Electronic Notes in Theoretical Computer Science 66 No. 4 (2002)
URL: http://www.elsevier.nl/locate/entcs/volume66.html 15 pages

Compatibility of Linda-based Component Interfaces

Antonio Brogi [1]

Dipartimento di Informatica, Università di Pisa, Italy

Ernesto Pimentel [2,4]

Dpto. de Lenguajes y Ciencias de la Computación,University of Málaga, Spain

Ana M. Roldan [3,4]

Dpto. de Ingeniería Electrónica, Sistemas Informáticos y Automática, University of Huelva, Spain

Abstract

Linda is a coordination language, originally presented as a set of inter-agent communication primitives which can virtually be added to any programming language. In this paper, we analyse the use of Linda to specify the interactive behaviour of software components. We first introduce a process algebra for Linda and we define a notion of process compatibility that ensures the safe composition of components. In particular, we prove that compatibility implies successful computation. We also argue that Linda features some advantages with respect to similar proposals in the context of dynamic compatibility checking. In this perspective, we propose an alternative definition of compatibility that takes into account the state of a global store, which gives some relevant information about the current execution of the system.

1 Introduction

Component-Based Software Engineering (CBSE) is an emerging discipline in the field of Software Engineering. In spite of its recent birth, a lot of activities are being devoted to CBSE both in the academic and in the industrial

[1] Email: brogi@di.unipi.it
[2] Email: pimentel@lcc.uma.es
[3] Email: amroldan@diesia.uhu.es
[4] The work of Ana M. Roldán and E. Pimentel has been partially supported by the Spanish project TIC2001-2705-C03-02

world. The reason of this growing interest is the need of systematically developing open system and "plug-and-play" reusable applications, which has led to the concept of "commercial off-the-self" (COTS) components. The first component-oriented platforms were CORBA [16] and DCE [14], developed by OSF (Open Software Foundation) and OMG (Object Management Group). Several other platforms have been developed after them, like COM/DCOM [10], CCM [21], EJB [17], and the recent .NET [15].

Available component-oriented platforms address software interoperability by using Interface Description Languages (IDLs). Traditional IDLs are employed to describe the services that a component offers, rather than the services the component needs (from other components) or the relative order in which the component methods are to be invoked. IDL interfaces highlight signature mismatches between components in the perspective of adapting or wrapping them to overcome such differences.

However, even if all signature problems may be overcome, there is no guarantee that the components will suitably interoperate. Indeed, mismatches may also occur at the protocol level, because of the ordering of exchanged messages and of blocking conditions, that is, because of differences in the component behaviours. To overcome such a limitation, several proposals have been put forward in order to enhance component interfaces [12]. Many of them are based on process algebras, and extend interfaces with a description of their concurrent behaviour [1,2,7,8,13,20], such as behavioural types or role-based representations.

The objective of this work is to explore the usability of the coordination language Linda [9] for specifying the interaction behaviour of software components. Linda was originally presented as a set of inter-process communication primitives which allow processes to add, read, and delete data in a shared tuple space (store). Linda's communication model features interesting properties, such as space and time uncoupling [9], as well as a great expressive power to specify concurrent and distributed systems [3,4].

The contributions of this paper can be summarised as follows:

(i) We use Linda as the specification language to describe interface protocols. Syntactically, this corresponds to extending traditional IDL interfaces with a Linda description of component behaviours. The formal meaning of a Linda protocol is given by means of the process algebra presented in [5].

(ii) We define a notion of process *compatibility* that guarantees the safe composition of components. More precisely, we prove that the compatibility of two processes implies that their interaction will be successful.The importance of the notion of compatibility relates to the possibility of performing *a priori* verification of complex interacting systems.

(iii) We then define a notion of *store sensitive compatibility* to formalise the compatibility of two processes with respect to a given state of the

store.The state of the store is particularly significant in Linda as it is the only means by which (all) Linda processes communicate. The store hence provides relevant information on the results of the current execution of the system, and it allows to contextualise the compatibility of processes in the perspective of dynamic compatiblity checking.

The rest of the paper is organized as follows. Section 2 presents a process calculus for Linda. The use of Linda for specifying component protocols is also illustrated by means of a simple example. Section 3 is devoted to introduce the notion of process compatiblity and to prove that compatibility ensures safe compositions. An alternative, store sensitive definition of compatibility is then given, and the relation between the two notions is stated. Finally, some concluding remarks are discussed in Section 4.

2 Specifying component protocols in Linda

2.1 A Linda calculus

Linda [9] was the first coordination language [11], originally presented as a set of inter-agent communication primitives which can virtually be added to any programming language. Linda's communication primitives allow processes to add, delete and test for the presence/absence of tuples in a shared *tuple space*. The tuple space is a multiset of data (tuples), shared by concurrently running processes. Delete and test operations are blocking and follow an associative naming scheme that operates like *select* in relational databases.

