Towards a calculus for generative communication

P. Ciancarini, R. Gorrieri, G. Zavattaro
Department of Computer Science, University of Bologna
Piazza di Porta S. Donato 5, I-40127 Bologna, Italy
Telephone: +39 51 354516 Fax: +39 51 354510
email: `cianca,gorrieri,zavattar@cs.unibo.it`

Abstract

We introduce a theory for generatively communicating concurrent processes. Generative communication is an asynchronous interprocess communication mechanism based on a shared data structure; information items can be introduced into, read or withdrawn from such a data structure by parallel processes. The most representative language based on such a paradigm is the coordination language Linda. Our idea is to embed generative communication in a process algebra like CCS. The advantage of having a process algebraic framework is that formal techniques developed in the process algebra area can be easily adapted to the field of generative communication. We investigate three standard observational equivalences (bisimulation, failure, and trace) and we observe that the failure semantics is the most appropriate to model the features of generative communication.

Keywords

Coordination languages, generative communication, process algebras, observational equivalences

1 INTRODUCTION

Asynchronous communication realized by means of the insertion, reading, and withdrawal of elements to and from a shared multiset, is the peculiar feature of a family of coordination languages (Gelernter and Carriero, 1992). This communication mechanism is referred to as *generative communication*, and it was introduced for the first time in the coordination language Linda (Gelernter, 1985). Linda provides interprocess communication via a medium called *Tuple Space* (TS for short), that is a shared memory which contains a set of messages that are produced by a set of processes. TS is accessible by every parallel process by means of three primitives:

`out(Message)`: produces a message;
`read(Message)`: reads (without consuming) a message;
`in(Message)`: reads and consumes a message.

The peculiar features of generative communication can be listed as follows:

- A process can always insert a message in TS performing an `out` operation.
- A process can perform an `in` or a `read` operation only if the required message is in TS; if not, it blocks. A side effect of the execution of an `in` operation is the withdrawal of the read message.
- The insertion order of messages in TS has no influence on their reading order.
- Multiple occurrences of the same message can be in TS at the same time (TS is a multiset of messages).

This communication mechanism is said to be generative because a message generated by a process has an independent existence in TS until it is explicitly withdrawn. In fact, after its insertion in TS, a message becomes equally accessible to all processes, and it is bound to none.

Generative communication is provided not only by Linda, but also by other languages such as Shared Prolog (Brogi and Ciancarini, 1991) and Bauhaus-Linda (Carriero, Gelernter, and Zuck, 1995). The main difference among these languages is the type of messages in the shared memory: Linda uses ordered tuples, Shared Prolog logic terms, and Bauhaus-Linda unordered multisets. For the sake of generality and simplicity, messages will be treated as atomic items in the remainder of the paper and each of them will be referred to by means of an identification name.

Our aim is to introduce a framework to reason formally about generative communication. Our idea is to embed such a communication mechanism in a process algebra. In this way, all the formal techniques for analyzing concurrent systems used in the process algebra area will be easily adapted to the field of generative communication. Standard process algebras like CCS (Milner, 1989), CSP (Hoare, 1978), and ACP (Bergstra and Klop, 1986) provide a synchronous handshake communication mechanism. On the other hand, generativeness is based on asynchronous communication. Hence, in order to embed generative communication in a process algebra, the crucial problem is the way the communication medium is represented. A first trivial proposal can be to provide an extra agent $TS\langle M\rangle$ which is able to receive messages from the senders, to store them as the multiset M, and to give the required messages to the receivers. This means that we are implementing asynchronous communication by means of synchronous communication using an auxiliary communication manager. However, some problems arise if the communication medium is represented by one single agent:

- Since the agent $TS\langle M\rangle$ has the full responsibility of managing the interprocess communication, its design may become too complex and it may become an execution bottleneck.
- A unique centralized store for messages may not exist. For example, in distributed Linda implementations (Carriero and Gelernter, 1986) the messages are usually distributed too.

