* Manuscript

Software Composition with Linda

Ana M. Roldan [1]

Department of Computer Science, University of Huelva, Spain

Ernesto Pimentel [1]

Department of Computer Science, University of Málaga, Spain, Spain

Antonio Brogi

Department of Computer Science, University of Pisa, Italy

Abstract

Nowadays, Tuple Spaces have turned out to be one of the most fundamental abstractions for coordinating communicating agents. Some models, as Linda, were presented as a set of inter-agent communication primitives which can virtually be added to any programming language. These models have the advantage of capturing both communication and synchronisation in a natural and simple way. In this paper, we analyze 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. This definition of compatibility takes into account the state of a global store (tuple space), which gives relevant information about the current execution of the system. 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. In particular, we prove that compatibility implies successful computation (i.e. without deadlock). We also argue that Linda features some advantages with respect to similar proposals in the context of dynamic compatibility checking. In this context, the success of the composition of a pair of agents in presence of a suitable store can be useful to condition the acceptance of a given component into an open running system. In order to extend our approach to complex systems, where constructing a system involves more than two components, we propose the use of distributed tuple spaces as the glue to join components.

Key words: Coordination languages, components, software architecture, compatibility, interaction, process algebras.

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 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-shelf" (COTS) components. The first component-oriented platforms were CORBA [23] and DCE [22], developed by OSF (Open Software Foundation) and OMG (Object Management Group). Several other platforms have been developed after them, like COM/DCOM [12], CCM [28], EJB [24], and the recent .NET [20].

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 [18]. Many of them are based on process algebras, and extend interfaces with a description of their concurrent behaviour [1,2,8,9,19,21], such as behavioural types or role-based representations.

The objective of this work is to explore the usability of the coordination language Linda [10] 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). Tuple Spaces are considered one of the most fundamental and successful abstractions for coordinating concurrent activities [10,13]. These are a number of reasons, both technical and pragmatic, to consider Linda an appropriate alternative for specifying protocol behaviour of software

Email addresses: `amroldan@diesia.uhu.es` (Ana M. Roldan), `ernesto@lcc.uma.es` (Ernesto Pimentel), `brogi@di.unipi.it` (Antonio Brogi).

[1] The work of Ana M. Roldán and Ernesto Pimentel has been partially supported by the Project TINC2004-07943-C04-01 funded by the Spanish Ministry of Education and Science

components. On the technical reasons, tuple spaces have the advantage of capturing both communication and synchronization in a simple and natural way. Tuples themselves represent resources than can be communicated, shared and exchanged, without the need to use additional synchronization mechanisms. On the other hand, there are many kinds of problems that map naturally to the tuple space view of the system, namely, when there are many different sorts of concurrent agents that want to exchange information among them. In this sense, tuples represent the least common denominator of data structures and so, can be used to easily model almost any variety of stuff. Note that the use of Linda [30] can provide a unifying framework for interface definition protocols for interoperability, data exchange protocols for heterogenous databases, software components, etc. Linda's communication model features interesting properties, such as space and time uncoupling [10], as well as a great expressive power to specify concurrent and distributed systems [3].

The main 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 an extension of the process algebra presented in [6].
(ii) We define a notion of *store sensitive compatibility* to formalize the compatibility of two agents 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 contextualize the compatibility of agents in the perspective of dynamic compatibility checking. We show that the compatibility of two agents implies that their interaction will be successful, in the sense we will define later (Definition 1).The importance of the notion of compatibility relates to the possibility of performing *a priori* verification of complex interacting systems [5].
(iii) We present a software architecture as a collection of interconnected computational and data components. Indeed we consider software systems as compositions of specifications of their components.

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 compatibility with respect to a store. In the section 4, we study the compatibility in a real example (*an electronic auction*). The next section, shows software systems as architectural descriptions of their components and finally, some concluding remarks and future work are discussed.

2 Specifying component protocols in Linda

2.1 A Linda calculus

Linda [10] was the first coordination language [16], 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 [6], 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 [25].