In this paper, following [5], we shall consider a process algebra $\mathcal{L}$ containing the communication primitives of Linda. These primitives permit to add a tuple (*out*), to remove a tuple (*in*), and to test the presence/absence of a tuple (*rd*, *nrd*) in the shared dataspace. The language $\mathcal{L}$ includes also the standard prefix, choice and parallel composition operators in the style of CCS [18].

The syntax of $\mathcal{L}$ is formally defined as follows:

$$P ::= 0 \mid A.P \mid P + P \mid P \parallel P \mid recX.P$$
$$A ::= rd(t) \mid nrd(t) \mid in(t) \mid out(t)$$

where 0 denotes the empty process and t denotes a tuple.

Following [5], the operational semantics of $\mathcal{L}$ can be modeled by a labelled transition system defined by the rules of Table 1. Notice that the configurations of the transition system extend the syntax of processes by allowing parallel composition of tuples. Formally, the transition system of Table 1 refers to the extended language $\mathcal{L}'$ defined as:

$$P' ::= P \mid P' \mid\mid \langle t \rangle$$

Rule (1) states that the output operation consists of an internal move which creates the tuple $\langle t \rangle$. Rule (2) shows that a tuple $\langle t \rangle$ is ready to offer itself to the environment by performing an action labelled $\overline{t}$. Rules (3), (4) and (5) describe the behaviour of the prefixes $in(t)$, $rd(t)$ and $nrd(t)$ whose

(1)	$out(t).P \xrightarrow{\tau} \langle t \rangle \parallel P$	(6)	$\dfrac{P \xrightarrow{\alpha} P'}{P + Q \xrightarrow{\alpha} P' + Q}$
(2)	$\langle t \rangle \xrightarrow{\overline{t}} 0$	(7)	$\dfrac{P \xrightarrow{t} P' \; Q \xrightarrow{\overline{t}} Q'}{P \parallel Q \xrightarrow{\tau} P' \parallel Q'}$
(3)	$in(t).P \xrightarrow{t} P$	(8)	$\dfrac{P \xrightarrow{\underline{t}} P' \; Q \xrightarrow{\overline{t}} Q'}{P \parallel Q \xrightarrow{\tau} P' \parallel Q}$
(4)	$rd(t).P \xrightarrow{\underline{t}} P$	(9)	$\dfrac{P \xrightarrow{\alpha} P' \; \alpha \neq \neg t}{P \parallel Q \xrightarrow{\alpha} P' \parallel Q}$
(5)	$nrd(t).P \xrightarrow{\neg t} P$	(10)	$\dfrac{P \xrightarrow{\neg t} P' \; Q \not\xrightarrow{\overline{t}}}{P \parallel Q \xrightarrow{\neg t} P' \parallel Q}$

Table 1
Transition system for $\mathcal{L}$.

labels are t, $\underline{t}$ and $\neg t$, respectively. Rule (6) is the standard rule for choice composition. Rule (7) is the standard rule for the synchronization between the complementary actions t and $\overline{t}$: It models the effective execution of an $in(t)$ operation. Rule (8) defines the synchronization between two processes performing a transition labelled $\underline{t}$ and $\overline{t}$, respectively. Notice that the process performing $\overline{t}$ is left unchanged, since the read operation $rd(t)$ does not modify the dataspace. The usual rule (9) for the parallel operator can be applied only to labels different from $\neg t$. Indeed a process P can execute a $nrd(t)$ action in parallel with Q only if Q is not able to offer the tuple $\langle t \rangle$, as stated by rule (10). Notice that, following [5], there are no rules for recursion since its semantics is defined by structural axiom $recX.P \equiv P[recX.P/X]$ which applies an unfolding step to a recursively defined process.

The rules of Table 1 are used to define the set of derivations for a Linda system. Following [5], both reductions labelled τ and reductions labelled $\neg t$ are considered. Formally, this corresponds to introducing the following derivation relation:

$$P \longmapsto P' \quad \text{iff} \quad (P \xrightarrow{\tau} P' \text{ or } P \xrightarrow{\neg t} P').$$

Notice that the above operational characterization of $\mathcal{L}$ employs the so-called *ordered* semantics of the output operation [6]. Namely, when a sequence of outputs is executed, the tuples are rendered in the same order as they are emitted. It is also worth noting that also the store can be seen as a process which is the parallel composition of a number of tuples.

Let us finally introduce another derivation relation that will be used as a

shorthand in the rest of the paper:

$$P \stackrel{\alpha}{\Longrightarrow} P' \quad \text{iff} \quad (P \longmapsto^* \stackrel{\alpha}{\longrightarrow} P')$$

where $\alpha \in \{t, \underline{t}, \overline{t}\}$.

2.2 Component protocols in Linda

We now describe how the Linda language can be effectively used to specify the interactive behaviour of components. To exemplify the appropriateness of Linda for specifying component protocols, we will illustrate its application to the standard client/server interaction model.