As far as the solution of these problems is concerned, we propose to assume the communication manager as *fully distributed*. To be more precise, we propose to consider each message as an active entity able to give its contents to every potential reader. In this

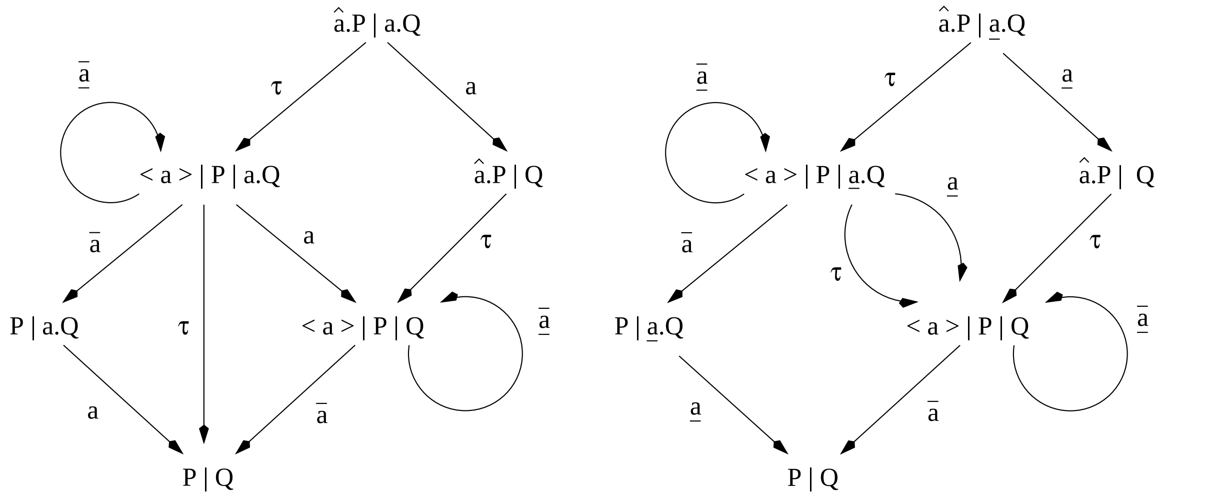

Figure 1 Generatively communicating processes.

way, there is no explicit centralized communication manager because all the messages are treated as autonomous agents. The basic features of our proposal are the following:

- The autonomous agent $\langle a \rangle$ is introduced to represent the sent message a.
- The prefix $\hat{a}$ denotes a message which can be sent. The execution of $\hat{a}$ consists of the addition of the agent $\langle a \rangle$ to the environment:
  $\hat{a}.P \xrightarrow{\tau} \langle a \rangle | P$
  The label is τ because this step, representing an `out` operation, is a local autonomous step of computation which does not depend on the environment.
- An extra prefix $\underline{a}$ is introduced to represent the request of reading message a without consuming it.
- The agent $\langle a \rangle$ can be consumed by an agent which performs an `in` operation:
  $\langle a \rangle \xrightarrow{\overline{a}} \underline{0}$
  and it can be read by an agent which performs a `read` operation:
  $\langle a \rangle \xrightarrow{\overline{\underline{a}}} \langle a \rangle$
  The labels $\overline{a}$ and $\overline{\underline{a}}$ represents the "complementary" actions for a and $\underline{a}$ respectively.

Figure 1 compares two different cases of generative communication between processes: in the first graph a reader which performs an `in` operation is considered, while the second graph describes the case of a `read` operation. The most important difference between the two cases is due to the behavior of the synchronization (i.e. the simultaneous execution of the complementary actions a, $\overline{a}$ or $\underline{a}$, $\overline{\underline{a}}$). In fact, if the reader performs an `in` operation, the message is withdrawn, while if it executes a `read`, the agent $\langle a \rangle$ is not removed.

