*Journal of Universal Computer Science, vol. 11, no. 10 (2005), 1676-1694*
*submitted: 20/5/05, accepted: 23/9/05, appeared: 28/10/05 © J.UCS*

# Coordinating Behavioral Descriptions of Components

**Silvia Amaro**
(National University of Comahue, Argentina
samaro@uncoma.edu.ar)

**Ernesto Pimentel**
(University of Málaga, Spain
ernesto@lcc.uma.es)

**Ana M. Roldán**
(University of Huelva, Spain
amroldan@diesia.uhu.es)

**Abstract:** Component-based Software Development is an emerging discipline in the field of Software Engineering. In this context, coordination languages may be used to specify the interactive behavior of software components. Our proposal is oriented towards defining a framework for describing the behavior of components in terms of coordination models. In particular, we define a way to complement interface description languages in order to describe components such that the information about the services provided by a component can be extended with details on how these services should be used. We illustrate our approach by applying the proposed framework to two substantially different coordination models: Linda and Reo; the former representing the family of data-oriented coordination models, and the latter a new channel-based model. Although we consider both models to show the feasibility of our proposal we hope this study help us to define an interaction description language based on Reo for component coordination, as has already been done in the context of Linda.
**Key Words:** Coordination, process algebra, modular embedding, expressiveness.
**Category:** D.2.12

## 1 Introduction

Component-Based Software Engineering (CBSE) is an emerging discipline in the field of Software Engineering. In spite of its relative newness, a lot of attention has being devoted to CBSE both in the academic and in the industrial world. The reason for this growing interest is the need for systematically developing open systems 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 and DCE, developed by OSF (Open Software Foundation) and OMG (Object Management Group). Several other platforms have been developed subsequently, such as COM/DCOM where components may have several interfaces, each one describing the signature of the supported operations; CCM (CORBA Components Model) also contemplates that components may describe not only the services they support, but also the interfaces they require from others components during their execution; EJB, and the recent .NET.

Available component-oriented platforms address software interoperability by using Interface Description Languages (IDLs). An interface is described as a service abstraction which defines the operations that the service supports, independently of any particular implementation. Interfaces can be described using many different notations, depending on the information that the designer wants to include, and the level of detail of the specification. 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 with the aim of adapting or wrapping them in order 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 variances in the different component behaviours. In general, the use of IDL descriptions during run-time is quite limited. They are mainly used to discover services and to dynamically build service calls. However, there are no mechanisms currently in place to deal with automatic compatibility checks or dynamic component adaption which are among the most commonly required facilities for building component-based applications in open and independently extensive systems.

The objective of this work is to explore the capability of coordination models for specifying the interaction behavior of software components. Indeed, our aim is to propose these models as a means of complementing current interface description languages in a similar way as behavioral types [Magee et al. 99] or role-based representations [Canal 01, Canal et al. 01]. In order to evaluate the advantages and drawbacks of our approach we consider two well known coordination models to illustrate their possibilities for composing software. Taking into account traditional classification of data-oriented and control-oriented coordination languages, we have chosen a very representative model of the first group (Linda) and a modern evolution of the second one (Reo).

Linda [Carriero and Gelernter 89] is one of the most representative coordination languages, originally presented as a set of inter-agent communication primitives which can be added to virtually 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*. 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. Reo [Arbab 04] is a channel-based coordination model which enforces the use of connectors for the coordination of concurrent processes or component instances in a component-based system. Channels are the basic connectors from which more complex connectors can be constructed through composition. The channel composition mechanism in addition to the great diversity of channel types (with a well defined behavior) allows the construction of many different connectors, where each connector imposes a specific coordination pattern.

Our approach is based on proposing a generic specification language based on process algebras, which can be instantiated by different coordination models. In particular we propose a CCS-like notation (possibly) parameterized with the communication medium (e.g. tuple spaces or channels). To illustrate how this language can be used to specify the interaction exhibited by components, we instantiate it to Linda and Reo respectively, showing their application by means of an example.

The expressive power of both models (Linda and Reo) has been independently studied using different approaches. A very complete study on the expressive power of Linda was carried out by Brogi and Jacquet in [Brogi et al. 01, Brogi and Jacquet 98]. On the other hand, when Reo was introduced, Arbab [Arbab 04] provided a number of examples to show the expressive power of his proposal, simulating in a simple and elegant way different communication mechanisms.

The rest of the paper is organized as follows. In Section 2 we outline some issues related to component interoperability and an illustrative example is presented. Section 3 is devoted to introducing both interaction models, their semantics and the corresponding calculi used to encapsulate both models. A case study is presented in section 4 together with a comparative analysis of the expressiveness of both models. Finally, we present some conclusions and ideas for future work.

## 2 Interoperability of components

In the process of application assembly, one of the key issues is interoperability. Syntactic interoperability is well defined and understood by commercial components models and platforms. They allow the interoperation of heterogeneous components based on syntactic agreements. This is possible because of the interface definitions generated by the use of interface description languages (IDL).

However this sort of interoperability is not enough in large systems, where knowledge of the services offered by components, and in some cases the services required from other components in run-time, is not enough to guarantee that they will suitably interoperate. In fact, two more levels of interoperability can be identified: the semantic level and the protocol level [Vallecillo et al. 03]. In any case, it is necessary to consider compatibility and substitutability checks. *Compatibility* can be described as the ability of two components to work together properly if connected; *substitutability* refers to the possibility of one component being replaced by another one. At protocol level the notion of compatibility among components implies the need to take into account: (a) their blocking conditions, and (b) the order in which components expect their services to be required. This is necessary to solve coordination and synchronization problems, to ensure that the restrictions imposed on the components interactions when communicating are preserved, and their communication is deadlock free. On the other hand in order to check substitutability, that is, to test if component A can be replaced by component B, we need to check that both components are consistent with respect to the relative order among the incoming and outgoing messages. Moreover, we need to verify that all messages accepted by A are also accepted by B, and that B's outgoing messages are a subset of A's outgoing messages.