A *tuple space* is a finite multi-set of tuples, where a *tuple* is a finite sequence $\langle n_1, \ldots, n_h \rangle$ of names. The set of all tuples will be denoted by $\mathcal{T}$ and ranged over by t, u (possibly indexed). To express pattern matching, processes employ *tuple templates* inside *rd()*, *in()* and *nrd()* operations. A tuple template is a finite sequence of names, some of which are possibly prefixed by "?" and are used as placeholders. A tuple template $\langle m_1, \ldots, m_k \rangle$ *matches* a tuple $\langle n_1, \ldots, n_h \rangle$ via a *name substitution* σ (denoted by $match(\langle m_1, \ldots, m_k \rangle, \langle n_1, \ldots, n_h \rangle, \sigma)$) iff:

(i) $k = h$, and
(ii) $\forall\ i \in [1..k]$ either $m_i = n_i$ or ($m_i = ?x_i$ and $\sigma(x_i) = n_i$).

When the substitution is not relevant, we will simply write $match(\langle m_1, \ldots, m_k \rangle, \langle n_1, \ldots, n_h \rangle)$ to express that $\langle m_1, \ldots, m_k \rangle$ matches $\langle n_1, \ldots, n_h \rangle$.

Name substitutions can be applied to tuples and tuple templates as one may expect. Namely if $\langle m_1, \ldots, m_h \rangle$ is a tuple (or a tuple template) and σ a name substitution then

$$\langle m_1, \ldots, m_h \rangle \sigma = \langle n_1, \ldots, n_h \rangle$$

where $\forall\ i \in [1..h]$:

$$n_i = \begin{cases} m_i & if\ m_i = ?x_i \\ \sigma(m_i) & \text{otherwise} \end{cases}$$

(1)	$P \| 0 \equiv P$
(2)	$P \| Q \equiv Q \| P$
(3)	$(P \| Q) \| R \equiv P \| (Q \| R)$
(4)	$P + 0 \equiv P$
(5)	$P + P \equiv P$
(6)	$P + Q \equiv Q + P$
(7)	$(P + Q) + R \equiv P + (Q + R)$

Table 1
Structural Congruence

For instance the tuple template $\langle foo, ?x \rangle$ matches the tuple $\langle foo, 3 \rangle$ via the substitution $[3/x]$. The application of the former substitution to the term :

$$out(\langle data, x \rangle).in(\langle foo2, ?x \rangle)$$

yields

$$out(\langle data, 3 \rangle).in(\langle foo2, ?x \rangle)$$

For the sake of simplicity, we do not consider a hiding operator -namely every new name introduced in a process is considered as a global name.

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

$$P ::= 0 \mid \alpha.P \mid P + P \mid P \parallel P \mid A(\tilde{x})$$
$$\alpha ::= rd(t) \mid nrd(t) \mid in(t) \mid out(t)$$

where 0 denotes the empty process, t denotes a tuple or a tuple template and where $A(\tilde{x})$ is a process invocation. For any process identifier A there must be a unique defining equation $A(\tilde{y}) = P$. Defining equations provide recursion since P may contain any process identifier, even A itself.

In order to define the operational semantics for $\mathcal{L}$, we first define a *structural congruence* as usual in process algebra. The structural congruence $\equiv$ is defined as the smallest congruence satisfying the axioms in Table 1.

Extending [6], the operational semantics of $\mathcal{L}$ can be modeled by a labelled transition system defined by the rules of Table 2. Notice that the configurations of the transition system extend the syntax of agents by allowing parallel composition of tuples. Formally, the transition system of Table 2 refers to the extended language $\mathcal{L}'$ defined as:

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

(1)	$out(t).P \xrightarrow{\tau_{out}} \langle t \rangle \parallel P$	(6)	$\dfrac{P \xrightarrow{\alpha} P'}{P + Q \xrightarrow{\alpha} P'}$
(2)	$\langle t \rangle \xrightarrow{\bar{t}} 0$	(7)	$\dfrac{P \xrightarrow{t} P' \wedge Q \xrightarrow{\bar{t}} Q'}{P \parallel Q \xrightarrow{\tau} P' \parallel Q'}$
(3)	$\dfrac{\exists \sigma \in \Sigma,\ u \in \mathcal{T}\ : match(t, u, \sigma)}{in(t).P \xrightarrow{u} P\sigma}$	(8)	$\dfrac{P \xrightarrow{\underline{t}} P' \wedge Q \xrightarrow{\bar{t}}}{P \parallel Q \xrightarrow{\tau} P' \parallel Q}$
(4)	$\dfrac{\exists \sigma \in \Sigma,\ u \in \mathcal{T}\ : match(t, u, \sigma)}{rd(t).P \xrightarrow{\underline{u}} P\sigma}$	(9)	$\dfrac{P \xrightarrow{\alpha} P' \wedge \alpha \neq \neg t}{P \parallel Q \xrightarrow{\alpha} P' \parallel Q}$
(5)	$nrd(t).P \xrightarrow{\neg t} P$	(10)	$\dfrac{A(\tilde{y}) = P \wedge P[\tilde{x}/\tilde{y}] \xrightarrow{\alpha} P'}{A(\tilde{x}) \xrightarrow{\alpha} P'}$
(11)	$\dfrac{P \xrightarrow{\neg t} P' \wedge \nexists u \in \mathcal{T} : (match(t, u) \wedge Q \xrightarrow{\bar{u}})}{P \parallel Q \xrightarrow{\neg t} P' \parallel Q}$		