The paper is organized as follows. In Section 2 we formally define the syntax and the operational semantics of the language. In Section 3 some examples of concurrent systems are presented in order to highlight the peculiar features of the language. In Section 4 we analyze three observational semantics for our language: bisimulation, failure and trace. We observe that failure is the most appropriate: in fact, bisimulation is not abstract enough to describe all the features of generative communication in which we are interested while trace is too coarse. In Section 5 we report some conclusive remarks: we analyze the originality

Table 1 Syntax

E	$::=$	$\underline{0}$	null agent
	$\mid$	$\langle a \rangle$	message agent
	$\mid$	$p.E$	*prefix* operator
	$\mid$	$E \mid E$	*parallel* operator
	$\mid$	$E + E$	*choice* operator
	$\mid$	$E \setminus L$	*restriction* operator
	$\mid$	$E[f]$	*relabeling* operator
	$\mid$	x	agent variable
	$\mid$	$rec\ x.E$	*recursion* operator

where: $L \subseteq Message$

$f : Label \longrightarrow Label$

$$\text{such that:} \begin{cases} f(\overline{\gamma}) & = & \overline{f(\gamma)} \\ f(\underline{\gamma}) & = & \underline{f(\gamma)} \\ f(\tau) & = & \tau \end{cases}$$

of our language with respect to possible representations of the generative communication mechanism made in standard CCS and we compare our framework with other proposals for generativeness and for embedding asynchronous communication in process algebras. The proofs of propositions are reported in the full paper (Ciancarini, Gorrieri, and Zavattaro, 1995).

2 THE LANGUAGE AND ITS SEMANTICS

Let:

- $Message$, ranged over by a, b, etc., be the set of possible messages;
- $Prefix = \{a, \underline{a}, \hat{a} \mid a \in Message\} \cup \{\tau\}$, ranged over by p, q, etc., be the set of prefixes;
- $Label = \{a, \underline{a}, \overline{a}, \underline{\overline{a}} \mid a \in Message\} \cup \{\tau\}$, ranged over by α, β, etc., be the set of labels;
- $Obs = Label \setminus \{\tau\}$, ranged over by γ, η, etc., be the set of visible labels;
- $\overline{\cdot}$, $\underline{\cdot}$, $\underline{\overline{\cdot}}$: $Obs \xleftrightarrow{1-1} Obs$ be three bijections such that $\overline{(a)} = \overline{a}$, $\overline{(\overline{a})} = a$, $\overline{(\underline{a})} = \underline{\overline{a}}$, $\overline{(\underline{\overline{a}})} = \underline{a}$, $\underline{(a)} = \underline{a}$, $\underline{(\overline{a})} = \underline{\overline{a}}$, $\underline{(\underline{a})} = a$, $\underline{(\underline{\overline{a}})} = \overline{a}$, and $\underline{\overline{(\gamma)}} = \underline{(\overline{(\gamma)})}$;
- $\mathcal{X}$, ranged over by x, y, etc., be the set of agent variables.

The agent expressions, ranged over by E, F, etc., are defined in Table 1. The null agent $\underline{0}$ and the class of terms $\langle a \rangle$ represent the possible elementary agents. The agent $\underline{0}$ is deadlocked (i.e. it is not able to perform any kind of action) whereas $\langle a \rangle$ represents a message a ready to be read or withdrawn.

The *prefix operator* is used to define the possible actions executed by the agents: there are four actions depending on the kind of prefixes. The first and the second one, i.e. a and $\underline{a}$, correspond to the operation `in` and `read`, representing the request of the withdrawal and the reading of the message a, respectively. The third prefix $\hat{a}$ represents the `out` operation which causes the addition of $\langle a \rangle$ to the environment, and the last one is the