In order to solve the interoperability problems mentioned we propose the use of coordination models to enhance component interfaces with a description of an abstract component interaction protocol. We suggest the use of Linda and Reo as mechanisms for describing the abstract interaction protocol of software components. Intuitively, when using the Linda-based model for checking compatibility and substitutability the state of the tuple space must be considered, whereas using the model based on Reo will depend on the connector considered.

We are interested in addressing the following major points:

1. how components can be provided with an interface useful for solving protocol interoperability problems,

2. how components specified using the calculus based on coordination models can be assembled together and what the impact of the composition is on the overall behavior,

3. how checking for compatibility and substitutability on the components in an assembled system can be done dynamically.

In order to illustrate our proposal, let us describe a simplified version of a real patient monitoring system. It was first introduced by Papadopoulos and Arbab [Papadopoulos and Arbab 98] to show the potential there is for controlling coordination languages in order to express dynamically reconfigurable software architectures. The basic scenario involves a number of monitors and nurses. There is one monitor for each patient, recording readings of the patient's state of health, in response to a request received. In addition, a monitor can also send data in the case of exceptional situations. A nurse is responsible for periodically checking the patient's state of health by asking the corresponding monitor for readings; furthermore a nurse should respond to receiving exceptional data readings.

As we can see in the interface below, a monitor offers one method that allows the user to request the periodical readings. The nurse interface defines two methods to be invoked by the environment. Method *normal* implements the main service offered by the process, it receives readings of the patient's state of health on the parameter *normalState* and processes them. On the other hand the method *signal* allows the nurse to treat emergency cases, which are captured on the *emergencyState* parameter.

```
interface Monitor {
    void request();
}

interface Nurse {
    void signal ([in]Data emergencyState);
    void normal ([in] Data normalState);
}
```

From these interfaces it is very difficult to discern the way in which a monitor and a nurse will behave if they are integrated in a software application. Nothing is said concerning their interactions and the rules governing them. In fact, this interface says nothing about the possibility of a monitor sending emergency signals. Moreover the fact that the monitor will deliver the emergency signals is not specified and no information is given about the nurse's obligation to firstly deal with on emergency situation. In the next section we will concentrate on how to add protocol information to the description of the interfaces using the two different alternatives we are analyzing: Linda and Reo. In particular how an emergency signal must be captured with a higher priority than a normal reading. The simplicity of the selected example is related with the idea of showing the problems suggested in such a simple way and the fundamental differences in the solutions presented.

## 3 Interaction models based on Coordination models

From an architectural point of view application construction is seen as a fundamentally compositional activity, in which existing elements are used. That is, systems consist of a collection of components, interconnected in some way. In this environment process algebras are the formalism frequently used for describing concurrent systems. Given their expressivity and characteristics they are widely accepted as a method for describing and analyzing software systems, as a combination of components. In this context a process is an entity capable of performing some internal actions and interacting with other processes in its environment. Interactions are primitive synchronization actions that may cause data exchanges between processes influencing their behaviors. A process algebra focuses on the specification and manipulation of process terms as induced by a collection of operator symbols. Most process algebras contain in their syntaxis a set of process names, a set of channel or event names used for the synchronization and communication between processes, constants for representing the inactive processes, internal actions and a set of data names transmitted. They also contain basic operators to build finite processes, communication operators to express concurrency and some notion of recursion to capture infinite behavior. The meaning of the processes generated which combine these elements is established by an operational semantics given by a transition system which associates processes with behavior.

On the other hand in the context of software architectures the fundamental concepts are components, connectors and configurations. However, other formalisms for their specification and analysis are coordination models and languages. Indeed configuration of architectural descriptions and coordination are very closely linked concepts. They both view systems as being comprised of components and interconnections and support the construction of complex entities as well as the composition of more elementary ones. Finally they both understand changing the state of a system is an activity performed at the level of component interconnection rather than within the purely internal computational functionality of a particular component.

In order to accomplish our aim of defining a way to complement interface description languages in order to describe components, we suggest combining the advantages provided by process algebras with those offered by coordination models in the definition of a framework for describing and composing components. To do this we propose complementing current interface description languages by adding the specification of components behavior in terms of a CCS-like notation, where primitive actions correspond to a coordination model and the synchronization rules depend on the communication framework. For instance, in the case of Linda, primitives like *rd*, *in*, and *out* must be considered, whereas in Reo other actions are taken into account (*write*, *take* and *read*). In the same sense while in Linda the synchronization is based on a tuple space, in Reo the communication is effected through channels. This must be reflected by the operational semantics of the corresponding process calculus thus introducing the convenient transition rules.

In the context of coordination models we can identify data-driven languages and control oriented ones. One of the most representative data-oriented coordination models is Linda, which is based on a set of communication primitives accessing to a shared tuple space. On the other hand, a new channel-based language, based on composition of communication channels, is being consolidated. Reo [Arbab 04] can be considered a sophisticated evolution of a control-oriented coordination model. In spite of the different abstraction level exhibited by both models, we will compare both of them in order to show the very high expressive power provided by Reo for simulating different communication protocols.