The typical basic behaviour of a server can be described by the following protocol:

```
SERVER = in(c,tos,qry).out(c,ans).SERVER
```

The server repeatedly exhibits the same interactive behaviour: It first inputs a request and then outputs the answer it computed for the received request. The input operation has three parameters which denote the name (`c`) of the client who produced the request, the type of service required (some constant `tos`), and the actual request (`qry`). The server then returns its answer to the query by placing a tuple of the form `<c,ans>` in the shared dataspace.

The typical basic behaviour of a client is instead described by the following protocol:

```
CLIENT = out(#me,tos,qry).in(#me,ans).CLIENT
```

where `#me` is the identifier of the client process [9].

Notice that, in the above specification, the client request does not refer to the name of a specific server. Most importantly, a client does not need to be aware of which servers are currently available. Notably, the above specification allows several clients and servers to be dynamically and transparently plugged in an open system.

The above specification describes the basic behaviour of clients and servers. A more refined specification may include for instance the brokerage of the servers currently available for a given type of service. Indeed, the server protocol may be rewritten so that the first operation a server performs is to inform the system that she is a server featuring a certain type of service. This can be done by outputing a tuple that associates the process identifier with a certain type of service, as specified in the following protocol:

```
SERVER = out(tos,#me).CYCLE
CYCLE  = in(c,#me,qry).out(#me,c,ans).CYCLE
```

where `CYCLE` is a process name. Notice also that the `SERVER` protocol employs the tuple format `<sender,receiver,message>` for the message exchanged between clients and servers on the shared dataspace.

Server brokerage can be then easily included in the client protocol as follows:

```
CLIENT = (rd(tos,srv).out(#me,srv,qry).in(srv,#me,ans).CLIENT)
         +
         (nrd(tos,srv).EXCEPTION)
```

Namely the client determines the name of a server offering the desired type of service by means of the

`rd(tos,srv)` operation. If there is no server available for such type of service (`nrd(tos,srv)`), then the client will have to handle the unexpected situation by means of some process `EXCEPTION`. Note that this behavior description forces to the server to provide a service to the client, in order to avoid an `EXCEPTION`. Obviously a real client would wait for some time before entering in the second branch. However, with the given specification are modelling the possibility of having this situation, abstracting from temporal conditions.

3 Correct composition of components

3.1 Interfaces compatibility

We now introduce a notion of *compatibility* among processes in order to determine whether two processes — specified as two Linda agents — conform one another. The idea is applying this notion to the analysis interfaces compatibility.

Let us first define the notion of *successful computation* which, intuitively speaking, denotes the absence of deadlock in all possible alternative executions of a process.

Definition 3.1 [Successful computation] A process P is a *possible failure* if there exists an agent P' such that $P \longmapsto^* P' \not\longmapsto$ and P' is not structurally equivalent to a store (i.e., a parallel composition of tuples). On the contrary, a process P is *successful* if P is not a possible failure.

Before defining the notion of compatibility among processes,we introduce the notion of *synchronizable processes*. This notion is needed for technical reason, and its need will be better clarified later.

Definition 3.2 [Synchronizable processes] A process P provides an input a for an agent Q if there exist two processes P' and Q', such that $P \stackrel{\overline{a}}{\Longrightarrow} P'$ and $Q \stackrel{\alpha}{\Longrightarrow} Q'$, where $\alpha \in \{a, \underline{a}\}$. Two processes P and Q are *synchronizable* if P provides an input for Q or Q provides an input for P.

Definition 3.3 [Compatible processes] A process P is semi-compatible with a process Q, written $P \mathcal{C} Q$ (and the relation $\mathcal{C}$ is called a semi-compatibility), iff:

(i) If P is not successful then P and Q are synchronizable

(ii) If P only can proceed by $\neg t$ transition then $Q \longmapsto^* \stackrel{\bar{t}}{\not\longrightarrow}$.

(iii) If $P \stackrel{\tau}{\longrightarrow} P'$ then $P' \; \mathcal{C} \; Q$.

(iv) If $P \stackrel{t}{\longrightarrow} P'$ and $Q \stackrel{\bar{t}}{\longrightarrow} Q'$ then $P' \; \mathcal{C} \; Q'$.

(v) If $P \stackrel{\underline{t}}{\longrightarrow} P'$ and $Q \stackrel{\bar{t}}{\longrightarrow}$ then $P' \; \mathcal{C} \; Q$.

(vi) If $P \stackrel{\neg \, t}{\longrightarrow} P'$ and $Q \stackrel{\bar{t}}{\not\longrightarrow}$ then $P' \; \mathcal{C} \; Q$.

A relation $\mathcal{C}$ is a compatibility if both $\mathcal{C}$ and $\mathcal{C}^{-1}$ are semi-compatibilities. We say that two processes P and Q are compatible, and we denote it by $P \diamond Q$, if there exists a compatibility relation $\mathcal{C}$, such that $P \; \mathcal{C} \; Q$.