Table 2 Operational semantics

$a.P \xrightarrow{a} P$	$\dfrac{P \xrightarrow{\alpha} P'}{P+Q \xrightarrow{\alpha} P' \quad \& \quad Q+P \xrightarrow{\alpha} P'}$
$\underline{a}.P \xrightarrow{\underline{a}} P$	$\dfrac{P \xrightarrow{\gamma} P' \quad \gamma, \overline{\gamma}, \underline{\gamma}, \underline{\overline{\gamma}} \notin L}{P \setminus L \xrightarrow{\gamma} P' \setminus L}$
$\tau.P \xrightarrow{\tau} P$	$\dfrac{P \xrightarrow{\tau} P'}{P \setminus L \xrightarrow{\tau} P' \setminus L}$
$\hat{a}.P \xrightarrow{\tau} \langle a \rangle \vert P$	$\dfrac{P \xrightarrow{\alpha} P'}{P[f] \xrightarrow{f(\alpha)} P'[f]}$
$\langle a \rangle \xrightarrow{\overline{a}} \underline{0}$	$\dfrac{P[rec\, x.P/x] \xrightarrow{\alpha} P'}{rec\, x.P \xrightarrow{\alpha} P'}$
$\langle a \rangle \xrightarrow{\underline{\overline{a}}} \langle a \rangle$	
$\dfrac{P \xrightarrow{\alpha} P'}{P \vert Q \xrightarrow{\alpha} P' \vert Q \quad \& \quad Q \vert P \xrightarrow{\alpha} Q \vert P'}$	
$\dfrac{P \xrightarrow{\gamma} P' \quad Q \xrightarrow{\overline{\gamma}} Q'}{P \vert Q \xrightarrow{\tau} P' \vert Q'}$	

invisible prefix τ which stands for local autonomous steps of computation. The meaning of the other operators is the usual one. The *parallel operator* is used to combine agents which are able to perform actions in parallel and to synchronize on complementary actions. The *choice* operator is used to represent a non-deterministic alternative choice between two combined agents. The *restriction* operator is used to define local actions. The *relabeling* operator allows dynamic changes of the name of the messages. The *recursion* operator is used for the definition of recursive agents.

We say that x is bound in E if each occurrence of x is within some subexpression $rec\ x.F$; x is also guarded if each occurrence in F is within some subexpression $p.F'$. We say that E is closed and guarded if only bound and guarded variables occur in it. Let *Agent*, ranged over by P, Q, etc., be the set of closed and guarded terms.

The operational semantics of our language is described by a *labeled transition system* $(Agent, Label, \longrightarrow)$. The labeled relation $\longrightarrow \subseteq (Agent \times Label \times Agent)$ is the smallest one which satisfies the axioms and rules of Table 2.

3 EXAMPLES

3.1 Dining philosophers

The classical problem of dining philosophers can be represented in our language giving to the message agents $\langle a \rangle$ the meaning of objects. The message agents can be used to represent the forks on the table: a fork can be grabbed or freed by a philosophers like a message can be sent or withdrawn by an agent of our language. Hence the fork f_i is modeled by the agent $\langle f_i \rangle$.

Table 3 introduces two different representations for the problem of dining philosophers. In the first representation (i.e. agent $\mathbf{DinPhi_1}$) each philosopher $\mathbf{P_i}$ grabs the forks in a fixed order: first he grabs the fork on its right (fork f_i) and after he takes the one on

Table 3 Dining philosophers specification

$\mathbf{DinPhi_1}$	$\stackrel{def}{=}$	$\langle f_0\rangle \vert \langle f_1\rangle \vert \dots \vert \langle f_{n-1}\rangle \vert \mathbf{P_0} \vert \mathbf{P_1} \vert \dots \vert \mathbf{P_{n-1}}$
$\mathbf{DinPhi_2}$	$\stackrel{def}{=}$	$\langle f_0\rangle \vert \langle f_1\rangle \vert \dots \vert \langle f_{n-1}\rangle \vert \mathbf{P_0} \vert \mathbf{P_1} \vert \dots \vert \mathbf{P_{n-1}}[f_{n-1}/f_0, f_0/f_{n-1}]$
$\mathbf{P_i}$	$\stackrel{def}{=}$	$rec\ x.(f_i.f_{i+_n1}.\widehat{f_{i+_n1}}.\widehat{f_i}.x)$
where $+_n$ stands for the sum modulo n		

