begin{aligned}
\textit{tuple})\ & A_v[\![t]\!] && = recX.(\overline{rd(t)}.X \ +\ \overline{in(t)}.Nil) \\
\textit{eval})\ & A_v[\![\mathbf{eval}(\mathrm{P}).\mathrm{P}']\!] && = \tau.(A_v[\![\mathrm{P}]\!] \mid A_v[\![\mathrm{P}']\!]) \\
\textit{out})\ & A_v[\![\mathbf{out}(\mathrm{t}).\mathrm{P}]\!] && = \tau.(A_v[\![\mathrm{t}]\!] \mid |A_v[\![\mathrm{P}]\!]) \\
\textit{rd})\ & A_v[\![\mathbf{rd}(\mathrm{s}).\mathrm{P}]\!] && = rd(t).\mathbf{if}\ match(s,t) \\
& && \qquad \mathbf{then}\ A_v[\![\mathrm{P}]\!]\ \mathbf{else}\ A_v[\![\mathbf{rd}(\mathrm{s}).\mathrm{P}]\!] \\
\textit{in})\ & A_v[\![\mathbf{in}(\mathrm{s}).\mathrm{P}]\!] && = in(t).\mathbf{if}\ match(s,t) \\
& && \qquad \mathbf{then}\ A_v[\![\mathrm{P}]\!]\ \mathbf{else}\ (A_v[\![\mathbf{in}(\mathrm{s}).\mathrm{P}]\!] \mid A_v[\![t]\!]) \\
\textit{var})\ & A_v[\![\mathrm{X}]\!] && = X \\
\textit{rec})\ & A_v[\![\mathbf{rec}\mathrm{X}.\mathrm{P}]\!] && = recX.A_v[\![P]\!] \\
\textit{ch})\ & A_v[\![P_1[]P_2]\!] && = (\tau.A_v[\![P_1]\!] \ +\ \tau.A_v[\![P_2]\!]) \\
\textit{end})\ & A_v[\![\mathbf{end}]\!] && = Nil \\
\textit{ts})\ & A_v[\![M_1 \oplus M_2]\!] && = A_v[\![M_1]\!] \mid A_v[\![M_2]\!] \\
& CCS_v[\![\mathrm{Linda}]\!] = A_v[\![Linda]\!] \setminus \{rd(t), in(t) \mid t \in Active\}
\end{aligned}
$$

Again, to preserve the uncoupling of Linda processes we have chosen to model each individual tuple as a CCS agent (no unnecessary centralization through server processes). Such a tuple agent simply returns its tuple and either becomes the agent itself or the Nil process when synchronized with respectively an **rd** or **in** operation. It is the responsibility of the agent issuing the **rd** or **in** operation to test if the returned tuple matches: if it does not, it must reissue the operations (and recreate an agent implementing the tuple in case of an **in** operation). The translation of the other terms are identical to the pure CCS translation.

This translation of the Linda calculus into value–passing CCS highly resembles a Remote Procedure Calls implementation of tuple space: the set of tuple agents are simply a number of tuple servers. The exact number of servers will be determined by the actual number of nodes of the host systems. That is, a set of "logical" tuple agents will be implemented by a single server.

An interesting property of this implementation—a property which comes from our desire to avoid unbound nondeterminism—is that the transition system does not guarantee processes to make progress even if a matching tuple exists. This is because we have not imposed any ordering on tuples.

Another reason for the translation not to be a realistic implementation is the communication overhead: tuple agents and the translation of a Linda process should only synchronize if there is a high probability of a successful match. We shall not here go into the details of how to implement this, but it should be clear that the CCS agents should be organized in some well–defined network and that we should use the CCS restriction heavily to model this point–to–point

communication. The interested reader is referred to [7, 8] for a discussion of how to implement tuple space communication efficiently.

The possibility of not making progress in CCS_v comes from the associativity implied in unrestricted synchronization; in order to avoid it, communication must necessarily be point–to–point. However, the dynamic process creation of Linda demands a higher–order calculus, like CHOCS [33] or π-calculus [26], to model such a realistic Linda implementation.

6 Abstract Implementation by Petri Nets

When defining the semantics of a parallel language in the interleaving approach, concurrency is just a shorthand for nondeterminism: two events are concurrent if they can be executed in any order. Conversely, the so called *true concurrent approach* allows one to explicitly express the causality relationships between events, i.e., concurrency is the simultaneous evolution in local regions of activity. The components involved in an event can be singled out, whereas in the interleaving approach all components of a system participate in every event. Petri Nets are a natural tool to express distribution and causality which we here use to give a true concurrent semantics for our Linda calculus.

A Petri Net is defined as a 5-tuple, $Net =< \mathcal{P}, \mathcal{T}, \mathcal{I}, \mathcal{O}, \mu >$, where $\mathcal{P}$ is a set of *places*, $\mathcal{T}$ a set of *transitions*, and $\mathcal{I}$ and $\mathcal{O}$ is respectively an *input* and an *output* function; the Petri Nets defined here are also called Place/Transition Nets. The input function is a mapping from transitions to a multi-set of places, $\mathcal{I} : \mathcal{T} \rightarrow \{\!|\ p : \#[p] \mid p \in \mathcal{P}\ |\!\}$, and similarly the output function is a mapping from places to a multi-set of transitions $\mathcal{O} : \mathcal{P} \rightarrow \{\!|\ t : \#[t] \mid t \in \mathcal{T}\ |\!\}$. Finally, μ is the marking function for the net: it associates a number of tokens with each place, $\mu : \mathcal{P} \rightarrow Nat$ (we shall also represent μ as a multi-set).