Intuitively speaking, two Linda processes are compatible if for each possible action offered by one of them there is a corresponding answer from the other one, and vice-versa. Notice that condition(i) has a technical justification as it avoids considering two unrelated processes (viz., two processes that do not share any action) compatible.

When processes are defined with a finite number of states (even if they present an infinite behavior), it is worth observing that it is possible to implement a tool capable of automatically checking the compatibility of two processes. Obviously, depending on the structural complexity of the processes, the cost of checking might be very high. In any case, even when infinite behavior is dealt with, the usefulness of a tool is clear. Thus, a negative answer showing the non-compatibility of two components could prevent from wrong compositions. Obviously, the compatibility of two generic processes is not always decidible. In fact, the second condition of previous definition may introduce a very high cost when two Linda agents are analysed to check their compatibility. This condition has been included only to deal with the *nrd* action. Observe that always an agent P exhibits a $\neg t$ transition, a compatible agent Q must prevent the existence of a tuple $\langle t \rangle$, even after a (indeterminate) number of $\longmapsto$ transitions. And this could be rather expensive in terms of resource consuming. However, from a practical point of view, it is usual to complement the description of a component with state finite processes, and we could restrict us to a subset of Linda (no including the *nrd* primitive). We can, for instance, verify that the processes:

```
CLIENT = out(#me,tos,qry).in(#me,ans).CLIENT
```

and:

```
SERVER = in(c,tos,qry).out(c,ans).SERVER
```

are compatible. Since `CLIENT` $\stackrel{\tau}{\longrightarrow} \stackrel{\overline{qry}}{\longrightarrow}$ `CLIENT'` and `SERVER` $\stackrel{qry}{\longrightarrow}$ `SERVER'`, we check the compatibility of the two new processes

```
CLIENT'=in(#me,ans).CLIENT
```

and

```
SERVER'=out(c,ans).SERVER.
```

Now, CLIENT' $\xrightarrow{ans}$CLIENT and SERVER' $\xrightarrow{\tau}\xrightarrow{\overline{ans}}$SERVER. Therefore, both processes belong to a compatibility relation, and then, we can conclude that CLIENT and SERVER are compatible.

On the other hand, we can observe that an "eager" server (that may terminate if there are no pending queries) such as:

```
SERVER2 = in(c,tos,qry).out(c,ans). SERVER2
          +  nrd(c,tos,qry).0}
```

is not compatible with CLIENT.

If we consider the notion of bisimilarity defined in other process calculi, we can observe that compatibility provides a different way of comparing processes. In fact, whereas two bisimilar processes present the "same" behavior, two compatible processes describe two "complementary" behaviors.

Definition 3.4 [Bisimilar processes] Two processes P and Q are similar, written $P\ \mathcal{S}\ Q$, (and the relation $\mathcal{S}$ is called a similarity) iff $P\ \mathcal{S}\ Q$ implies that $\forall\alpha\ \epsilon A\bigcup\{\tau\}$:

$$\text{If } P \xrightarrow{\alpha} P',\ \exists Q'\ \ Q \xrightarrow{\alpha} Q' \quad \text{and } P'\ \mathcal{S}\ Q'.$$

A relation $\mathcal{S}$ is a bisimilarity if both $\mathcal{S}$ and $\mathcal{S}^{-1}$ are similarities. We say that two processes are bisimilar (and we denote it by $P \sim Q$), if exists a bisimilarity relation, such that $P\ \mathcal{S}\ Q$.

Theorem 3.5 *If $P \diamond Q$ and $Q \sim R$ then $P \diamond R$.*

Proof. The proof is this result is direct from conditions imposed in the definition of compatibility. We only have to prove that the relation $\mathcal{C} = \{(P, R) : P \diamond Q \text{ and } Q \sim R\}$ is a semi-compatibility. Let P,R be two $\mathcal{C}$-related processes, i.e $P\ \mathcal{C}\ R$. We analyze the different possibilities:

(i) If P is not successful, by the definition of compatibility (condition 1 for $P \diamond Q$) and bisimilarity, P and R are synchronizable.

(iii) If $P \xrightarrow{\tau} P'$, we observe that $P' \diamond Q$. And if we consider $Q \sim R$, then we obtain $P'\ \mathcal{C}\ R$.

(iv) If $P \xrightarrow{t} P'$ and $R \xrightarrow{\overline{t}} R'$. As $Q \sim R$, there exists Q' such that $Q \xrightarrow{\overline{t}} Q'$ and $Q' \sim R'$. In the same way, as $P \diamond Q$, we have by condition 4 that $P' \diamond Q'$ Therefore $P'\ \mathcal{C}\ R'$.

(v) If $P \xrightarrow{\underline{t}} P'$ and $R \xrightarrow{\overline{t}} R'$. As $Q \sim R$, there exists Q' such that $Q \xrightarrow{\overline{t}} Q'$ and $Q' \sim R'$. In the same way, as $P \diamond Q$, we have by condition 5 that $P' \diamond Q$. Therefore $P'\ \mathcal{C}\ R$.

