Safe Composition of Linda-based Components

Ana M. Roldan [3,1]

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

Ernesto Pimentel [4,1]

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

Antonio Brogi [5,2]

Dipartimento di Informatica, Università di Pisa, Italy

Abstract

Component-Based Software Development is an emerging discipline in the field of Software Engineering. When constructing component-based systems, we must be sure that the cooperative behaviour of the components and their interaction will be successful. In this paper, we use Linda to specify the interactive behaviour of software components. To do this, we first introduce a process algebra for Linda, and then we define a compatibility relation providing conditions that ensure safe composition. This relation takes into account the state of a shared tuple space which represents the current execution. 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 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.

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

[1] The work of Ana M. Roldán and E. Pimentel has been partially supported by the Spanish project TIC2001-2705-C03-02

[2] The work of Antonio Brogi has been partially supported by the Italian MIUR project NAPOLI — Models, Calculi and Languages for Network Aware Programming.

[3] Email: amroldan@diesia.uhu.es

[4] Email: pimentel@lcc.uma.es

[5] Email: brogi@di.unipi.it

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 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 after them, like COM/DCOM, CCM, EJB, and the recent .NET.

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 [11]. Many of them are based on process algebras, and extend interfaces with a description of their concurrent behaviour [1,2,5,6,12], such as behavioural types or role-based representations.

The objective of this work is to explore the usability of the coordination language Linda for specifying the interaction behaviour of software components and to present a software architecture as a collection of interconnected computational and data components [7].

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, as well as a great expressive power to specify concurrent and distributed systems. The contributions of this paper can be summarised as follows:

(i) We uses a notion of *store sensitive compatibility* [3] to formalize the compatibility of two agents with respect to a given state of the store (share data-space).The store 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 consider that the compatibility of two agents 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.

(ii) 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. Section 3 is devoted to introduce the notion of compatibility with respect to a store. In the following section, we show software systems as architectural descriptions of their components and finally, some concluding remarks and future works are discussed.

2 A Linda Calculus

Linda was the first coordination language [9,10], 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 [4], 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.

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 [4], 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 agents by allowing parallel composition of tuples. Formally, the transition system of Table 1 refers to the extended language $\mathcal{L}'$ defined as:

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

Rule (1) states that the output operation consists of an internal move (τ_{out}) 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 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

(1)	$out(t).P \xrightarrow{\tau_{out}} \langle t \rangle \parallel P$	(6)	$\dfrac{P \xrightarrow{\alpha} P'}{P + Q \xrightarrow{\alpha} P' + Q}$
(2)	$\langle t \rangle \xrightarrow{\bar{t}} 0$	(7)	$\dfrac{P \xrightarrow{t} P' \; Q \xrightarrow{\bar{t}} Q'}{P \parallel Q \xrightarrow{\tau} P' \parallel Q'}$
(3)	$in(t).P \xrightarrow{t} P$	(8)	$\dfrac{P \xrightarrow{\underline{t}} P' \; Q \xrightarrow{\bar{t}} Q'}{P \parallel Q \xrightarrow{\tau} P' \parallel Q}$
(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)	$\dfrac{P \xrightarrow{\neg t} P' \; Q \overset{\bar{t}}{\not\longrightarrow}}{P \parallel Q \xrightarrow{\neg t} P' \parallel Q}$

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

the complementary actions t and $\bar{t}$: 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 $\bar{t}$, respectively. Notice that the process performing $\bar{t}$ 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 [4], 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. We also consider the transition system closed under the usual structural axioms for parallel or choice operators.

The rules of Table 1 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 action (τ_{out}) and another one for synchronization (τ). Formally, this corresponds to introducing 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 $\mathcal{L}$ 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 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:

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

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

3 Safe Composition of Components

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 consider a compatibility relation that takes into account the situation of the store. 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. Before defining the notion of compatibility with respect to a store, we introduce the notion of *synchronizable processes.* After that, we show the definition of compatibility with respect to a store, which was introduced in [3].

Definition 3.1 [Synchronizable processes] A process P provides an input a for an agent Q if there exist two processes P' and Q', such that $P \overset{\overline{a}}{\Longrightarrow} P'$ and $Q \overset{\alpha}{\Longrightarrow} Q'$, 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 3.2 [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 $Q \longmapsto^{*} \overset{\overline{t}}{\not\longrightarrow}$ 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 $P \xrightarrow{t} P'$ and $St \xrightarrow{\overline{t}} St'$ then $P'\ \mathcal{C}_{St'}\ Q$.

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

(vi) If $P \xrightarrow{\underline{t}} P'$ and $St \xrightarrow{\overline{t}}$ 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 $P\mathcal{C}_{St}Q$.

When agents 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 agents. Obviously, depending on the structural complexity of the agents, the cost of checking might be very high.