The dynamics of a Petri Net is defined by the firing of transitions, i.e., by changes in the marking of the net. A transition, t_j, is enabled if all its input places have at least a number of tokens corresponding to its input cardinality, $\forall i : \mu(p_i) \geq \#[p_i, \mathcal{I}(t_j)]$. A transition may fire whenever it is enabled, and the firing yields a new marking, μ', for the Petri Net: $\forall p \in \mathcal{P} : \mu'(p) = \mu(p) - \#[p, \mathcal{I}(t_j)] + \#[p, \mathcal{O}(t_j)]$. A Petri Net has a natural graphical representation as an oriented, bipartite graph where places are represented by circles and transitions by bars; the input function is represented by edges from places to transitions, and the output function is represented by edges from transitions to places.

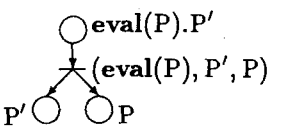
In our definition of a Petri Net semantics for the Linda Calculus places correspond to tuples and Linda processes, while transitions are the representation of Linda operations. Thus, we let the set of places be defined as the syntactic representation of tuples and processes, i.e., $\mathcal{P} = \mathit{Tuple} \cup \mathit{Process}$. Similarly, we define the set of transitions as $\mathcal{T} = \mathit{Op} \times \mathit{Process} \times (\mathit{Tuple} \cup \mathit{Process})$. Here the first element is the issued operation, the second the process as defined after the operation, and the third element the involved tuple or created process.

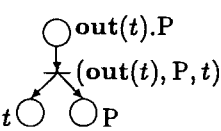
Besides the conventions for the structured naming of places and transitions, we also need a notation for the composition of Petri Nets. Let $Net =<$

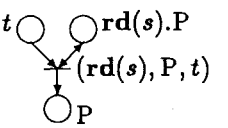
$P, T, I, O, \mu >$ and $Net' =< P', T', I', O', \mu' >$, then we define their composition as $Net \cup Net' =< P \cup P', T \cup T', I \cup I', O \cup O', \mu \uplus \mu' >$. Furthermore, we need a function $mark : M \rightarrow \mathcal{P}^{\infty}$ which maps a Linda term to an initial marking. The marking for a simple term (a tuple or process) is a place holding a single token, and the marking of a compound term is the union of the marking of its subterms: $mark(p) = \{\!|\ p\ |\!\}$, $mark(t) = \{\!|\ t\ |\!\}$, and $mark(M_1 \oplus M_2) = mark(M_1) \uplus mark(M_2)$.

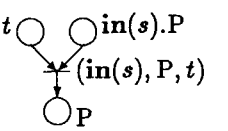
Definition: *Net*[Linda] – *A Petri Net Semantics for Linda*

tuple) $G[\![t]\!] = \bigcirc t$

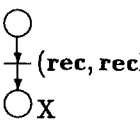
eval) $G[\![\mathbf{eval}(\mathrm{P}).\mathrm{P}']\!] = G[\![\mathrm{P}]\!] \cup G[\![\mathrm{P}']\!] \cup$ 

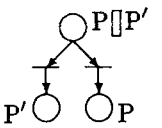
out) $G[\![\mathbf{out}(t).\mathrm{P}]\!] = G[\![\mathrm{P}]\!] \cup$ 

rd) $G[\![\mathbf{rd}(s).\mathrm{P}]\!] = G[\![\mathrm{P}]\!] \bigcup_{(s,t)\in match}$ 

in) $G[\![\mathbf{in}(s).\mathrm{P}]\!] = G[\![\mathrm{P}]\!] \bigcup_{(s,t)\in match}$ 

var) $G[\![\mathrm{X}]\!] = \bigcirc \mathrm{X}$

rec) $G[\![\mathbf{rec}\mathrm{X}.\mathrm{P}]\!] = G[\![\mathrm{P}]\!] \cup$ 

ch) $G[\![\mathrm{P}[]\mathrm{P}']\!] = G[\![\mathrm{P}]\!] \cup G[\![\mathrm{P}']\!] \cup$ 

end) $G[\![\mathbf{end}]\!] = \bigcirc \mathbf{end}$

ts) $G[\![M_1 \oplus M_2]\!] = G[\![M_1]\!] \cup G[\![M_2]\!]$

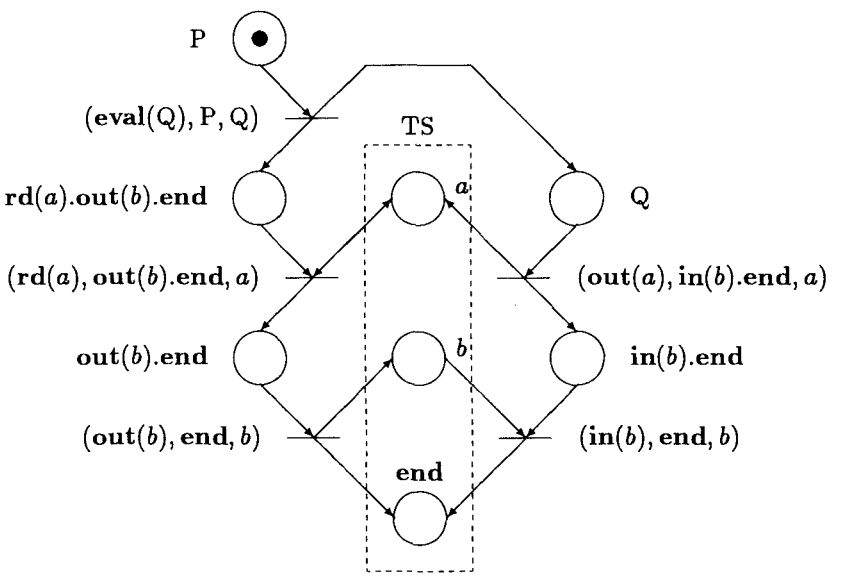
$Net[\![\mathrm{Linda}]\!] =< P, T, I, O, mark(Linda) >$, where $< P, T, I, O, \emptyset >= G[\![Linda]\!]$

In the definition of *Net*[[Linda]] we have used the graphical representation of a Petri Net: G defines the structure of the net and *mark* its initial marking. Each syntactic structure of a Linda process is represented by a corresponding Petri Net graph which should be composed with the graph for each of the substructures (a bidirectional arrow between a place and a transition indicates that the place is both input and output for the transition).