(vi) If $P \xrightarrow{\neg t} P'$ and $R \not\xrightarrow{\overline{t}}$. As $Q \sim R$, we can affirm that exists Q' such that $Q \not\xrightarrow{\overline{t}}$ and $Q' \sim R'$. As $P \diamond Q$, by condition 6 we obtain $P' \diamond Q$. Therefore $P'\ \mathcal{C}\ R$.

□

We also can prove that the compatibility of two processes implies the success of their parallel composition. This result is not surprising, because conditions of Definition 3.3 have been selected to ensure that parallel composition is deadlock free.

Proposition 3.6 *If $P \diamond Q$ then $P \parallel Q$ is successful.*

Proof. Suppose that $P \diamond Q$ but $P \parallel Q$ is not successful.Then, there exists $\mathbf{F}$ (a possible failure) such that $P \parallel Q \longmapsto^* \mathbf{F}$, $\mathbf{F}$ is not a set of tuples and $\mathbf{F} \not\longmapsto$. We will prove that it is not possible by induction on the number n of τ-transitions leading to $\mathbf{F}$.

(i) *Base Case.* Suppose $n = 0$. Then $P \parallel Q = \mathbf{F} \not\longmapsto$, therefore $P \overset{\tau}{\not\longrightarrow}$ and $Q \overset{\tau}{\not\longrightarrow}$. If P or Q is a stuck process, then, by the first condition of compatibility, we infer that P and Q are synchronizable (it means that a exists such that $P \overset{\overline{a}}{\Longrightarrow} P'$ and $Q \overset{\alpha}{\Longrightarrow} Q'$, where $\alpha \in \{a, \underline{a}\}$). And then, we have $P \parallel Q \Longrightarrow P' \parallel Q' \overset{\tau}{\longrightarrow}$, which is a contradiction. Another possibility is that $P \overset{\neg t}{\longrightarrow}$ and $Q \overset{t}{\longrightarrow}$ for some action t (or viceversa); but this is contradictory with the second condition of compatibility.

(ii) *Inductive hypothesis.* $\forall P', Q'.\ P' \diamond Q'$, if $P' \parallel Q' \longmapsto^k \mathbf{F}$ with $k < n$, then either $\mathbf{F} \longmapsto^*$ or $\mathbf{F}$ is structurally equivalent to a store (i.e. a parallel composition of tuples).

(iii) *General Case.* Suppose that $(P \parallel Q) \longmapsto (P' \parallel Q') \longmapsto^{n-1} \mathbf{F}$. Then the initial transition is due to one of the following situations:
   (a) $P \overset{\tau}{\longrightarrow} P'$. Then, since $Q = Q'$ we have that $P' \diamond Q'$
   (b) $P \overset{t}{\longrightarrow} P'$ and $Q \overset{\overline{t}}{\longrightarrow} Q'$. Then, we have that $P' \diamond Q'$
   (c) $P \overset{\underline{t}}{\longrightarrow} P'$ and $Q \overset{\overline{t}}{\longrightarrow}$. Then, we have $P' \diamond Q'$
   (d) $P \overset{\neg t}{\longrightarrow} P'$ and $Q \overset{\overline{t}}{\not\longrightarrow}$. Then, we have $P' \diamond Q'$
   (e) Or the symmetrical situations for Q.
   Because of $P \diamond Q$ in every situation, we can apply the inductive hypothesis and deduce that either $\mathbf{F} \overset{\tau}{\longrightarrow}$ or $\mathbf{F}$ is a parallel composition of tuples, again obtaining a contradiction.

□

For instance, the previously described processes `CLIENT` and `SERVER` are compatible, and Proposition 3.6 ensures that their parallel composition is a success. Although this is not a surprising result, we will introduce below the notion of compatibility with respect to a set of tuples, which will capture some interesting details making the successful composition of components dependent on the open running system.

3.2 Store sensitive compatibility

In Linda, inter-process communication occurs only via a shared store (or dataspace) which is a (multi)set of tuples inserted, extracted or deleted by the concurrent processes.

In order to have an explicit treatment of the store, we now define a compatibility relation that takes into account the situation of the store. As we will see, we can obtain a similar result concerning successful computation in the presence of compatibility. An advantage of having an explicit reference to the store is the possibility of establishing dynamic compatibility checking. Indeed, a Linda-based computation is characterized by the store's evolution, so that the set of tuples included into the store governs each computation step. This way, the aim of the following definition is to enable run-time (store-sensitive) compatibility checking.