### 3.1 The Linda-based interaction model

The main feature of these coordination models is that the state of the computation at any time is defined in terms of both values of the data received or sent and the actual configuration of the coordinated components. Following [Busi et al. 00], we propose the language $\mathcal{L}$ containing the communication primitives of Linda. These primitives permit us to add a tuple ($out$), to remove a tuple ($in$), and to test for the presence of a tuple ($rd$) in the shared dataspace. As the only medium for synchronization and communication between processes is the shared dataspace there are no channel names. The language $\mathcal{L}$ also includes the standard prefix, choice and parallel composition operators in the style of CCS.

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

$$P ::= 0_{\mathcal{L}} \;\mid\; A.P \;\mid\; P + P \;\mid\; P \parallel P \mid recX.P$$
$$A ::= rd(t) \;\mid\; in(t) \;\mid\; out(t)$$

where $0_{\mathcal{L}}$ denotes the empty process and $t$ denotes a tuple.

The operational semantics of $\mathcal{L}$ can be modelled by a labelled transition system defined by the rules of Table 1. 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 1 refers to the extended language $\mathcal{L}'$ defined as:

$$P' ::= P \;\mid\; P' \,||_{\mathcal{L}}\, \langle t \rangle$$

Rule $(1)_{\mathcal{L}}$ states that the output operation consists of an internal move which creates the tuple $\langle t \rangle$. Rule $(2)_{\mathcal{L}}$ shows that a tuple $\langle t \rangle$ is ready to offer itself to the environment by performing an action labelled $\overline{t}$. Rules $(3)_{\mathcal{L}}$ and $(4)_{\mathcal{L}}$ describe the behavior of the prefixes $in(t)$ and $rd(t)$ whose labels are $t$, and $\underline{t}$, respectively. Rule $(5)_{\mathcal{L}}$ is the standard rule for choice composition. Rule $(6)_{\mathcal{L}}$ 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 $(7)_{\mathcal{L}}$ 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 rule $(8)_{\mathcal{L}}$ models the behavior of the parallel operator. We also consider the transition system

| | | | |
|---|---|---|---|
| $(1)_{\mathcal{L}}$ | $out(t).P \xrightarrow{\tau} \langle t \rangle \parallel_{\mathcal{L}} P$ | $(5)_{\mathcal{L}}$ | $\dfrac{P \xrightarrow{\alpha} P'}{P +_{\mathcal{L}} Q \xrightarrow{\alpha} P'}$ |
| $(2)_{\mathcal{L}}$ | $\langle t \rangle \xrightarrow{\overline{t}} 0$ | $(6)_{\mathcal{L}}$ | $\dfrac{P \xrightarrow{t} P' \; Q \xrightarrow{\overline{t}} Q'}{P \parallel_{\mathcal{L}} Q \xrightarrow{\tau} P' \parallel_{\mathcal{L}} Q'}$ |
| $(3)_{\mathcal{L}}$ | $in(t).P \xrightarrow{t} P$ | $(7)_{\mathcal{L}}$ | $\dfrac{P \xrightarrow{\underline{t}} P' \; Q \xrightarrow{\overline{t}}}{P \parallel_{\mathcal{L}} Q \xrightarrow{\tau} P' \parallel_{\mathcal{L}} Q}$ |
| $(4)_{\mathcal{L}}$ | $rd(t).P \xrightarrow{\underline{t}} P$ | $(8)_{\mathcal{L}}$ | $\dfrac{P \xrightarrow{\alpha} P'}{P \parallel_{\mathcal{L}} Q \xrightarrow{\alpha} P' \parallel_{\mathcal{L}} Q}$ |

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

closed under the usual structural axioms for parallel and choice operators. There are no rules for recursion since its semantics are 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. We consider the output action $\tau$ as observable transitions. Notice that the above operational characterization of $\mathcal{L}$ employs the so-called *ordered* semantics of the output operation. 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 the store can also be seen as a process which is the parallel composition of a number of tuples.

### 3.2 The Reo-based interaction model

When using channel-based coordination models the framework evolves by means of performing communication actions over input or output ends of channels to which the coordinated components are connected. In the case of Reo, the communication actions are performed over the input/output ends of a connector, then the interaction model will be parameterized with respect to the connector being considered.

Reo [Arbab 04] is a channel-based coordination model which enforces the use of connectors for the coordination of concurrent processes or component instances in a component-based system. Channels are the basic connectors from which more complex ones can be constructed through composition. The channel composition mechanism in addition to the great diversity of channel types with different semantics from the traditional ones, allows us the construction of many different connectors imposing very interesting coordination patterns. In this context communication among component instances takes place exclusively by means of input and output actions over connector ends, acting as connection points. In [Arbab and Rutten 03] a coinductive calculus based on timed data streams (TDS) for defining the semantics of Reo connectors was presented. The operational model for the behavior of Reo connectors, based on Constraint Automata introduced by Arbab et. al in [Arbab et. al. 04] can be used as a semantic model to describe the TDS language induced by Reo connectors networks .