Thus, e.g. the definition of the **eval** operation is read as $G[\![\mathbf{eval}(\mathrm{P}).\mathrm{P}']\!] = G[\![\mathrm{P}]\!] \cup G[\![\mathrm{P}']\!] \cup < \{pl\}, \{tr\}, \{tr, \{\!|\ \mathrm{P}, \mathrm{P}'\ |\!\}\}, \{pl, \{\!|\ tr\ |\!\}\}, \emptyset >$, where $pl = \mathbf{eval}(\mathrm{P}).\mathrm{P}'$ and $tr = (\mathbf{eval}(\mathrm{P}), \mathrm{P}', \mathrm{P})$. The Petri Net semantics assumes that free variables used in recursion (X'es) are unique, i.e., that the same variable does not occur in two different recursions.

Example: Evaluation under *Net*

To illustrate the Petri Net semantics, the process $\mathrm{P} = \mathbf{eval}(\mathrm{Q}).\mathbf{rd}(\mathrm{a}).\mathbf{out}(\mathrm{b}).\mathbf{end}$ (where $\mathrm{Q} = \mathbf{out}(\mathrm{a}).\mathbf{in}(\mathrm{b}).\mathbf{end}$) is again instantiated in the empty tuple space. The Petri Net graph and its initial marking is shown below; for simplicity we have only included the "tuple–places" which are involved in the evaluation.



Given the firing-rules for transitions, it is easily seen that the Petri Net may evolve into a net with a marking where place **end** holds two tokens and place a a single token.

The Petri Nets defined by *Net*[[Linda]] have an interesting structure: every transition has at most two input and at most two output arcs. Furthermore, a "process–place" has one input arc for each process in the system which, after a Linda operation, may become the process denoted by the place. Thus, both a Linda process and its computation are identified by a sequence of places

and transitions: transitions corresponding to the sequential computation of the agent only have one output and one input arc, while additional arcs denote the interaction with tuple space. Given this intuition, the interpretation of the net resembles the operational semantics.

Theorem: Every process has an equivalent execution under respectively *ITS* and *Net*

> **Outline of proof**: In order to prove the theorem we view the firing of a transition as an atomic and instantaneous action. Thus we define a transition relation for a Petri Net as the firing of one of the transitions in the net: $n =< P, T, I, O, \mu > \rightarrow < P, T, I, O, \mu' >$ *iff* a transition in n is enabled and μ' is the marking after the firing of this transition. The theorem now follows by proving that *Net*[P] is in bisimulation equivalence with *ITS*[P] for all P's.
> Since both transition systems are unlabeled this is just a matter of proving that *Net* is closed under transitions. The proof for this is similar to the proof for the correctness of the pure CCS translation and follows from structural induction in the syntactic representation of P. □

The above illustrates a methodology for extending the SOS style to the true concurrent case first proposed in [13]: identify the sequential components and define a SOS style transition relation for these. The structure of states then has a parallel operation (here $\oplus$) which gives the structure of a Petri Net. It is easily seen that the operation $\oplus$ forms a free monoid where $\emptyset$ is the identity; thus, the structure is isomorphic to a Place/Transition Net [22, 30]. Previous works, e.g. [13], have used Petri Nets as the domain for giving true concurrent semantics to other languages.

The Petri Net defined by *Net*[Linda] provides a true concurrent semantics for our Linda calculus, where the computation of the *match* predicate is inherent in the structure of these nets. That is, such a Petri Net represents an optimized implementation of tuple space where the run-time matching is replaced by a nondeterministic choice between tuple residing in the tuple space (i.e. enabled "tuple–places"). Given the structure of every process which may be created as part of a program execution, the tuple space component may furthermore be reduced to the set of "tuple–places" which may be enabled by these processes. In this, the Petri Net semantics corresponds to a distributed implementation on a network.

Similarly to *Net*[Linda] we could also define a Petri Net semantics for the full Linda concept, though the value semantics would make Place/Transition Nets less suited for such a task. Instead, higher-order nets like Colored Petri Nets [18] should be used: tokens would be typed values and to each transition we would associate a predicate on the "occurrence color" of its input tokens defining when the transition is enabled. In this semantics, the tuple space would be a single place and the occurrence color of its tokens would be passive tuples; similarly, the tokens of processes would be a field identification of the active tuple in which they reside.

7 A Parallel Abstract Machine for Linda

An interesting feature of the Linda concept is that data and programs are mingled: a Linda-term is a process, and tuples may contain processes. This introduces the idea of a global environment where tuples and processes live together, and agents react to the environment according to some simple rules; for instance, Brownian motion. This analogy has inspired Berry and Boudol to the definition of a concurrent computing model called the *Chemical Abstract Machine* [4]. This model seems extremely promising for coordination languages.

7.1 The Chemical Abstract Machine

The CHAM is a model for concurrent computation based on the idea of abstract machines inherited from the theory of sequential languages. This model has enough descriptive power to implement other process calculi or to behave as a generalization of the lambda-calculus, remaining simple and intuitive. A CHAM is specified by defining molecules, solutions, and rules for the reaction of such. Solutions are multi-sets of molecules, written $\{\!|\ m_1, m_2, \ldots, m_n\ |\!\}$, but can also be considered as molecules themselves, appearing as subsolutions of another solution.