If we observe the Definition 3.3, the processes P and Q are agents that belong to $\mathcal{L}'$. The fact of compatibility checking implies the processes must have tuples, because they are obtained when we apply τ-transitions (`out`-actions). But we must put the tuples into the store, so we have the processes without tuples. How do we get this? We can define two types of τ-transitions: one of this transition appears when it synchronizes an output ($\overline{t}$) and an input (t or $\underline{t}$) and the second type emerges from the creation of the tuples (out). In this way, we can separate the processes which originate the tuples and put them (the tuples) into the store. Therefore it is necessary to rewrite the rule number (1) of the transition system. We will name τ-actions(rules 7 and 8)when we want to synchronize processes and τ_{out}-actions when we want to indicate output :

$$out(t).P \xrightarrow{\tau_{out}} \langle t \rangle \parallel P$$

And now we can define a relation of compatibility w.r.t a data shared space.

Definition 3.7 [Compatible processes with respect to a store] Let P and Q be two processes in $\mathcal{L}$, P is semi-compatible with Q w.r.t a store St, written $P\ \mathcal{C}_{St}\ Q$, iff:

(i) If P is not successful then P and $Q \parallel Store$ are synchronizable.

(ii) If P only can proceed by $\neg t$ transition then $Q \longmapsto^* \xrightarrow{\overline{t}}\!\!\!\!\!\!/\ \ $ and St does not include the tuple $\langle t \rangle$.

(iii) If $P \xrightarrow{\tau_{out}} \langle t \rangle \parallel P'$ then $P'\ \mathcal{C}_{St \parallel \langle t \rangle}\ Q$.

(iv) If $P \xrightarrow{t} P'$ y $St \xrightarrow{\overline{t}} St'$ then $P'\ \mathcal{C}_{St'}\ Q$.

(v) If $P \xrightarrow{\underline{t}} P'$ y $St \xrightarrow{\overline{t}}$ then $P'\ \mathcal{C}_{St}\ Q$.

A relation $\mathcal{C}_{St}$ is a compatibility w.r.t. the store St if both $\mathcal{C}_{St}$ and $\mathcal{C}_{St}^{-1}$ are semi-compatibilities w.r.t. the same store St. We say that two processes P

and Q are compatible w.r.t. St, and we denote it by $P \diamond_{St} Q$, if there exists a compatibility relation $\mathcal{C}_{St}$, such that $P \mathcal{C}_{St} Q$.

As in Definition 3.3, condition (1) is introduced for technical reasons to avoid two unrelated processes to be considered compatible. In this case, since the compatibility relation is relative to a certain store, we allow to have unrelated (without sharing complementary actions) compatible processes, whenever they might be related through the store. That is, when two processes do not present any common (complementary) behavior, but one of them, after consuming a tuple from the store, synchronizes with the other process, then they may present a compatible behavior.

Notice that two processes which are not compatible in the sense of Definition 3.3 can be compatible with respect to a convenient store. For example, if we consider the process `CLIENT2=in(schd,init).CLIENT`, modeling a client which needs to be initialized by some scheduler before behaving as `CLIENT`, it is not compatible with the process `SERVER` previously defined in this Section. However, we observe that they are compatible w.r.t. a store containing the tuple `<schd,init>`. Here, we can see how the new introduced notion makes more flexible the compatibility checking between two processes. In fact, the compatibility of a client and a server could depend on the actions already made by a third component previously created (the one which has put the convenient tuple into the store.)

And then, if we study the two relations $\diamond$ and $\diamond_{St}$ in a basic case, we obtain the next proposition.

Proposition 3.8 *If $P \diamond Q$ then $P \diamond_{\emptyset} Q$.*

Proof.

(i) Trivial.

(iii) If $P \xrightarrow{\tau_{out}} \langle t \rangle \parallel P'$. As $P \diamond Q$ then $\langle t \rangle \parallel P' \diamond Q$, which implies by the Definition 3.7(cond. 3) $P' \diamond_{\langle t \rangle} Q$.

□

The result of Proposition 3.6 can be extended to $\diamond_{St}$, obtaining the following result.

Proposition 3.9 *If $P \diamond_{St} Q$ then $P \parallel Q \parallel St$ is successful.*

Proof. Analogous to the proof of Proposition 3.6. □

Proposition 3.9 ensures the success of the computation of a pair of processes in presence of a suitable store. In practice, this result can be used:

- both for checking the compatibility of a component and of a running system w.r.t. the current store (characterizing the current state of the execution),
- and for conditioning the acceptance of a given component into an open running system so as to wait for a suitable state of the store in order to ensure the success of the overall system.

It is worth observing that the two relations $\diamond$ and $\diamond_{St}$ are closely related. This is rather natural, because the notion of compatibility w.r.t. a store is defined in terms of the complementary behavior of a process with respect to another one (as it is made in the notion of compatibility), and with respect to the store, which is dealt with as one more process (a parallel composition of tuples). Informally, $\diamond$ can be seen as a different presentation of $\diamond_{St}$. What we mean with this is that the compatibility with respect to a store could be defined in terms of the compatibility relation, where the store can be seen as one more process: a parallel composition of tuples.