Table 2
Transition system for $\mathcal{L}$.

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 $\bar{t}$. Rules (3) and (4) describe the behaviour of the prefixes $in(t)$ and $rd(t)$ wwhose labels are (u and $\underline{u}$ respectively) are tuples matched by the template t. Rule (5) shows the behaviour of the prefixe $nrd(t)$ with the label $\neg t$. Rule (6) is the standard rule for choice composition. Rules (7) and (8) model complementary input (rd,in)/output (out) operations. Notice that rd operation has not effect over Q as it 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 any tuple u matching with t, as stated by rule (11). Finally, rule (10) models process definition.We consider the transition system closed under the usual structural axioms for parallel and choice operators such as shown in Table 1.

The rules of Table 2 are used to define the set of derivations for a Linda system. Note that we distinguish two different silent transitions: one corresponding to the output actions (τ_{out}) and another one for synchronization (τ). In order to model derivations that are valid, we consider transitions which can proceed by means of τ, τ_{out} or nrd steps. Formally, this corresponds to introduce the following derivation relation:

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

Notice that the above operational characterization of $\mathcal{L}$ assumes the so-called *ordered* semantics of the output operation [7]. 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 is 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. In order to show 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:

```
SERVER1 = in(?c,tos,?qry).out(c,ans).SERVER1
```

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:

```
CLIENT1 = out(me,tos,qry).in(me,?ans).CLIENT1
```

where `me` is the identifier of the client process [10].

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 it 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 now employs the tuple format `<sender,receiver,message>` for the messages 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`.

3 Correct composition of components

In Linda, inter-process communication occurs only via a shared store (or dataspace) which is a parallel composition 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. 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.

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

(iv) of definition 2 we infer $P' \diamond_{St} (Q \parallel R)$. Then by definition of $\mathcal{C}_{St}$, we obtain $P' \parallel Q\, \mathcal{C}_{St}\, R$.

(b) $P \parallel Q \xrightarrow{t} P \parallel Q'$ and $St \xrightarrow{\bar{t}}$. As $Q \xrightarrow{t} Q'$, then $Q \parallel R \xrightarrow{t} Q' \parallel R$. By the definition of $\mathcal{C}_{St}$, we know $P \diamond_{St} Q \parallel R$, so $P \diamond_{St} Q' \parallel R$. Finally, $P \parallel Q'\, \mathcal{C}_{St}\, R$.

(v) If $P \parallel Q \xrightarrow{\neg t}$

(a) $P \parallel Q \xrightarrow{\neg t} P' \parallel Q$. As $P \diamond_{St} Q \parallel R$, by the condition ($v$) of definition 2 we infer $P' \diamond_{St} Q \parallel R$ and $\forall u \in \mathcal{T}$ such that $match(u,t)$ $St \not\xrightarrow{u}$. Then by definition of $\mathcal{C}_{St}$, we obtain $P' \parallel Q\, \mathcal{C}_{St}\, R$.

(b) $P \parallel Q \xrightarrow{\neg t} P \parallel Q'$. As $P \diamond_{St} Q \parallel R$, by the condition ($v$) of definition 2 we infer $P' \diamond_{St} Q \parallel R$ and $\forall u \in \mathcal{T}$ such that $match(u,t)$ $St \not\xrightarrow{u}$. Then by definition of $\mathcal{C}_{St}$, we obtain $P \parallel Q'\, \mathcal{C}_{St}\, R$.

The rest of the cases are inferred in a similar way, and the proof that ${\mathcal{C}_{St}}^{-1}$ is also a semi-compatibility is analogous.