For each CHAM, molecules are terms of a specific algebra. The reaction rules define transformations of molecules and are presented by means of rule schemata, as in the SOS style. The unary operator $\{\!|\ \_\ |\!\}$ is called the membrane operator; much of the power of the model is based on this operator and on the possibility of defining subsolutions and complex molecules.

7.2 A Chemical Abstract Machine for the Linda Calculus

To view Linda as a CHAM-based language we simply define a set of molecule transformation rules:

Definition: *CHAM*[Linda] – *CHAM for the Linda Calculus*

$$
\begin{aligned}
\textit{par})\quad & M_1 \oplus M_2 \leftrightarrow M_1,\ M_2 \\
\textit{eval})\quad & \mathbf{eval}(\mathrm{P}).\mathrm{P}' \rightarrow \mathrm{P},\ \mathrm{P}' \\
\textit{out})\quad & \mathbf{out}(t).\mathrm{P} \rightarrow \mathrm{P},\ t \\
\textit{rd})\quad & \mathbf{rd}(s).\mathrm{P},\ t \rightarrow \mathrm{P}[t/s],\ t \qquad \text{if } (s,t) \in \textit{match} \\
\textit{in})\quad & \mathbf{in}(s).\mathrm{P},\ t \rightarrow \mathrm{P}[t/s] \qquad \text{if } (s,t) \in \textit{match} \\
\textit{rec})\quad & \mathbf{rec}\mathrm{X}.\mathrm{P} \rightarrow \mathrm{P}[\mathbf{rec}X.\mathrm{P}/\mathrm{X}] \\
\textit{leftch})\quad & \mathrm{P}[]\mathrm{P}' \rightarrow \mathrm{P} \\
\textit{rightch})\quad & \mathrm{P}[]\mathrm{P}' \rightarrow \mathrm{P}' \\
\textit{end})\quad & \mathbf{end} \rightarrow
\end{aligned}
$$

These rules are more or less identical to the interleaving semantics (*ITS*): the only difference is the use of pure multi-set notation (and the CHAM laws) instead of the $\oplus$ operator. As was predicted by Berry and Boudol, the definition of *CHAM*[Linda] is simple and straightforward: Linda is based on multi-set rewriting exactly like the CHAM.

Example: Evaluation under *CHAM*[Linda]

As usual, let P be the Linda process **eval**(Q).**rd**(a).**out**(b).**end** where Q is the process **out**(a).**in**(b).**end**. The execution of *CHAM* with P instantiated in an otherwise empty tuple space thus may evolve into a tuple space holding the tuple *a* as follows:

$$
\begin{aligned}
CHAM(\mathrm{P}) &= \{\!|\ \mathbf{eval}(\mathrm{Q}).\mathbf{rd}(a).\mathbf{out}(b).\mathbf{end}\ |\!\} && (eval)\\
&\rightarrow \{\!|\ \mathbf{out}(a).\mathbf{in}(b).\mathbf{end},\ \mathbf{rd}(a).\mathbf{out}(b).\mathbf{end}\ |\!\} && (out)\\
&\rightarrow \{\!|\ a,\ \mathbf{in}(b).\mathbf{end},\ \mathbf{rd}(a).\mathbf{out}(b).\mathbf{end}\ |\!\} && (rd)\\
&\rightarrow \{\!|\ a,\ \mathbf{in}(b).\mathbf{end},\ \mathbf{out}(b).\mathbf{end}\ |\!\} && (out)\\
&\rightarrow \{\!|\ a,\ \mathbf{in}(b).\mathbf{end},\ b,\ \mathbf{end}\ |\!\} && (in)\\
&\rightarrow \{\!|\ a,\ \mathbf{end},\ \mathbf{end}\ |\!\} && (end)\\
&\rightarrow^2 \{\!|\ a\ |\!\}
\end{aligned}
$$

A tuple space configuration is represented by a solution (a multi-set) where each tuple and Linda process is a molecule. The synchronization and generative communication is then represented by the chemical reaction between a process and a tuple molecule. The *match* condition on the *rd* and *in* rules states that a pair of process and tuple molecules only will react when they have a certain structure.

7.3 Implementing Matching in the CHAM

The specification of the abstract machine *CHAM*[Linda], gives a parallel operational semantics to the Linda calculus. However, it does not provide any machinery for computing the *match* predicate. The reaction rules for the **rd** and **in** operations may be viewed as the transitive closure of a set of reaction rules specifying the computation of the *match* relation. This computation is defined by the following rules:

Definition: *CHAM*[Match] – *A Matching CHAM*

$$\textit{pair)} \qquad \{\!|\ op(i, s_i),\ (i, t_i) \mid 1 \leq i \leq \#s = \#t\ |\!\}.\mathrm{P} \leftrightarrow op(s).\mathrm{P},\ t$$

$$\textit{match)} \qquad op(i, a:\tau),\ (i, a:\tau) \leftrightarrow op(i, a:\tau/a:\tau),\ (i, a:\tau)$$

$$\textit{match)} \qquad op(i, \bot:\tau),\ (i, a:\tau) \leftrightarrow op(i, a:\tau/\bot:\tau),\ (i, a:\tau)$$

$$\textit{match)} \qquad op(i, a:\tau),\ (i, \bot:\tau) \leftrightarrow op(i, a:\tau/a:\tau),\ (i, \bot:\tau)$$

$$\textit{rd)}\ \{\!|\ \mathbf{rd}(i, r_i/s_i),\ (i, t_i) \mid 1 \leq i \leq \#r = \#s = \#t\ |\!\}.\mathrm{P} \rightarrow \mathrm{P}[r/s],\ t$$

$$\textit{in)}\ \{\!|\ \mathbf{in}(i, r_i/s_i),\ (i, t_i) \mid 1 \leq i \leq \#r = \#s = \#t\ |\!\}.\mathrm{P} \rightarrow \mathrm{P}[r/s]$$