The advantage of using the presentation given by $\diamond_{St}$ is its usefulness from the automatic checking perspective. Although the new compatibility relation is relevant *per se* (because the store plays an important role in the interaction of components, and it is explicitly considered), a more interesting point is the possibility of building an automatic checking tool capable of determining which is the store (if any) that makes two given processes compatible. Such a tool would manipulate processes belonging to $\mathcal{L}$, that is, no containing tuples. Whereas agents involved in the compatibility relation given in Definition 3.3 are from $\mathcal{L}'$ (note that the *out* primitive introduces tuples), the idea is to distinguish agents modeling a component (with no occurrences of tuples) from those representing a store (a parallel composition of tuples). However, if we observe the second condition of Definition 3.7, we see that the process P may proceed to a process P' containing a tuple: it happens when the τ transition comes from an *out* action (obviously, this is not the case when the τ transition corresponds to a synchronization action). This inconvenient could be easily solved by considering two different τ transitions, one to represent *out* actions, and one to model synchronization. Thus, we assume that processes modeling component protocols will always belong to $\mathcal{L}$.

Theorem 3.10 *Let P and Q be two processes, and let St be a set of tuples. If $P \diamond_{St} Q$ then $P \diamond (Q \parallel St)$.*

Proof. This proof can be reduced to proving that the relation $\mathcal{C} = \{(P, Q \parallel St) : P \diamond_{St} Q\}$ is a semi-compatibility. Thus, given $P \mathcal{C} (Q \parallel St)$, which means $P \diamond_{St} Q$, we analyze the different possibilities that are considered into the Definition 3.3 (where second and sixth conditions are not considered, because we are not dealing with the *nrd* action.)

(i) If P is not successful, by the Definition 3.7 (condition (1) for $P \diamond_{St} Q$), P and $(Q \parallel St)$ are synchronizable.

(iii) If $P \xrightarrow{\tau_{out}} P'$, we observe that $P' \diamond_{St} Q$. This implies $P' \mathcal{C} Q \parallel St$.

(iv) If $P \xrightarrow{t} P'$ and $(Q \parallel St) \xrightarrow{\overline{t}}$. we observe the following situation:

- $St \xrightarrow{\overline{t}} St'$. Then, as $P \diamond_{St} Q$, by the condition (4), we have $P' \diamond_{St'} Q$. And by the definition of $\mathcal{C}$, $P' \mathcal{C} Q \parallel St'$.

(v) If $P \xrightarrow{\underline{t}} P'$ and $(Q \parallel St) \xrightarrow{\overline{t}}$, we have the next alternative:

- $St \xrightarrow{\bar{t}}$. Then, as $P \diamond_{St} Q$, by the condition (5), we have $P' \diamond_{St} Q$. And by the definition of $\mathcal{C}$, $P'\ \mathcal{C}\ Q \parallel St$.

□

If we analyze the proof of this theorem, we observe that a more general result could have been proved. For example, the following theorem.

Theorem 3.11 *Let P and Q be two processes, and let St be a set of tuples. If $P \diamond (Q \parallel St)$ then $P \diamond_{St} Q$.*

Proof. Analogous to the proof of Theorem 3.10, it can be reduced to proving that the relation $\mathcal{C}_{St} = \{(P, Q) : P \diamond (Q \parallel St)\}$ is a semi-compatibility.

(i) Trivial

(iii) If $P \xrightarrow{\tau_{out}} \langle t \rangle \parallel P'$. As $P \diamond (Q \parallel St)$ then we infer that $\langle t \rangle \parallel P' \diamond (Q \parallel St)$, where P' does not contain any tuple. Then $P'\ \mathcal{C}_{St \parallel \langle t \rangle}\ Q$.

(iv) If $P \xrightarrow{t} P'$ and $St \xrightarrow{\bar{t}} St'$ then $Q \parallel St \xrightarrow{\bar{t}} Q \parallel St'$. As $P \diamond (Q \parallel St)$ then by the Definition 3.7 (cond. 4), $P' \diamond (Q \parallel St')$, and $P'\ \mathcal{C}_{St'}\ Q$.

(v) If $P \xrightarrow{\underline{t}} P'$ and $St \xrightarrow{\bar{t}}$.As $P \diamond (Q \parallel St)$ then by the Definition 3.7 (cond.5) $P' \diamond (Q \parallel St)$, therefore $P'\ \mathcal{C}_{St}\ Q$.

□

In fact, if two processes P and Q are compatible under a store St, for every partition of that store, $St = St_P \parallel St_Q$, we could prove that $P \parallel St_P$ and $Q \parallel St_Q$ are compatible too.

4 Concluding remarks

Linda is a coordination language where inter-process communication can only occur through a set of tuples, and the main novelty of our proposal consists of defining a compatibility relation taking into account the situation of the store. The advantage of this is the possibility of establishing dynamic compatibility checking. That is, when a component has to be incorporated into an already executing system (seen as another component), the compatibility has to be analyzed dynamically, and the "static" specification is not enough because it presents the behavior of a component from its instantiation. Indeed, the advantage of using a Linda-based formalism is that a Linda computation is characterized by the store's evolution, in such a way that the set of tuples included into the store governs each computation step. This is not made in other proposal, where other formalisms, like CSP or π-calculus, are used. We believe that this Linda's feature can be potentially used to establish the compatibility of executing components, by using the store to have information about the current state of the component.