For the specification of component interaction protocols we define a process algebra $\mathcal{R}$ based on the communication primitives of Reo. We consider a set $\mathcal{I}$ of input ends, a set $\mathcal{O}$ of output ends, and the basic actions to insert an item in a connector (*write*), to remove an item from the connector (*take*) and to capture an item without removing it (*read*). Agents in $\mathcal{R}$ are constructed by means of the prefix operator, the nondeterministic choice and the parallel composition. Formally, the syntax of $\mathcal{R}$ is defined as follows:

$$P ::= 0_{\mathcal{R}} \mid A.P \mid P + P \mid P \parallel P \mid recX.P$$
$$A ::= wr(c, v) \mid tk(c, [v]) \mid rd(c, [v])$$

where $0_{\mathcal{R}}$ denotes the empty process and $c \in \mathcal{I} \cup \mathcal{O}$ denotes an input or output end of a connector. The prefixes $wr$, $tk$ and $rd$ are shorthand for the basic operations *write*, *take* and *read* respectively. Note that in output operations the variable is optional, if it is not specified the operation succeeds when any data item is available for taking (or reading) and it is removed through the specified connector end. To make the representation of a system in $\mathcal{R}$ we can think of specifying each component by an $\mathcal{R}$-agent and then do a parallel composition of their specifications in the presence of a suitable connector.

In Reo communication is possible only in the presence of a connector, and then in order to define the operational semantics of $\mathcal{R}$ we must consider the semantics of the selected connector. We consider a connector C defined by a tuple $\langle \mathcal{I}_{\mathrm{C}}, \mathcal{O}_{\mathrm{C}}, \Sigma_{\mathrm{C}}, \longmapsto_C \rangle$, where $\mathcal{I}_{\mathrm{C}}$ and $\mathcal{O}_{\mathrm{C}}$ represent the input ends set and output ends set of connector C, respectively, $\Sigma_{\mathrm{C}}$ is the set of states, that is the possible configurations of the connector, and $\longmapsto_{\mathrm{C}} \subseteq (\Sigma_{\mathrm{C}} \times MAct) \times MAct \times (\Sigma_{\mathrm{C}} \times MAct)$ represents the labelled transition relation defining the connector behavior. $MAct$ denotes the multiset of communication actions.

When $(\langle \mathrm{C}, act \rangle, act_1, \langle \mathrm{C}', act_2 \rangle) \in \longmapsto_C$ we will write

$$\langle \mathrm{C}, act \rangle \overset{act_1}{\longmapsto}_{\mathrm{C}} \langle \mathrm{C}', act_2 \rangle$$

with the following intuitive interpretation: $act$ denotes the set of actions which when applied in parallel over the ends of the connector may result in its evolution, eventually producing a change of state. The set $act_1$ denotes the actions actually applied, and $act_2$ represents pending actions at any end of the connector. Pending actions are actions *write* or *take* which, in the presence of a

| | |
|---|---|
| $(1)_{\mathcal{R}}$ | $act \cdot P \xrightarrow{act} P$ |
| $(2)_{\mathcal{R}}$ | $\dfrac{P_1 \xrightarrow{act} P_1'}{P_1 + P_2 \xrightarrow{act} P_1'}$ |
| $(3)_{\mathcal{R}}$ | $\dfrac{P_1 \xrightarrow{act} P_1'}{P_1 \parallel P_2 \xrightarrow{act} P_1' \parallel P_2}$ |
| $(4)_{\mathcal{R}}$ | $\dfrac{P_1 \xrightarrow{act_1} P_1' \; P_2 \xrightarrow{act_2} P_2'}{P_1 \parallel P_2 \xrightarrow{act_1 \uplus act_2} P_1' \parallel P_2'}$ |
| $(5)_{\mathcal{R}}$ | $\dfrac{P \xrightarrow{act} P' \; \langle \mathrm{C}, act\rangle \overset{act}{\rightarrowtail}_{\mathrm{C}} \langle \mathrm{C}', \emptyset\rangle}{\langle P, \mathrm{C}\rangle \longrightarrow_{\mathrm{C}} \langle P', \mathrm{C}'\rangle}$ |
| $(6)_{\mathcal{R}}$ | $\dfrac{P_1 \xrightarrow{act_1} P_1' \; P_2 \xrightarrow{act_2} P_2' \; \langle \mathrm{C}, act\rangle \overset{act_1}{\rightarrowtail}_{\mathrm{C}} \langle \mathrm{C}', act_2\rangle}{\langle P_1 \parallel P_2, \mathrm{C}\rangle \longrightarrow_{\mathrm{C}} \langle P_1' \parallel P_2, \mathrm{C}'\rangle}$ |

**Table 2:** Transition System for $\mathcal{R}$

synchronous behavior, remain pending when applied in parallel with read actions. These multisets must respond to the relation $act = act_1 \uplus act_2$.

When it is clear from the context, we omit the subindex C when referring to the sets $\mathcal{I}$, $\mathcal{O}$,$\Sigma$. The rules giving the connector behavior will be generated from its corresponding constraint automata.

The operational semantics of $\mathcal{R}$ depends on the connector considered. Formally, given a connector C with a behavior defined via a labelled transition relation $\overset{\alpha}{\longmapsto}_{\mathrm{C}}$, we define the transition system $\langle \mathcal{R}, \mathrm{C}, \longrightarrow_{\mathrm{C}}\rangle$, where $\mathcal{R}$ is the set of programs described in the process algebra, C is the considered connector and $\longrightarrow_{\mathrm{C}} \subseteq (\mathcal{R} \times \mathrm{C}) \times (\mathcal{R} \times \mathrm{C})$ the transition relation defined by rules $(5)_{\mathcal{R}}$ and $(6)_{\mathcal{R}}$ of table 2. Note that the definition of $\longrightarrow_{\mathrm{C}}$ depends on the auxiliary labelled transition system $\langle \mathcal{R}, Act, \longrightarrow\rangle$ where $\longrightarrow \subseteq \mathcal{R} \times Act \times \mathcal{R}$ is the transition relation defined by rules $(1)_{\mathcal{R}}$ to $(4)_{\mathcal{R}}$. The transition $\overset{\overline{act}}{\rightarrowtail}_{\mathrm{C}}$ represents the derivation $\overset{\overline{act_1}}{\longmapsto}_C \overset{\overline{act_2}}{\longmapsto}_C \ldots \overset{\overline{act_n}}{\longmapsto}_C$, where $\overline{act} = \biguplus_i \overline{act}_i$. We also consider both systems to be closed with respect to the structural axioms for choice and parallel operators. There are no rules for recursion, its semantics is defined by the structural axiom $recX.P \equiv P[recX.P/X]$.