Proposition 3.3 *If $P \diamond_{St} Q$ then $P \parallel Q \parallel St$ is successful.*

Proof. The full proof is in [3]. $\square$

In this context, the notion of compatibility allows to:

- check the compatibility of a component and a running system w.r.t. the current store (characterizing the current state of the execution),
- condition 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.

Thus, a negative answer showing the non-compatibility of two components could prevent from wrong compositions. Obviously, the compatibility of two generic agents is not always decidible. Although this 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 what is the store (if any) that makes two given agents compatible.

4 Linda-based 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 [7]. 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 partial interfaces or *roles* for describing the behaviour of each component and we explicitly represent system architecture as a set of roles for each component.

Definition 4.1 [Component] 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 4.2 [Attachment] 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 attachments.

Definition 4.3 [Context] 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 $\mathcal{S}$, $|\psi(St) \bigcap Comp_i| \leq 1$.

(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 4.4 [Successful context] 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. In the following proposition, we formalized the idea that permit us to get this objective. This result could be extended, and so, a *context* is *successful* if the set of roles that describe the components can be separate in small sets of roles (composed by two roles)where roles are compatible w.r.t a store. Although, this can be complex and difficult to automatically check.

Proposition 4.5 *Given a context* $< \mathcal{C}, \mathcal{S}, \psi >$, *if* $|\psi(St)| = 2$ *for all St in* $\mathcal{S}$ *and the two roles in* $\psi(St)$ *are compatible w.r.t St, then the context is successful.*

Proof. The proof descends from the disjoint distribution of the roles among stores (def.4.3), where roles are compatible w.r.t. a store (def.3.2). $\square$

The previous proposition can be applied only when the architecture is described as a set of components connected through roles, and when these connections are established pairwise.

When a new component is inserted into a running system, the resulting context may exhibit an erroneous behaviour (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 behaviours. 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.

To illustrate the use of Linda for specifying component protocols, we now show a partial description of the interaction between the auctioneer and a bidder.

The auctioneer may start a session by adding to the store a tuple of the form ⟨"on_sale", goodOnSale, initialPrice⟩. The auctioneer then inputs (and consumes) the bids that are possibly made by bidder(s). If the bid received is higher than the highest value offered so far for the good, then the auctioneer updates the current price of the good. Otherwise the auctioneer continues to analyse the bids received. When no bid has been made, the auctioneer closes the auction session, publishes the final result of the session on the store, and terminates.

```
Auctioneer.Bid(good,price) =
    in("auction_closed",_,_,_).StartSession(good,price)
    +
    nrd("auction_closed",_,_,_).StartSession(good,price)

StartSession(good,price) =
   out("on_sale",good,price). AuctionSession(good,price,"nobody")

AuctioneerSession(good,price,currentWinner) =
    in("bid",idBidder,good,offer).
    (
     in("on_sale",_,_).out("on_sale",good,offer).
     AuctioneerSession(good,offer,idBidder)
     +
     AuctioneerSession(good,price,currentWinner)
    )
   +
   nrd("bid",_,_,_).in("on_sale",_,_).
   out("auction_closed",good,price,currentWinner).0
```

A bidder instead waits for the start of an auction session. When a new session is opened, the bidder decides whether to participate in the session (by putting his bid in the store) or to simply wait for its end.

```
Bidder.Auc(id) =
   rd("on_sale",good,price).
   (
    out("bid",id,good,offer).BidderWait(id)
    +
    BidderWait(id)
   )

BidderWait(id) =
   rd("auction_closed",good,price,winner).0
```

5 Concluding Remarks

Linda is a coordination language where inter-process communication can only occur through a set of tuples. The main novelty of our proposal consists of defining a software architecture taking into account the specifications of its components. 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.

Certainly, some of the issues covered in this paper also have 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, and in [8] 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 [5]. Some ideas proposed in [5] already have been applied to CORBA in [6].

Our proposal somehow combines these two last lines by defining a notion of process compatibility in the style of [5,6], 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 [7].

Our future work will be devoted to define an *inheritance* relation over agents in order to promote the reusability and substitutability of interaction descriptions, to study how this affects compatibility and successful computations, and 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, E. Pimentel, and A. Roldán. Compatibility of Linda-based Component Interfaces, ICALP'02(FMCI). Electronic Notes in Theoretical Computer Science, 2002.

[4] N. Busi, R. Gorrieri, and G. Zavattaro. *A process algebraic view of linda coordination primitives.* Electronic Theoretical Computer Science, 192:167–199, 1998.

[5] 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.

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

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

[8] 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.*

[9] D. Gelernter. *Generative Communication in Linda.. ACM Transactions on Programming Languages and Systems , 7:1 (1985), pp.80-112.*

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

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

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