Table 4 Drink dispenser specifications

Dispenser	$\stackrel{def}{=}$	$rec\ x.(switch.rec\ y.(request.\widehat{drink}.y + switch.x) + request.x)$
DistrDisp	$\stackrel{def}{=}$	$(\langle off\rangle \vert \langle ok\rangle \vert \mathbf{SwButton} \vert \mathbf{ReqButton}) \setminus \{on, off, ok\}$
SwButton	$\stackrel{def}{=}$	$rec\ x.(ok.(switch.(on.\widehat{off}.\widehat{ok}.x + off.\widehat{on}.\widehat{ok}.x) + \widehat{ok}.x))$
ReqButton	$\stackrel{def}{=}$	$rec\ y.(ok.(request.(\underline{on}.\widehat{drink}.\widehat{ok}.y + \underline{off}.\widehat{ok}.y) + \widehat{ok}.y))$

its left (fork f_{i+_n1}). In this way the system can give rise to deadlock. In fact, if all the philosophers grab their first fork at the same time, the system will be no more able to proceed. In the second representation (i.e. agent $\mathbf{DinPhi_2}$) the possibility of deadlock is removed by changing the forks grabbing order of philosopher $\mathbf{P_{n-1}}$. It must be observed that the grabbing order has been changed only by using the relabeling operator without altering the specification of philosopher $\mathbf{P_{n-1}}$.

3.2 Drink dispenser

In Table 4 an automatic drink dispenser with two different buttons (*switch* and *request*) is specified. Button *switch* is used to turn on or off the dispenser. Button *request* is used to ask for a drink: if the dispenser is turned on, the drink is returned; instead, if it is turned off, the button *request* is not enabled.

Two possible specifications for the dispenser are presented: **Dispenser** and **DistrDisp**. The first specification supposes the existence of one centralized manager for both the buttons, while the second consider two separated managers, i.e., **SwButton** and **ReqButton** which manage the buttons *switch* and *request* respectively. In the distributed version, three objects are used in order to allow the communication between the two managers: $\langle on\rangle$, $\langle off\rangle$ and $\langle ok\rangle$. The objects $\langle on\rangle$ and $\langle off\rangle$ are used to represent the state of the dispenser (turned on and turned off respectively), while $\langle ok\rangle$ is used to prevent the possibility to have both the manager enabled (a manager is active only if it holds the object $\langle ok\rangle$). The manager of button *switch* only replaces the message agents $\langle on\rangle$ or $\langle off\rangle$ for $\langle off\rangle$ or $\langle on\rangle$, respectively. Instead, the manager of button *request* must test if the dispenser is turned on before returning the drink. The state of the dispenser can be tested by **ReqButton** simply by reading (without removing) the message *on* or *off*: if message *on* is read, then the drink is returned, otherwise nothing is done. The two agents **Dispenser** and **DistrDisp** have different operational semantics, but they can be considered equivalent, because they are able to reply in the same way to every possible users. This equivalence is formalized in the following section where observational seman-