Here, *op* is either the **rd** or **in** primitive, and the notation $m \leftrightarrow \{\!|\ m_i \mid 1 \leq i \leq \#m\ |\!\}$ express the "heating" of a compound molecule such that it becomes the solution of its sub-molecules. Thus, the *pair* rule expresses the fact that a process must get mutual exclusion to a tuple, i.e., the tuple and the operation must be isolated in a solution where they are "heated" to molecules corresponding to the individual fields. The *match* rules express the matching of their individual fields (formal–to–actual or actual–to–actual) and the construction of a substitution, $op(i, r_i/s_i)$. Finally the *rd* and *in* rules commit to the **rd** or **in** operation once a tuple is matched successfully.

Example: Matching under *CHAM*[Match]

To illustrate the matching of a tuple under *CHAM*[Match], let $a = (5 : \mathbf{int})$ be a one-field tuple, and let $\mathrm{P}' = \mathbf{out}(b).\mathbf{end}$ and $\mathrm{P}'' = in(b).\mathbf{end}$. Then, the execution of the **rd** operation from our usual example may be as follows:

$$\begin{aligned}
CHAM(\mathrm{P}) &\rightarrow^{\star} \{\!|\ (5 : \mathbf{int}),\ \mathbf{in}(b).\mathbf{end},\ \mathbf{rd}(5 : \mathbf{int}).\mathbf{out}(b).\mathbf{end}\ |\!\}\ (\textit{pair}) \\
&\leftrightarrow \{\!|\ \{\!|\ \mathbf{rd}(1, 5 : \mathbf{int}),\ (1, 5 : \mathbf{int})\ |\!\}.\mathrm{P}',\ \mathrm{P}''\ |\!\} \qquad (\textit{match}) \\
&\leftrightarrow \{\!|\ \{\!|\ \mathbf{rd}(1, 5 : \mathbf{int}/5 : \mathbf{int}),\ (1, 5 : \mathbf{int})\ |\!\}.\mathrm{P}',\ \mathrm{P}''\ |\!\}\ (rd) \\
&\rightarrow \{\!|\ (5 : \mathbf{int}),\ \mathbf{in}(b).\mathbf{end},\ \mathbf{out}(b).\mathbf{end}\ |\!\}
\end{aligned}$$

CHAM[Match] is based on the intuition that tuples are matched field by field in some undetermined order (possibly in parallel). Except for the rules for the **rd** and **in** operations (which commit to the match), the reaction rules are reversible. Reversible rules express the fact that a tuple has been chosen for matching, but it is not yet fully matched. Thus, the *match* predicate may still yield *false* in which case it must be possible to "backtrack" to the initial state and try another tuple. An operation matches a tuple *iff* every tuple field may react with the corresponding field of the operation up to a substitution.

Theorem: *CHAM*[Match] implements the match relation of Linda

Proof: We must prove that an **rd** or **in** operation may find a substitution *iff* a matching tuple exists in tuple space, i.e.,

$$op(s).\mathrm{P},\ t \leftrightarrow^{\star} \{\!|\ op(i, r_i/s_i),\ (i, t_i) \mid 1 \leq i \leq \#r = \#s = \#t\ |\!\}.\mathrm{P} \ \Leftrightarrow\ (s, t) \in \textit{match}$$

Assume $op(s).\mathrm{P},\ t \leftrightarrow^{\star} \{\!|\ op(i, t_i/s_i) \mid 1 \leq i \leq \#r = \#s = \#t\ |\!\}.\mathrm{P}$. Thus, for $1 \leq i \leq \#s = \#t$ there is only one pair of molecules for the ith field, and we may use one of the *match* rules, $op(i, s_i),\ (i, t_i) \leftrightarrow op(i, r_i/s_i),\ (i, t_i)$, for this pair of fields. Given this, each corresponding pair of fields of s and t match either actual–to–actual or formal–to–actual.
Assume $match(s, t)$. Thus, for $1 \leq i \leq \#s = \#t : (s_i = a : \tau \wedge t_i = a' : \tau)$ where either $a = a' \neq \bot$ or exactly one of a and a' is a formal ($\bot$). Given this, it easily follows that the solution constructed from $op(s).\mathrm{P}$ has a set of corresponding pair of $op(i, s_i)$ and (i, t_i) molecules, and furthermore that each pair may react to a substitution by use of the *match* rules. In consequence, there will be exactly $\#s = \#t$ pairs of molecules left in the solution, each pair of the form $op(i, r_i/s_i),\ (i, t_i)$. □

Once an operation is matched and a substitution found, the solution may finally be "cooled" again. By this, the solution becomes a molecule corresponding to the process after matching a tuple, and for the **rd** operation also a molecule corresponding to the matched tuple. The CHAM implementation may look a little tricky, whereas it is just an operational specification of Linda's *match* relation: it describes the matching processes in details, but it is still abstract enough not to commit towards any particular implementation.