The rules giving the connector behavior will be generated from its corresponding constraint automata, using the following algorithm: let C be a connector defined by the sets $\mathcal{I}$ and $\mathcal{O}$ of input ends and output ends respectively, and its constraint automata $CA_{\mathrm{C}}$ given by:

$$CA_{\mathrm{C}} \equiv (Q_{\mathrm{C}}, \mathcal{N}_{\mathrm{C}}, \rightarrow_{\mathrm{C}}, Qo_{\mathrm{C}})$$

where $\mathcal{N}_\mathrm{C} = \mathcal{I} \cup \mathcal{O}$. We associate a name $\mathrm{C}_q$ to every $q \in Q_\mathrm{C}$ to indicate the connector C is in a state $q$. As the automata transitions are labelled with the maximum number of nodes over which data can flow simultaneously, we can identify from them the input ends and output ends of the connector over which input or output operations occurring synchronously produce a state change. The symbol $\models$ represents the satisfaction relation resulting from interpreting data constraints over data assignments. Now, the transitions can be generated as follows:

1. For each transition $(q \xrightarrow{N,g}_\mathrm{C} p) \in \rightarrow_\mathrm{C}$, a transition in $\longmapsto$ is generated as follows:

$$\langle \mathrm{C}_q, act_\delta \rangle \overset{act_\delta}{\longmapsto}_\mathrm{C} \langle \mathrm{C}_p, \emptyset \rangle$$

   where $\delta$ is any data assignment function such that $\delta \models g$, y $act_\delta$ is defined as $(act^{wr})_\delta \cup (act^{tk})_\delta$, where:

$$(act^{wr})_\delta = \{wr(I, \delta(I)) : I \in \mathcal{I} \cap N\}$$
$$(act^{tk})_\delta = \{tk(O, \delta(O)) : O \in \mathcal{O} \cap N\}$$

2. for each transition rule $\langle \mathrm{C}_q, \overline{act} \rangle \overset{\overline{act}}{\longmapsto}_\mathrm{C} \langle \mathrm{C}_p, \emptyset \rangle$ generated in (i), suppose $\overline{act} = \overline{act^{tk}} \dot{\cup} \overline{act^{wr}}$, the disjunt union of the $tk$ actions and the $wr$ actions that can be applied over the connector ends. For each $\overline{act}' \subseteq \overline{act^{tk}}$, we construct $\overline{act^{rd}} = \{rd(O,t) : tk(O,t) \in \overline{act}'\}$ and generate a rule $\langle \mathrm{C}_q, (\overline{act} - \overline{act}') \cup \overline{act^{rd}} \rangle \overset{\overline{act^{rd}}}{\longmapsto}_\mathrm{C} \langle \mathrm{C}_q, \overline{act} - \overline{act}' \rangle$

We need to consider the $rd$ operation particularly because of its non destructive condition. The last rule considers the situation in which at least one $rd$ operation is applied synchronously with other communication operations. In this case only $rd$ operations succeed, the other communication actions ($wr$ and $tk$) remain pending over the corresponding ends until the environment provides the necessary conditions for them to proceed, by the application of some other rule.

In the process of composing components specified in $\mathcal{R}$, the connector constrains the behavior of the overall system, imposing its own behavior. This leads to a level of composition flexibility which is highly desirable in component based systems. Due to the great diversity of communication patterns possible in Reo, this model, in contrast with the one based on Linda, makes the production of different systems composed out of the same set of components possible, by the use of different connectors with a well defined semantic.

## 4 Specifying components protocols

As we have already mentioned, it is very difficult to know the behavior of a component just from its interface. Thus, this sort of specification has proved to be inadequate in many situations. Therefore, more complete specifications are needed. One way of fulfilling this need is by providing “behavioral” specifications of the components. In this sense, we propose the use of the process algebras defined previously for the specification of dynamic and evolving systems.

The following protocol intends to model the interaction between a nurse and a monitor, such as was introduced in Section 3.1 using the calculus $\mathcal{L}$ based on Linda.

Firstly, we show the protocol describing the behavior of a monitor, (`Monitor`). A monitor is periodically requested to send the readings of the state of health of its corresponding patient, and to respond accordingly. It is possible that it detects emergency situations in the patient, sending this (emergency) signal to be captured by a nurse.

```
Monitor =
   out(signal,emergencyState).Monitor
   +
   in(request).
      (out(normal,normalState).Monitor
       +
       out(signal,emergencyState).out(normal,normalState).Monitor
      )
```

On the other hand, a nurse (`Nurse`) must check the patient's state. Thus he/she could request from the monitor new readings about patient's state of health. But it is important to notice that any emergency warning must be attended to first.

```
Nurse =
   in(signal,emergencyState).Monitor
   +
   nrd(signal,emergencyState).out(request).
      (in(normal,normalState).Nurse
       +
       in(signal,emergencyState).in(normal,normalState).Nurse
      )
```

It is worth noting that the protocol specified by `Nurse` and `Monitor` only deals with behavioral descriptions, whereas computational details are abstracted from. For instance, no information is given about how the nurse should proceed when an emergency is detected. Note also as the sum operator is understood in the classical way, priority in attending emergency cases is not ensured.