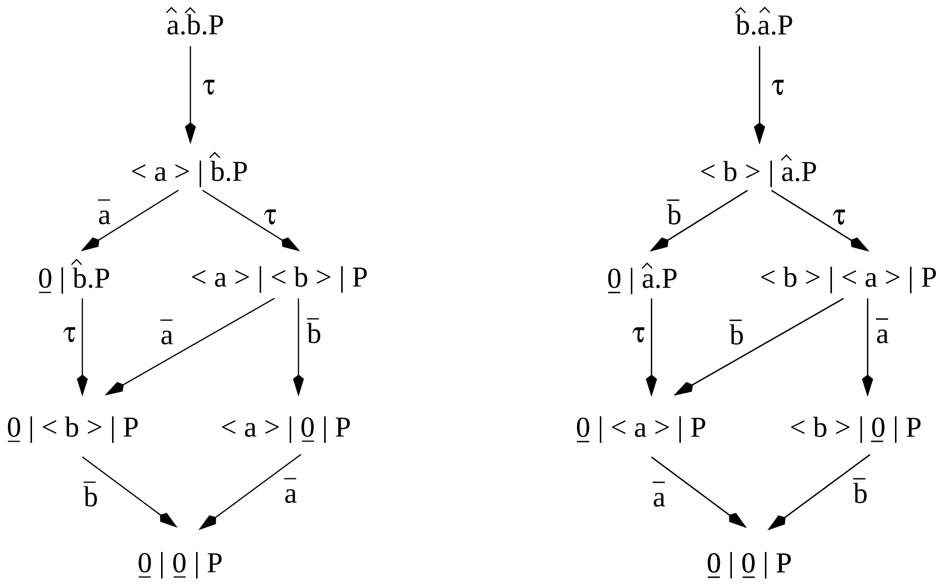

Figure 2 Example of semantically equivalent agents.

tics are defined. For example, the *bisimulation equivalence* that we are going to adapt to our language, equates the agents **Dispenser** and **DistrDisp**.

4 OBSERVATIONAL SEMANTICS

We investigate several standard semantic equivalences (bisimulation, failure, and trace) in order to describe formally a significant class of expected properties of the generative communication paradigm. For example, the agents in Figure 2, i.e. $\hat{a}.\hat{b}.P$ and $\hat{b}.\hat{a}.P$, have different operational semantics, but they can be considered semantically equivalent because, as already stated, the insertion order of the messages in TS is not a determinating factor. For the sake of simplicity and in order to define finite equational proof systems for the congruences that we are going to introduce, only finite agents (i.e. *recursion* free agents) are considered in this section.

4.1 Bisimulation model

We investigate if the standard semantic equivalence used for CCS, i.e. bisimulation (Milner, 1989), is abstract enough to identify the agents in Figure 2. In order to treat the τ labeled transition steps as unobservable actions, we consider the standard *weak* bisimulation.

Definition 41 $P \stackrel{\alpha}{\Longrightarrow} Q \quad \stackrel{def}{\Longleftrightarrow} \quad P = R_1 \stackrel{\tau}{\longrightarrow} \ldots \stackrel{\tau}{\longrightarrow} R_n \stackrel{\alpha}{\longrightarrow} S_1 \stackrel{\tau}{\longrightarrow} \ldots \stackrel{\tau}{\longrightarrow} S_m = Q$ where $n, m \geq 1$.

Definition 42 $P \stackrel{\alpha}{\models\!\!\Longrightarrow} Q \quad \stackrel{def}{\Longleftrightarrow} \quad (P \stackrel{\alpha}{\Longrightarrow} Q) \ \vee \ (P = Q \wedge \alpha = \tau)$

Definition 43 A relation $\mathcal{R} \subseteq Agent \times Agent$ is a bisimulation if it satisfies the following condition:

$$\begin{array}{l} \text{if } \ P\,\mathcal{R}\,Q \ \text{ then } \ \forall \alpha \in Label: \\ \quad (i) \ \text{ if } \ P \xrightarrow{\alpha} P' \ \text{ then } \ \exists Q': \ Q \models\!\!\!\xRightarrow{\alpha} Q' \ \wedge \ P'\,\mathcal{R}\,Q' \\ \quad (ii) \ \text{ if } \ Q \xrightarrow{\alpha} Q' \ \text{ then } \ \exists P': \ P \models\!\!\!\xRightarrow{\alpha} P' \ \wedge \ P'\,\mathcal{R}\,Q' \end{array}$$

Definition 44 (*Bisimulation equivalence* $\approx$)

$$\approx \ = \ \bigcup \{ \ \mathcal{R} \subseteq Agent \times Agent \mid \mathcal{R} \text{ is a bisimulation} \}$$

The bisimulation equivalence identifies, for example, the following agents:

$\hat{a}.\hat{b}.P \approx \hat{b}.\hat{a}.P$

$\hat{a}.b.P \approx \hat{a}.b.P + b.\hat{a}.P$

$\hat{a}.\underline{b}.P \approx \hat{a}.\underline{b}.P + \underline{b}.\hat{a}.P$

However, the bisimulation equivalence $\approx$ is not a congruence. In fact:

$\hat{a}.P \approx \langle a \rangle | P$

but:

$(\hat{a}.P) + \langle b \rangle \not\approx (\langle a \rangle | P) + \langle b \rangle$

As for CCS, the fully abstract semantics w.r.t. the bisimulation model is the observational equality.

Definition 45 (*Observational equality* $\simeq$)

$$\begin{array}{l} P \simeq Q \ \stackrel{def}{\Longleftrightarrow} \ \forall \alpha \in Label: \\ \quad (i) \ \text{ if } \ P \xrightarrow{\alpha} P' \ \text{ then } \ \exists Q': \ Q \xRightarrow{\alpha} Q' \ \wedge \ P' \approx Q' \\ \quad (ii) \ \text{ if } \ Q \xrightarrow{\alpha} Q' \ \text{ then } \ \exists P': \ P \xRightarrow{\alpha} P' \ \wedge \ P' \approx Q' \end{array}$$

Proposition 46 (*Full abstractness of* $\simeq$ *w.r.t.* $\approx$)

$P \simeq Q$ iff $\forall \mathbf{C}[\,] : \mathbf{C}[P] \approx \mathbf{C}[Q]$

where $\mathbf{C}[\,]$ ranges over all possible contexts.

In the first part of the proof of the last proposition (Ciancarini, Gorrieri, and Zavattaro, 1995), the observational equality is proved to be a congruence. This allows to investigate about an equational proof system for $\simeq$. Our axiomatization uses an auxiliary prefix operator.

Definition 47 (*Multiple read prefix* $a^\star$)

$a^\star.P \xrightarrow{\overline{a}} a^\star.\underline{0}|P$

$a^\star.P \xrightarrow{\overline{a}} P$

In order to extend the syntax of our language introducing the multiple read prefix, the set $Prefix$ must be redefined: $Prefix = \{a, \underline{a}, \hat{a}, a^\star \mid a \in Message\} \cup \{\tau\}$. This auxiliary prefix is used in our axiomatization in order to transform the class of message agents in equivalent prefix forms: it is easy to prove that $\langle a \rangle \simeq a^\star.\underline{0}$.

In Table 5 an axiomatic characterization for the observational equality is presented. Axioms (1) and (2) state that the *choice* composition operator $+$ is commutative and associative. Axioms (3) and (4) consist of the other standard axioms for the alternative