7.4 A CHAM for Linda

The basic relation between *CHAM*[Linda] and *CHAM*[Match] is that a tuple is considered as a molecule at the tuple space level and as a solution of its fields at the matching level. This illustrates the hierarchical composition of CHAMs: when we "cool" a solution to a molecule we take a global view on some local region of activity, and similarly we take a distributed view when we "heat" a molecule to a solution. Given this intuition, and the definition of processes in the Linda concept given by Equation 1, we may define a CHAM for the full Linda concept as follows:

Definition: *CHAM*$_f$[Linda] – *CHAM for full Linda*

CHAM[Match] augmented with the following reaction rules:

$$\textit{eval)} \qquad \mathbf{eval}(t'').\mathrm{P} \rightarrow \mathrm{P},\ t''$$

$$\textit{out)} \qquad \mathbf{out}(t).\mathrm{P} \rightarrow \mathrm{P},\ t$$

$$\textit{tuple)} \qquad t \leftrightarrow \{\!|\ (i, \{\!|\ t_i\ |\!\}) \mid 1 \leq i \leq \#t\ |\!\}$$

$$\textit{emit)}\ \{\!|\ (i, p) \triangleleft t'\ |\!\},\ t \leftrightarrow \{\!|\ (i, \{\!|\ t \triangleleft p\ |\!\}) \triangleleft t'\ |\!\}$$

The CHAM for the full Linda concept simply extends the reaction rules for the **rd** and **in** operations defined by *CHAM*[Match] with the corresponding reactions for the **eval** and **out** operations. Similarly, the last two rules defines how processes residing as active fields of a tuple interact with tuple space: a tuple may be "heated" to a solution of fields where each active field is a solution with

the process itself as the only molecule. Here, p is a solution variable denoting an active process, and t' a solution variable denoting the remainder of the active tuple. That is, p is the syntactic process and the tuples "in its scope" e.g. {| P, (5, 7) |}, and t the solution with other fields of the tuple, e.g. {| (1, {| "field" | }), (2, {| 5 |}) |}.

The reader is invited to trace the evolution of our usual example but it warned that it involves a fairly large number of reactions to establish the evaluation {| (P) |}$\rightarrow^{\star}${| $(a,\ (\mathbf{end}_q),\ (\mathbf{end}_p)$ |}. Here, $\mathbf{end}_p$ and $\mathbf{end}_q$ should be typed values that the P and Q processes yield when they terminate.

The complexity of this evaluation comes from the nesting of solutions within solutions: the outermost solution corresponds to the tuple space, and the next level of solutions to the tuples. The third level of solutions is the active fields of a tuple; this is the level at which processes react.

The reversible *tuple*, *emit*, and *pair* rules are the implementation of the propagation of tuples between the tuple space and the process solutions. This propagation of tuples uses the airlock law to isolate a molecule from one level of solutions and enable it to react with a molecule of the surrounding solution. Here, the isolated molecule is a process performing an **rd** or **in** operation, or a tuple generated by an **eval** or **out** operation. The above CHAM only defines the coordination through tuple space as it does not provide any reaction rules for the sequential computation of Linda processes.

However, given a host language and its operational semantics, a CHAM for its Linda embedding is easily defined. Similarly, the *pair* rule and subsequent *match* rules show the fact that tuples are reserved to a process until the matching succeeds or fails. Here, failure is the unability to find a substitution for every field, and thus the process must go back. Note that this representation of matching may be seen as a way of handling backtracking within this operational model; this approach also works when matching is generalized to logical unification.

Even if presented as an abstract machine, the CHAMs given in this section is a low-level description of Linda. What is the insight given by this implementation? — The definition includes the description of the matching process within the formalism, which allows us to describe it at the same level (and in the same framework) as the rest of the model. This also enables an evaluation of the cost of matching: the match relation is here represented by operational rules. Similarly, the reaction rules for the Linda primitives provide a distributed semantics in a simple and straight forward way, i.e., identifies local regions of activity which may evolve independently of each other.

8 A More Abstract Semantics

The semantics of a language is not completely defined if the assumptions about what is observable and what is not are not made clear. Different assumptions lead to different kinds of semantics. For instance, if a special action is considered as internal and hence it cannot be observed from outside of the system, the so-called weak equivalences arise. If not only action occurrences, but also the

causal dependencies between them can be observed, the so-called *truly concurrent* or *partial order* semantics are obtained [13]. Which observation, and thus which semantics is more useful, depends on what properties of a language one is interested in.

Once these assumptions are made explicit, an *equivalence* on agents can be defined. The study of equivalences of programs is important both from a theoretical and from a practical point of view. From a theoretical perspective, the study of equivalences is directly connected with the study of models and denotational semantics. When a congruence has been established, a (term) model can be immediately constructed, mapping each agent to its equivalence class. The problem of the equivalence of programs happens in many situations: in the development of correct rules for program transformation, to check if a program is an implementation of a specification, when substituting a module of a complex system with another, more efficient one, etc.

Several equivalences have been proposed in the literature for concurrent systems. Among them the most widespread are those based on the notion of *bisimulation*, which has been introduced in Section 5. Bisimulation relations are very fine, since they take into account the branching structure of the processes, and the equivalences based on them are very discriminating. While these distinctions have an intuitive justification as long as synchronous communication is considered, for the case of asynchronous communications it has been argued that they are *too* fine [12]. In the case of shared tuple space communication the equivalence has to be even coarser, since it may be difficult even to recognize which process has produced a tuple.

Actually, the equivalence used in this context should be based on the following statement:

> *Two programs are equivalent iff they behave the same when they interact with another program or with a user.*

Thus, if a user (or a program) interacts with two programs by sharing a common tuple space (reading, writing and removing tuples from it) and he cannot see any difference between both programs, those programs are considered equivalent. So, now we have to define which kind of interaction a user or an external program may provide and what is the behaviour of a system. The formal definition of the operational semantics turns out to be fundamental for this task.