Now we will show the use of $\mathcal{R}$ for specifying interaction protocols over the same example. A monitor receives a request for its data registers on the patient health state readings. Eventually the monitor may detect abnormal situations and in this case it has to send an emergency warning signal. Emergency situations have priority for being attended. We can specify the monitor behavior as follows:

```
MONITOR =
   tk(requestIn).
       (MONITOR1
        +
        wr(signalOut,<emergencyState>).MONITOR1
       )
   +
   wr(signalOut,<emergencyState>).MONITOR

MONITOR1 =
   wr(normalOut,<normalState>).MONITOR
```

The monitor only needs to receive a piece of data in the connection point *requestIn*, and this action is interpreted as a request for information.

The nurse is responsible for checking the patients state of health. He or she requests a monitor for their data registers writing a token in the connection point associated to this action. A nurse must also attend to any warning signals, which must be attended to first. The nurse behavior can be defined as follows:

```
NURSE =
   wr(requetOut,token).NURSE1
   +
   tk(signalIn,<emergencyState>).NURSE

NURSE1 =
   tk(normalIn,<normalState>).NURSE
   +
   tk(signalIn,<emergencyState>).tk(normalIn,<normalState>).NURSE
```

Note that in the specification neither the monitor, nor the nurse, ensures the priority in attending to emergency cases, because of the non deterministic choice between attending to the periodical readings and attending to the emergency readings. In this scenario it seems clear that the expected behavior of the composition of a nurse and a monitor is achieved only when selecting a connector which enforces the required priorities.

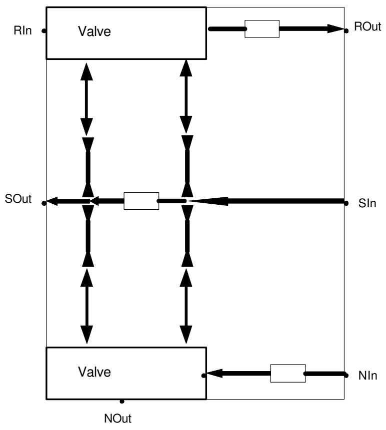


**Figure 1:** CMN Connector

With this aim in mind we selected the connector *CMN* shown in Figure 1. The connector is defined by the tuple

$$\langle \{RIn, SIn, NIn\}, \{ROut, SOut, NOut\}, \Sigma_{CMN}, \longmapsto_{CMN} \rangle$$

In Table 3 we give the transitions in the labelled transition relation which defines its behavior. Since this connector presents many possible states and so many transitions, we give only those related to the blocking situation.

This connector imposes certain restrictions over its connection points which seems appropriate for solving our priority problems. The expected behavior is imposed by the effect of the two *valve* connectors (see [Arbab 04]) present in the configuration of the connector, and the channels of type *syncSpout* and *syncDrain* which are connected to the valve control connection points. From the analysis of its transition relation, we conclude that it presents the necessary behavior. In fact, it shows an asynchronous behavior, which in some cases is disable by means of an input operation over the input end *SIn* —rules (2), (3), (4) and (6)—, leaving the connector in a state in which it remains blocked until an output operation is applied over the output end *SOut* —rules (7), (9), (11) and (12). Indeed, though blocked it is possible to apply input and output operations when it is for example in state $CMN_2$ (blocked but with a data item present in the buffer associated with ROut), or in state $CMN_4$ (blocked but with a d ata item present in the buffer associated with ROut, and a data item present in the buffer associated with NIn). When composing a nurse component and a monitor component via the connector CMN, the effect is achieved regarding the input and output ports for emergency signals, which are connected to the connection

$$
\begin{array}{rl}
(1) & \langle CMN_0, \{wr(RIn,t)\}\rangle \overset{\{wr(RIn,t)\}}{\longmapsto}_{CMN} \langle CMN_1, \emptyset\rangle \\
(2) & \langle CMN_0, \{wr(SIn,t)\}\rangle \overset{\{wr(SIn,t)\}}{\longmapsto}_{CMN} \langle CMN_3, \emptyset\rangle \\
(3) & \langle CMN_1, \{wr(SIn,t)\}\rangle \overset{\{wr(SIn,t)\}}{\longmapsto}_{CMN} \langle CMN_2, \emptyset\rangle \\
(4) & \langle CMN_6, \{wr(SIn,t)\}\rangle \overset{\{wr(SIn,t)\}}{\longmapsto}_{CMN} \langle CMN_4, \emptyset\rangle \\
(5) & \langle CMN_2, \{wr(NIn,t)\}\rangle \overset{\{wr(NIn,t)\}}{\longmapsto}_{CMN} \langle CMN_4, \emptyset\rangle \\
(6) & \langle CMN_7, \{wr(SIn,t)\}\rangle \overset{\{wr(SIn,t)\}}{\longmapsto}_{CMN} \langle CMN_5, \emptyset\rangle \\
(7) & \langle CMN_2, \{tk(SOut,t)\}\rangle \overset{\{tk(SOut,t)\}}{\longmapsto}_{CMN} \langle CMN_1, \emptyset\rangle \\
(8) & \langle CMN_2, \{tk(ROut,t)\}\rangle \overset{\{tk(ROut,t)\}}{\longmapsto}_{CMN} \langle CMN_3, \emptyset\rangle \\
(9) & \langle CMN_3, \{tk(SOut,t)\}\rangle \overset{\{tk(SOut,t)\}}{\longmapsto}_{CMN} \langle CMN_0, \emptyset\rangle \\
(10) & \langle CMN_4, \{tk(ROut,t)\}\rangle \overset{\{tk(ROut,t)\}}{\longmapsto}_{CMN} \langle CMN_5, \emptyset\rangle \\
(11) & \langle CMN_4, \{tk(SOut,t)\}\rangle \overset{\{tk(SOut,t)\}}{\longmapsto}_{CMN} \langle CMN_6, \emptyset\rangle \\
(12) & \langle CMN_5, \{tk(SOut,t)\}\rangle \overset{\{tk(SOut,t)\}}{\longmapsto}_{CMN} \langle CMN_7, \emptyset\rangle
\end{array}
$$