4 A case study

We show a system (*an electronic auction*) in evolution and prove that the composition is safe. In order to specify the protocols that define the behaviour of an auctioneer and a bidder, we use Linda as coordination language.

We consider a uniform second-price auction, often called Vickrey auction [Vic61], where the bids are sealed, each bidder is ignorant of other bids, and the item is awarded to highest bidder at a price equal to the second-highest bid. In other words, a winner pays less than the highest bid. If, for example, bidder A bids 10, bidder B bids 15, and bidder C offers 20, bidder C would win, however he would only pay the price of the second-highest bid, namely 15.

The Auctioneer starts an auction by adding to the store a tuple of the form `<onsale,good>` where good is a description of the good offered. The Auctioneer also starts a timer so as to continue receiving and processing bids till the end of the auction round, when the bidder who made the highest bid will be declared winner (by producing a tuple of the form `<sold,id>`).

```
Auctioneer = Auction || Timer

Auction=out(onsale,good).Auction(nobid,noid)

Auction(bid,id)=
```



```
        in(?newbid,?newid).Auction(bid',id')
        +
        in(timeout).in(onsale,?good).out(sold,id).0

Timer = rd(onsale,?good).out(timeout).0
```

On the other hand, a bidder waits for an auction round to start, it decides whether or not to sail a bid, and then waits for the end of the round.

```
Bidder(id) = rd(onsale,?good). ( out(bid,id).BidderWait(id)
                                 +
                                 BidderWait(id) )

BidderWait(id) = rd(sold,?winner).0
```

Note that internal computations are not explicitly expressed. For example, the way the auctioneer updates its value of the best offer received so far is abstracted by using fresh names (bid' and id') or the way in which the timer actually waits before producing the timeout tuple.

Following the previous protocols and the definition of the compatibility w.r.t. the store, we will study if the processes `Auctioneer` and `Bidder(id)` are compatible w.r.t a special store. Notice that 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.

Let us now analyze the compatibility of the processes `Auctioneer` and `Bidder(id)` w.r.t. the empty store.

Let us first verify whether `Auctioneer` is semi-compatible with `Bidder(id)` w.r.t. the empty store. The only compatibility condition that applies to `Auctioneer` w.r.t. the empty store is condition (ii). Therefore we observe that:

`Auctioneer` $\mathcal{C}_{\{\}}$ `Bidder(id)`
  if (by condition (ii))
`Auction(nobid, noid)||Timer` $\mathcal{C}_{\{<onsale,good>\}}$ `Bidder(id)`
  if (by condition (iv))
`Auction(nobid, noid)||out(timeout).0` $\mathcal{C}_{\{<onsale,good>\}}$ `Bidder(id)`
  if (by condition (ii))
`Auction(nobid, noid)||0` $\mathcal{C}_{\{<onsale,good>,<timeout>\}}$ `Bidder(id)`
  if (by condition (iii))
`in(onsale, ?good).out(sold, noid).0||0` $\mathcal{C}_{\{<onsale,good>\}}$ `Bidder(id)`
  if (by condition (iii))

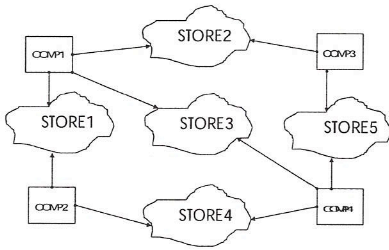

Fig. 1. Software architecture based on components

`out(sold, noid).0||0` $\mathcal{C}_{\{\}}$ `Bidder(id)`
  if (by condition (ii))
`0||0` $\mathcal{C}_{\{<\texttt{sold,noid}>\}}$ `Bidder(id)`
  if
true

Let us now verify that also `Bidder(id)` is semi-compatible with `Auctioneer` w.r.t. the empty store:

`Bidder(id)` $\mathcal{C}_{\{\}}$ `Auctioneer`
  if (by condition (i))
`out(bid, id).BidderWait(id)` + `BidderWait(id)` $\mathcal{C}_{\{<onsale,good>\}}$ `Auction(nobid, noid)||Timer`
  if (by condition (ii))
`BidderWait(id)` $\mathcal{C}_{\{<onsale,good>,<bid,id>\}}$ `Auction(nobid, noid)||Timer`
  if (by condition (i))
`0` $\mathcal{C}_{\{<\texttt{sold,id}>}$ `0}`
  if
true

Again, we conclude `Auctioneer` $\mathcal{C}_{\{\}}$ `Bidder(id)`. And finally, we can affirm `Auctioneer` is compatible with `Bidder(id)` w.r.t. the empty store.