Definition: *Computations*
A computation from a process P in an operational semantics *op* is a (finite) sequence of tuple space terms, $M_1; M_2; \ldots M_n$ such that $M_i \rightarrow M_{i+1}$ is a transition of the operational semantics *op* for $i = 1 \ldots n-1$ and $M_1 = P$.
A computation c from P is said to be *maximal* iff there is no computation c'' from P of the form $c'' = c; c'$.

We now define a set of *tests* that can be performed on a Linda program, and we also define when a program passes or fails the test. Two programs will

be equivalent iff they pass the same set of tests. This equivalence has been proposed for process description languages by De Nicola and Hennessy with the name *testing equivalence* [14]. A nice introduction to testing equivalence can be found in [16].

Since the Linda calculus defined so far has been proposed with the aim of describing interactions with the tuple space and Linda programs are supposed to interact mainly with Linda programs, it is reasonable to state that Linda programs describe the set of possible tests. However, in order to characterize an equivalence based on *observable* properties of processes, some restrictions about the possible tests seem very natural:

1. a test takes a finite amount of time, and observers are finite (in [16] it is shown that the equivalence does not change if one considers also (some) infinite tests); and
2. an observer does not stop if he can proceed. This fact is formalized with the notion of *maximal computations*, defined above.

Definition: *Experimenter*
An experimenter E is a Linda agent with no recursion which can output (but not input) a distinguished tuple $\surd$.

The experimental setting that we propose here is the following. An experiment is described by a Linda agent which may output a distinguished tuple called $\surd$ which is interpreted as "the experiment is successful". The experimenter and the program to be tested are started in parallel in an empty tuple space. The program pass the test if the tuple $\surd$ is put in the tuple space. Otherwise, the program fails the test. Notice that we are dealing with potentially nondeterministic programs, and thus the result can be different in two different runs.

Definition: *Experiments*
Given a Linda agent P and an experimenter E, an experiment C for P, E in the operational semantics *op* is a maximal computation $M_1; M_2; \ldots M_n$ in *op* from **eval**(P).E.
An experiment $M_1; M_2; \ldots M_n$ is said to be successful if $\surd \in M_n$, and it is unsuccessful if $\surd \notin M_n$.
The result of applying an experimenter E to an agent P with respect to the operational semantics *op* is $result(\mathrm{E}, \mathrm{P}) \subseteq \{\top, \bot\}$ defined by:
$\top \in result(\mathrm{E}, \mathrm{P})$ if there is a successful experiment for P, E
$\bot \in result(\mathrm{E}, \mathrm{P})$ if there is an unsuccessful experiment for P, E

Now we formally define the induced equivalence.

Definition: *Equivalence*
Given two agents P, P', P $\sim$ P' in the operational semantics *op* iff for every experimenter E $result(\mathrm{E}, \mathrm{P}) = result(\mathrm{E}, \mathrm{P}')$ in *op*.

A detailed study of the properties of this equivalence is outside of the scope of this paper. An interesting point would be to study whether the testing approach for shared memory languages and the approach of [12] coincide, and the study of axiomatizations for the equivalence $\sim$. Now, some examples are in order:

For any agent P, tuples t, t',

$$\mathbf{out}(t).\mathbf{out}(t').\mathrm{P} \sim \mathbf{out}(t').\mathbf{out}(t).\mathrm{P}$$

but

$$\mathbf{out}(t).\mathbf{end} \not\sim \mathbf{out}(t).\mathbf{in}(t').\mathbf{end}$$

since the experimenter $\mathbf{out}(t').\mathbf{in}(t').\mathbf{out}\ (\surd).\mathbf{end}$ can distinguish them.

The operational semantics give descriptions of the different abstract machines for the language. However, one should expect that from the point of view of the user of the language, these different implementations are equivalent. In other words, if two programs cannot be distinguished in one implementation, they should not be distinguished when implemented in another abstract machine.

The following theorem shows that, as far as the transition systems, Petri Net and CHAM implementations are considered, the operational semantics proposed are equivalent for a user of the language.

Theorem: Let *op*, *op'* be one of *ITS*, *MTS*, *Net* or *CHAM* transition relations, and let P, P' be two Linda agents. Then, P $\sim$ P' with respect to *op* iff P $\sim$ P' with respect to *op'*.

Proof: The proof proceeds by induction on the length of the computations. Given a computation $M_1; M_2; \ldots M_n$ from a process in one of the transition relations, we build a computation $M'_1; M'_2; \ldots M'_m$ for each other abstract machine with the property that $M'_1; M'_2; \ldots M'_m$ is a computation from the same process and $M_n = M'_m$.
The construction of *Net* computations from *MTS* computations is immediate. To construct *CHAM* computations from *Net* it is necessary to use the theorem that shows that the matching relation is correctly implemented in the *CHAM* rules; and the same theorem is used to construct *ITS* computations from the *CHAM*. In this case, the length of a *ITS* computation may have a different length: parallel moves in the *CHAM* have to be decomposed in any of its interleavings. Finally, *ITS* computations are also *MTS* computations.
Hence, for each successful (resp. unsuccessful) experiment in one operational semantics there exists a successful (resp. usuccessful) one on the other operational semantics, and this implies the desired result. □

Two consequences of this theorem are

1. The notion of equivalence is independent of the underlying transition relation, and thus the notation P $\sim$ P' can be used instead of P $\sim$ P' with respect to *op*.

2. From an abstract point of view, all the implementations proposed in the paper coincide.

Finally, let us mention that the notion of equivalence is very helpful for the verification of properties of programs. For message-passing languages, efficient algorithms have been provided to automatically check whether two (finite) processes are (testing) equivalent [10], they can be adapted for this case. Moreover, this notion of equivalence provides the basis for the verification of logical properties of Linda programs.