Indeed, some of the issues covered in this paper have also been dealt with in other proposals. In the context of software architecture Allen and Garlan

[1] use the process algebra CSP to describe synchronization of components and connectors, while having some limitations concerning the dynamic change of configurations. Another proposal improving the expressiveness of interaction descriptions by using π-calculus was presented by Canal [7]. Some of the ideas proposed in [7] have already been applied to CORBA in [8]. In this case, dynamic interaction among components (dynamic change of topology) can be better expressed than in CSP. Other works, like [2], propose the use of (a subset of) π-calculus to describe interaction patterns for components so as to reduce the cost of verifying correctness properties in dynamic, open systems. Our proposal somehow combines these two last lines by defining a notion of process compatibility in the style of [7,8], while focussing on the automatic, run-time checking of properties in dynamic, open systems in the style of [2].

Our future work will be devoted to define an *inheritance* relation over processes in order to promote the reusability and substitutability of interaction descriptions, and to study how this affects compatibility and successful computations. We are also planning to develop an automatic tool (by applying model checking techniques) to check compatibility in order to explore the practical application of our proposal and to analyze and experiment the cost of checking properties in practical real-word cases.

New generation component-based platforms (e.g., .NET) will allow protocol information to be directly included in the metalanguage description (e.g., in XML) of a component. In this perspective, our future work will be devoted to develop a methodology for coding protocol information as metalanguage descriptions and for checking composition properties by analyzing their metalanguage descriptions.

References

[1] R. Allen and D. Garlan,“ A formal basis for architectural connection,” ACM Transactions on Software Engineering and Methodology, 6(3):213–249, 1997.

[2] A. Bracciali, A. Brogi, and F. Turini. *Coordinating interaction patterns.* In Proceedings of 16th ACM Symposium on Applied Computing, 2001.

[3] A. Brogi and J. M. Jacquet. *On the Expressiveness of Linda-like Concurrent Languages.* Electronic Notes in Theoretical Computer Science, 16, 1998.

[4] A. Brogi and J. M. Jacquet. “On the expressiveness of coordination models”. In Coordination Languajes and Models: 3rd International Conference, number 1594 in lncs, pages 134–149. Springer-Verlag, 1999.

[5] N. Busi, R. Gorrieri, and G. Zavattaro. *Comparing three semantics for linda-like languages.* Theoretical Computer Science, 1998.

[6] N. Busi, R. Gorrieri, and G. Zavattaro. *A process algebraic view of linda*

coordination primitives. Electronic Theoretical Computer Science, 192:167–199, 1998.

[7] C. Canal. "Un Lenguaje para la Especificación y Validación de Arquitecturas de Software ". PhD thesis, Dept. Lenguajes y Ciencias de la Computación, University of Málaga, 2001.

[8] C. Canal, L. Fuentes, E. Pimentel, J. Troya, and A. Vallecillo. *Extending Corba Interfaces with Protocols*. The Computer Journal, 44(5):448–462, 2001.

[9] N. Carriero and D. Gelernter. *Linda in Context*. Communications of the ACM, 32(4):444–458, 1989.

[10] D. Chappell. "Understanding ActiveX and OLE " . Microsoft Press, 1996.

[11] D. Gelernter and N. Carriero. *Coordination Languages and Their Significance*. Communications of de ACM, 35(3):97–107, 1992.

[12] G. T. Leavens and M. Staraman, editors. "Foundations of Component-Based Systems ". Cambridge University Press, 2000.

[13] J. Magee, J. Kramer, and D. Giannakopoulou. *Behaviour analysis of software architectures*. Kluwer Academic Publishers, 1999.

[14] DCE. *The Open Group of Distributed Computing Environment*.

[15] Microsoft Corporation. *.NET Programming the Web*.

[16] OMG. *The Common Object Request Broker: Architecture and Specification*. Object Managemant Group.

[17] Sun Microsystems Inc. *Enterprise JavaBeans*.

[18] R. Milner. "A Calculus of communicating systems ". Springer-Verlag, 1989.

[19] R. Milner, J. Parrow, and D. Walker. *Calculus of mobie processes*. Journal of Information and Computation, 100:1–7, 1992.

[20] E. Najm, A. Nimour, and J. Stefani. *Infinite types for distributed objects interfaces*. In Proceedings of the third IFIP conference on Formal Methods for Open Object-based Distributed Systems - FMOODS'99. Kluwer Academic Publishers, 1999.

[21] N. Wang, D. C. Schmidt, and C. O'Ryan. "An Overview of the CORBA Component Model ". Object Technology Series. Addison-Wesley, 2000.