**Table 3:** Behavioral Transitions for *CMN*

points *Sin* and *Sout* respectively.

Component interaction protocols are specified to describe the behavior of given component interfaces. In general, there are no precise guidelines about what should and should not be included in a protocol specification. It will depend, of course, on the level of abstraction or details required. So, we consider in $\mathcal{L}$ that each method of the interface can be translated to a special tuple which contains the signature of the method (selector and arguments). On the other hand, because of in Reo communication is only possible by means of input and output operations over connector ends (connection points) we must take them into account when specifying protocols. Thus, we associate an input end with each method representing a service offered by the component, and we consider an output end for each service required by the component. In the case where the method has no arguments we do not consider any object in the input operation. However, for the output operation a token is needed, just as a signal for the requested service.

## 5 Concluding remarks

In Component-based Software Development the integration of possibly heterogeneous and distributed components together to form a single application require mechanisms for controlling and managing the interactions among the active entities. With the increasing use of distributed systems and COTS, interoperability is a major issue to consider.

To overcome the limitations of commercial component models and platforms with respect to interoperability at protocol level, several proposals have been put forward in order to enhance component interfaces [Leavens et al. 00]. For instance, Doug Lea [Lea 95] proposes in 1995 an extension of the CORBA IDL called PSL to describe the protocols associated with component's methods. This approach is based on logical and temporal rules relating situations, describing potential states with respect to the roles of components, attributes and events. Although it is a very expressive approach, it does not take into account the services a component may need from other components, nor is it supported by proving tools. Later, Yellin and Strom [Yellin and Strom 97] formalize a more general approach for describing component service protocols using finite state machines that describe the services offered and components required. However, it does not support multi-party interactions and the simplicity that allows the easy checking also makes it too rigid and unexpressive for general usage in open and distributed environments. Similarly, Jun Han [Han 99] proposes an extension to IDLs that includes semantic information using a non standard notation. The behavior of the components is described in terms of constraints that are expressed in a subset of temporal logic. They are somehow similar approaches, although none of them is associated to any commercial component platform like CORBA or EJB, nor are any supported by standard tools. Bastide et al. [Bastide and Palanque 99] use Petri nets to describe the behavior of components in CORBA, but this approach inherits some of the limitations imposed by the Petri nets notation: the lack of the modularity and scalability of the specifications. On the other hand, Shaw and Garlan [Shaw and Garlan 96] proposed the use of components specified by players indicating the nature of the expected interactions, and connectors specified by roles which will be associated to component players. Although this proposal is intended as a design tool for the construction of executable configurations based on components and expert connections, it has a fixed set of connector types, so limiting the types of possible interactions; and it has no support for composite connectors.

The main objective of this paper was to define a framework for describing the behavior of components in terms of coordination models. In this sense, the basic idea is based on extending interface description languages with an explicit description of the interactive behavior of a component. To do this, we consider two formal methods, Reo and Linda. We argue that both models of coordination are mature enough to be used in the design and validation of components of large distributed systems, and the use of such methods will lead to the better design of components and component-based applications in open systems.

Although Linda and Reo were defined with a different purpose (i.e. coordination), by applying the previously explained approach they can also be used to specify the interaction behavior of software components. The information provided by this kind of protocols (either specified in Linda or Reo) may be useful for analyzing a number of properties like compatibility [Brogi et al. 02] (when two components can interact without deadlocking) or substitutability (when a component can be substituted by another one, preserving its "safe" behavior in the system).

From the specifications based on Linda and Reo introduced in section 4 we can observe that they are apparently similar with respect to expressivity. Furthermore, if we analyze the two versions of processes `Monitor` and `Nurse` described, we have no arguments to state that one approach is better than the other one. However the Linda based specification does not ensure the priority of the treatment of emergency signals, sometimes these signals are postponed temporarily (note that sum is defined in the classical way); this situation can be solved only by increasing the complexity embedded in the interaction protocol considerably. On the contrary in Reo it is not necessary to modify the specification; the solution of this problem (i.e. the complexity added by the treatment of the emergency signal) is transferred to the connector, maintaining a simple and elegant specification. A more detailed comparison between both models is discussed in [Amaro et al. 04].