5 Software Architecture

Software Architecture refers to the level of software design in which the system is represented as a collection of computational and data components interconnected in a certain way [14]. Thus, it is focused on those properties of software systems that derive from their structure, i.e. from the way in which their components are combined.

Software systems can be described in Linda by composing the specifications of their components. We use the notion of role for describing the dynamic behavior of components, in the same sense as it is used in *ADLs*. *Roles* are partial interfaces for describing the behavior of each component, and we explicitly represent system architecture as a set of roles for each component. This allows modular specification of component behavior and permit more efficient pairwise compatibility checks.

Definition 7 *We define a* component *as a set of roles :*

$$Comp = \{R_1, R_2, ..., R_n\}$$

where each R_i is a process in $\mathcal{L}$.

We consider distributed stores that allow the connection of components through roles. In this situation, stores will contain shared tuples used to synchronize components. The connection of several components in an architecture will be represented by an attachment among roles and stores. This is formalized as follows.

Definition 8 *Let $\mathcal{S}$=$\{St_1, St_2, ..., St_k\}$ be a set of stores and let $\mathcal{C}$= $\{Comp_i\}_{i=1}^m$ be a set of components, where each $Comp_i$ is represented by a set of roles $\{R_{i_1}, R_{i_2}, ..., R_{i_{n_i}}\}$. We define an* attachment *as a mapping ψ from $\mathcal{S}$ to $\mathcal{P}(\{R_{i_j}\}_{i,j})$.*

Now, we can define an architecture as a *context* composed by a set of stores which contains the synchronization information (tuples), a set of components and an attachment ψ describing the links among roles.

Definition 9 *Consider a software system composed by several components, $\mathcal{C}$= $\{Comp_i\}_{i=1}^m$. Let $\mathcal{S}$=$\{St_1, St_2, ..., St_k\}$ be a set of stores. We say that*

$$< \mathcal{C}, \mathcal{S}, \psi >$$

is a context *if ψ is an attachment that satisfies the following properties:*

(i) $\forall$ *St* ϵ $\mathcal{S}$, $|\psi(St) \bigcap Comp_i| \leq 1$, *for each i.*
(ii) $\forall$ *St,St'* ϵ $\mathcal{S}$. $\psi(St) \bigcap \psi(St') = \emptyset$.

The previous properties show two important aspects of our notion of context. The first property states that two roles that describe the same component cannot interact with the same store. In the second one, we establish a disjoint distribution of the roles among the stores.

Definition 10 *A context is a* successful context *if the parallel composition of its components is successful.*

We consider that a software system is composed by a set of components that are described by roles. In fact, it is necessary to establish some property that ensures the composition of these components is successful, i.e. the interaction among components is deadlock free.

Considering these connections are established pairwise, we can claim that the context could be successful (free of deadlocks). This will be true if we can separate the tuple space in small tuple spaces with related roles. The compatibility of these subsystems will allow us to ensure the successful of the global system.

In the proposal we show in this section, we consider a software architecture where the components are related through stores. Each store is shared by two roles and each of them belongs to one component. In this sense, the roles describe the behavior of the components with respect to a particular/specific store. Although this proposal is satisfied sometimes, the problem will appear when several components (upper to 2) are related through the same store.

The way to proceed in these cases is twofold. First, a naive solution is based on separating the roles involved in each of these stores into two set of roles, and then, the parallel composition of each group of roles will provide a store with only two roles. Note that the separation of a group of roles into two groups to analyze their compatibility is independent of the any this separation is made. This claim is supported by Prop. 6. However, this solution, as we mentioned in Section 3, presents an important drawback: when parallel composition is massively used, the complexity of compatibility checking increases exponentially. Thus, a complementary approach to overcome this inconvenient is trying to construct a disjoint distribution of the roles sharing a store, then producing small independent stores, where the number of interacting roles is minimized. This process could be achieved by making a static analysis of the system in a similar way as Ciancarini and Castagnetti proposed in [11], in the context of Shared Prolog. Note that this process does not depend on the specific features of the components involved.