Acknowledgements: This paper was partially supported by ESPRIT BRA project 9102 - COORDINATION.

References

1. Jean-Marc Andreoli and Remo Pareschi. Linear Objects: Logical Processes with Built-in Inheritance. *New Generation Computing*, 9(3-4):445–473, 1991.
2. H. Bal, J. Steiner, and A. Tanenbaum. Programming languages for distributed computing systems. *ACM Computer Surveys*, 21(3):261–322, 1989.
3. JP. Banatre and D. LeMetayer. Programming by Multiset Transformation. *Communications of the ACM*, 36(1):98–111, January 1993.
4. G. Berry and G. Boudol. The Chemical Abstract Machine. *Theoretical Computer Science*, 96:217–248, 1992.
5. A. Brogi and P. Ciancarini. The concurrent language Shared Prolog. *ACM Transactions on Programming Languages and Systems*, 13(1):99–123, 1991.
6. P. Butcher. A behavioral semantics for Linda-2. *IEE Software Engineering Journal*, 6(4):196–204, July 1991.
7. N. Carriero. *Implementing Tuple Space Machines.* PhD thesis, Dept. of Computer Science, Yale University, New Haven, Connecticut, 1987.
8. N. Carriero and D. Gelernter. New optimization strategies strategies for the Linda Precompiler. In Greg Wilson, editor, *Linda-like systems and their implementation*, pages 74–83. Edimbourgh Computing Center, 1991.
9. N. Carriero and D. Gelernter. Coordination Languages and Their Significance. *Communications of the ACM*, 35(2):97–107, February 1992.
10. R. Cleaveland and M. Hennessy. Testing Equivalence as a Bisimulation Equivalence. In *Proc. Workshop on Automatic Verification for finite-state Systems*, volume 407 of *Lecture Notes in Computer Science*, pages 11–23, 1989.
11. R. Cridlig and E. Goubault. Semantics and Analysis of Linda-based Languages. In P. Cousot and M. Falaschi and G. File and A. Rauzy, editors, *Proc. Int. Workshop on Static Analysis (WSA 93)*, volume 724 of *Lecture Notes in Computer Science*, pages 72-86, 1993.
12. F. deBoer, J. Klop, and C. Palamidessi. Asynchronous Communication in Process Algebra. In *Proc. 7th IEEE Symp. on Logic In Computer Science - LICS.* IEEE Computer Society Press, 1992.
13. P. Degano, R. DeNicola, and U. Montanari. A distributed operational semantics for CCS based on Condition/Event systems. *Acta Informatica*, pages 59–91, 1988.
14. R. DeNicola and M. Hennessy. Testing equivalence for processes. *Theoretical Computer Science*, 34:83–133, 1983.
15. D. Gelernter. Generative communication in Linda. *ACM Transactions on Programming Languages and Systems*, 7(1):80–112, 1985.

16. M. Hennessy. *Algebraic Theory of Processes.* MIT Press, Cambridge, Cambridge, MA, 1988.
17. S. Jagannathan. Semantics and Analysis of First-Class Tuple-Spaces. Technical Report DCS/RR-783, Dept. of Computer Science, Yale University, New Haven, CT, April 1990.
18. K. Jensen. Coloured Petri Nets. In *Petri Nets: Central Models and Their Properties*, volume 254 of *Lecture Notes in Computer Science*, 1987.
19. K. Jensen. Decoupling of Computation and Coordination in Linda. In D Heidrich and J Grossetie, editors, *Computing with T.Node Parallel Architectures*, pages 43–62, 1991.
20. K. Jensen. *Towards a Multiple Tuple Space Model – On the Use of SOS in Design and Implementation.* PhD thesis, Dept. of CS, Aalborg University, Denmark, 1993 (forth-coming).
21. J. Leichter. *Shared Tuple Memories, buses and LANs - Linda implementations across the spectrum of connectivity.* PhD thesis, Dept. of Computer Science, Yale University, New Haven, Connecticut, July 1989.
22. J. Meseguer and U. Montanari. Petri Nets are Monoids. *Information and Computation*, 88:105–155, 1990.
23. R. Milner. *A Calculus of Communicating Systems*, volume 92 of *Lecture Notes in Computer Science.* Springer-Verlag, Berlin, 1980.
24. R. Milner. Calculi for synchrony and asynchrony. *Theoretical Computer Science*, 25:267–310, 1983.
25. R. Milner. *Communication and Concurrency.* Prentice Hall, 1989.
26. R. Milner. Functions as processes. *Mathematical Structures in Computer Science*, 2(2):119–141, 1992.
27. J. Narem. An Informal Operational Semantics of C-Linda version 2.3.5. Technical Report TR839, Dept. of Computer Sc., Yale Univ., New Haven, Connecticut, 1989.
28. J. Peterson. *Petri Net Theory and the Modeling of Systems.* Prentice Hall, 1981.
29. G. Plotkin. A structural approach to operational semantics. Technical Report DAIMI FN-19, Department of Computer Science, Aarhus University, Denmark, 1981.
30. W. Reisig. *Petri Nets. An Introduction.* EATCS Monographs on Theoretical Computer Science. Springer-Verlag, Berlin, 1985.
31. M. Rem. Associons: A program notation with tuples instead of variables. *ACM Transactions on Programming Languages and Systems*, 3(3):251–262, July 1981.
32. GC. Roman and HC.Cunningham. Mixed Programming Metaphors in a Shared Dataspace Model of Concurrency. *IEEE Transactions on Software Engineering*, 16(12):1361–1373, December 1990.
33. B. Thomsen. A calculus of higher order communicating systems. In *16th ACM Conf. on Principles of Programming Languages*, pages 143–154, Austin, Tx, January 1989.