These coordination models present very different abstraction levels: Linda is based on a set of communication primitives accessing to a shared tuple space, whereas Reo is also defined in terms of communications primitives, but acting on connectors which are constructed as a combination of different kinds of channels. In fact, the need to increase the complexity of the specification in Linda in order to manage certain constraints suggests that Reo is strictly more expressive than Linda in the presence of connectors imposing special communication patterns. Moreover, although the most of the connectors in Reo can be constructed by the composition of channels from a set of basic ones, there are no restrictions on channel types and behaviors. Indeed users can provide new types of channels with special behaviors.

Our future work will be devoted to carrying out a more exhaustive analysis of more complex connectors and different extensions of Linda, like MARS [Cabri et al. 00] and TuCSoN [Omicini and Zambonelli 98], where some of its expressiveness deficiencies are dealt with. We are currently exploiting the capabilities of Reo for the definition of compatibility and substitutability relations on our proposal oriented to the semiautomated checking of the aforementioned properties on an assembled system. We are interested in controlling these properties in an assembled system dynamically.

## References


[Amaro et al. 04] Amaro, S., Pimentel ,E., Roldán, A.M: "A Preliminary Comparative Study on the Expressive Power of Reo and Linda". Proc. FOCLASA'04. ENTCS (Elec. Notes Theor. Comp. Sci.), London (2004).

[Arbab 04] Arbab, F.: "Reo: a channel-based coordination model for component composition "; Mathematical. Structures in Comp. Sci. 14, 3, 329-366, ISSN 0960-1295, Cambridge University Press (2004).

[Arbab and Rutten 03] Arbab, F., Rutten, J.J.M.M.: "A Coinductive Calculus of Component Connectors", Proceed. of WADT (Recent Trends in ALgebraic Development Techniques), Wirsing, Martin, Pattinson, Dirk, Hennicker, Rolf editors. Lectures Notes in Comp. Sci., 2755, 34-55, ISBN 3-540-20537-3, Springer (2003).

[Arbab et. al. 04] Arbab, F., Baier, C., Rutten, J.J.M.M., Sirjani, M.: "Modeling Component Connectors in Reo by Constraint Automata: Extended Abstract",ENTCS (Elec. Notes Theor. Comp. Sci.), 97, 25-46 (2004).

[Brogi et al. 01] Brogi, A., Busi, N., Gabbrielli, M., Zavattaro, G.: "Comparative analysis of the expressiveness of shared dataspace coordination"; ENTCS (Elec. Notes in Comp. Sci.) 62 (2001).

[Busi et al. 00] Busi, N., Gorrieri, R., Zavattaro, G.: "Comparing three semantics for Linda-like languages"; Theo. Comp. Sci. 240, 1, 49-90 (2000).

[Brogi and Jacquet 98] Brogi, A., Jacquet, J.M.: "On the Expressiveness of Linda-like Concurrent Languages"; ENTCS (Elec. Notes in Comp. Sci), 16 (1998).

[Brogi et al. 02] Brogi, A., Pimentel, E., Roldán, A.M : "Compatibility of Linda-based Component Interfaces"; ENTCS (Elec. Notes in Comp. Sci), 66, 4 (2002).

[Bastide and Palanque 99] Bastide, R., Sy, O. Palanque, P.: "Formal specification and prototyping of CORBA systems (ECOOP'96) "; LNCS (Lect. Notes in Comp. Sci.), 1628, 474-494 (1999).

[Cabri et al. 00] Cabri, G., Leonardi, L., Zambonelli, F.: "MARS: A Programable Coordination Architecture for Mobile Agents', IEEE Internet Computing, 4,4, 26-35 (2000).

[Canal 01] Canal, C.: "Un Lenguaje para la Especificación y Validación de Arquitecturas de Software "; PhD thesis, Dp. Comp. Scie. Univ. of Málaga (2001).

[Canal et al. 01] Canal, C., Fuentes, L., Pimentel, E., Troya, J., Vallecillo, A.: "Extending Corba Interfaces with Protocols", The Computer Journal, 44, 5, 448–462 (2001).

[Carriero and Gelernter 89] Carriero, N., Gelernter, D.: "Linda in Context"; Comm. ACM 32, 4, 444-458 (1989).

[Han 99] Han, H.: "Semantic and usage packing for software components"; Proc. of WOI'99, 25-34 (1999).

[Lea 95] Lea, D.: "Interface-based protocol specification of open systems using PSL (ECOOP'95)"; LNCS (Lect. Notes in Comp. Sci.), 25-34, (1995).

[Leavens et al. 00] Leavens, G., Staraman, (eds.): "Foundations of Component-Based Systems "; Cambr. Univ. Press (2000).

[Magee et al. 99] Magee, J., Kramer, J., Giannakopoulou, D.: "Behaviour analysis of software architectures "; Kluwer Acad. Publ. (1999).

[Omicini and Zambonelli 98] Omicini, A., Zambonelli, F.: "TuCSoN: A Coordination Model for Mobile Agents"; J. Internet Research, 8,5, 400-413 (1998).

[Papadopoulos and Arbab 98] Papadopoulos, G., Arbab, F.: "Dynacmic Reconfiguration in Coordination Languages"; Adv. in Computer 46, 329-400 (1998).

[Shaw and Garlan 96] Shaw, M., Garlan, D.: "Software Architectures. Perspectives of an Emerging Discipline"; Prentice Hall (1996).

[Vallecillo et al. 03] Vallecillo, A., Hernández, J., Troya, J.: "Component Interoperability "; Technical Report ITI-2000-37 (2000).

[Yellin and Strom 97] Yellin, D.M., Strom, R.E.: "Protocol Specifications and Components Adaptors"; ACM Transactions on Programming Languages and Systems, 19, 1, 292-333 (1997).