When a new component is inserted into a running system, the resulting context may exhibit an erroneous behavior (e.g. deadlock situations) if the interaction provided by the incoming component is not the expected one. The information given by the store, together with the protocol specification of the components, may prevent us from undesired behaviors. In some situations, the insertion of a component may be delayed till certain expected tuples appear in the store, or alternatively, a component could be accepted as part of a system even if the resulting context is not successful, whenever it becomes successful by adding some tuples to the store (this situation is referred to as *feasible* context in [2]). All these possibilities correspond to different ways of applying the notion of compatibility w.r.t. a store to a Linda-based architecture.

6 Concluding remarks

Linda is a coordination language where inter-process communication can only occur through a set of tuples. Our proposal consists of defining a software architecture taking into account the specifications of its components [27] made in Linda. Thus, we consider a compatibility relation that permits us to establish 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 and in [15] show new challenges for component-based software engineering such mobility, adaptability and recourse awareness. Another proposal improving the expressiveness of interaction descriptions by using π-calculus was presented by Canal [8]. Some of the ideas proposed in [8] have already been applied to CORBA in [9]. 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 [8,9], while focussing on the automatic, run-time checking of properties in dynamic, open systems in the style of [2]. Following these approaches, we consider software systems are structured as a collection of interacting computational and data components interconnected through the specifications of their components [14]. Our future work will be devoted to define an *inheritance* relation over agents 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. *A framework for specifying and verifying the behavior of open systems.* Journal of Logic and Algebraic Programming. Elsevier. Vol 63(2), pages 215-240, 2005.

[3] A. Brogi and J. M. Jacquet. *On the Expressiveness of Linda-like Concurrent Languages.* Electronic Notes of Theoretical Computer Science, 16(2), 1998.

[4] A. Brogi and J. M. Jacquet. *On the Expressiveness of Coordination via Shared Dataspaces.* Science of Computer Programming, 46(1-2):71-98, 2003.

[5] A. Brogi, E. Pimentel, and A. Roldán. *Compatibility of Linda-based Component Interfaces.* Electronic Notes in Theoretical Computer Science, 2002.

[6] N. Busi, R. Gorrieri, and G. Zavattaro. *Comparing Three Semantics for Linda-like Languages.* Theoretical Computer Science, 240:49-90, 2000.

[7] N. Busi, R. Gorrieri, and G. Zavattaro. *A Process Algebraic View of Linda Coordination Primitives.* Electronic Theoretical Computer Science, 192:167–199, 1998.

[8] 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.

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

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

[11] T. Castagnetti and P. Ciancarini. *Static Analysis of a Parallel Logic Language Based on the Blackboard Model.* Journal of Parallel and Distributed Computing 13(4): 412-423 (1991)

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

[13] S. Ducase and T. Hofmann and O. Nierstrasz. *OpenSpaces: An Object-Oriented Framework For Reconfigurable Coordination Spaces.* In COORDINATION 2000. Springer-Verlag, 2000.

[14] D. Garlan and D.E. Perry. *Special Issue on Software Architecture. IEEE Trans. on Software Engineering, 21(4), April 1995.*

[15] D. Garlan and B. Schmerl. *Component-Based Software Engineering Pervasive Computing Environments.* In Proceedings of 4th ICSE Workshop on Component-Based Software Engineering, Toronto (Canada)2001.

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

[17] J. M. Jacquet and K. De Bosschere and A. Brogi. *On Timed Coordination Languages.* In COORDINATION 2000, pages 81-98. Springer-Verlag, 2000.

[18] G. T. Leavens and M. Staraman, editors. *Foundations of Component-based Systems.* Cambridge University Press, 2000.

[19] J. Magee, J. Kramer, and D. Giannakopoulou. *Behaviour Analysis of Software Architectures.* Kluwer Academic Publishers, 1999.

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

[21] 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.

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

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

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

[25] R. Milner. *A Calculus of Communicating Systems.* Springer-Verlag, 1989.

[26] R. Milner, J. Parrow, and D. Walker. *Calculus of Mobile Agents.* Journal of Information and Computation, 100:1–7, 1992.

[27] A. Roldán, E. Pimentel and A. Brogi. *Safe Composition of Linda-based Components.* Electronic Notes in Theoretical Computer Science 82,No.6, 2003.

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

[29] W. Vickrey *Counterspeculation, auctions, and Competitive Sealed Tenders.* Journal of Finance Vol. 16 (March, 1961) pp. 8-37.

[30] P. Wegner *Coordination as Constrainted Interaction.* In COORDINATION 1996, LNCS 1061, pp. 28-33, April 1996