Table 5 Axioms for observational equality

(1)	$P + Q$	$=$	$Q + P$
(2)	$P + (Q + R)$	$=$	$(P + Q) + R$
(3)	$P + \underline{0}$	$=$	P
(4)	$P + P$	$=$	P
(5)	$\hat{a}.P$	$=$	$\tau.(\langle a \rangle \vert P)$
(6)	$\langle a \rangle$	$=$	$a^\star.\underline{0}$
(7)	$P \vert Q$	$=$	$\sum_i p_i.(P_i \vert Q) + \sum_j q_j.(P \vert Q_j)+$ $\sum_{ij:((p_i = a \wedge q_j = a^\star) \vee (p_i = a^\star \wedge q_j = a))} \tau.(P_i \vert Q_j)+$ $\sum_{ij:(p_i = a^\star \wedge q_j = \underline{a})} \tau.((a^\star.\underline{0} \vert P_i) \vert Q_j)+$ $\sum_{ij:(p_i = \underline{a} \wedge q_j = a^\star)} \tau.(P_i \vert (a^\star.\underline{0} \vert Q_j))$ if $P = \sum_i p_i.P_i \wedge Q = \sum_j q_j.Q_j \wedge p_i, q_j \neq \hat{a}$
(8)	$(p.P) \setminus L$	$=$	$\begin{cases} p.(P \setminus L) & \text{if } (p = \tau) \vee (p = a \text{ (or } \underline{a},\ a^\star) \wedge a \notin L) \\ \underline{0} & \text{if } p = a \text{ (or } \underline{a},\ a^\star) \wedge a \in L \end{cases}$
(9)	$(P + Q) \setminus L$	$=$	$(P \setminus L) + (Q \setminus L)$
(10)	$\underline{0} \setminus L$	$=$	$\underline{0}$
(11)	$(p.P)[f]$	$=$	$\begin{cases} (f(a))^\star.(P[f]) & \text{if } p = a^\star \\ f(p).(P[f]) & \text{if } p = \tau \vee p = a \text{ (or } \underline{a}) \end{cases}$
(12)	$(P + Q)[f]$	$=$	$(P[f]) + (Q[f])$
(13)	$\underline{0}[f]$	$=$	$\underline{0}$
(14)	$p.\tau.P$	$=$	$p.P$
(15)	$P + \tau.P$	$=$	$\tau.P$
(16)	$p.(P + \tau.Q)$	$=$	$p.(P + \tau.Q) + p.Q$

choice operator. Axiom (5) is used to describe the `out` prefix operator in terms of other prefixes. Axiom (6) corresponds to the property of the multiple read prefix introduced above. Axiom (7) is the adaptation of the well known expansion theorem to our formalism. Axioms from (8) to (13) are the usual axioms for the *restriction* and *relabeling* operators. Finally, axioms from (14) to (16) are the standard τ laws for bisimulation.

In the remainder of the paper $\mathcal{A} \vdash P = Q$ means that P is proved equal to Q via standard equational reasoning with the set of axioms $\mathcal{A}$.

Proposition 48 (*Soundness and completeness of* $\mathcal{A}$)
$P \simeq Q$ iff $\mathcal{A} \vdash P = Q$

Table 6 Axioms for failure equivalence

(17)	$P + \tau.Q$	$=$	$\tau.(P + Q) + \tau.Q$
(18)	$p.P + p.Q$	$=$	$p.(\tau.P + \tau.Q)$
(19)	$\tau.(p.P + Q) + \tau.(p.R + S)$	$=$	$\tau.(p.P + p.R + Q) + \tau.(p.P + p.R + S)$

The following example, which was inspired by (De Boer and Palamidessi, 1990), describes a further property of the generative communication mechanism that the bisimulation semantic does not capture.

Example 49 Agent $P_1 = \hat{a}.(\hat{b}.P + \hat{c}.P)$ should be considered equivalent to $P_2 = \hat{a}.\hat{b}.P + \hat{a}.\hat{c}.P$. In fact, the choice between `out` operations does not depend on the environment: it is a local choice which is not influenced by external agents. In this way, the agent P_1, similarly to P_2, inserts in TS the message a and another one between b and c wherever the choice between b and c is completely internal. Bisimulation semantics is not abstract enough, in fact $P_1 \not\approx P_2$. Thus, we are forced to look for a more suitable equivalence.

4.2 Failure model

The failure semantic model consists in observing all the pairs $[s, X]$ obtained by associating to every trace s (where $s \in Obs^*$), a set X of actions which cannot be performed after the execution of s.

Definition 410 $P \stackrel{s}{\Longrightarrow} Q \quad \stackrel{def}{\Longleftrightarrow}$

- $Q = P \ \wedge \ s = \varepsilon$ or
- $P \stackrel{\gamma}{\longrightarrow} P' \stackrel{s'}{\Longrightarrow} Q \ \wedge \ s = \gamma s'$ or
- $P \stackrel{\tau}{\longrightarrow} P' \stackrel{s}{\Longrightarrow} Q$

where ε represents the empty string.

Definition 411 (*Failure set*)
$F[\![P]\!] \ = \ \{[s, X] \mid s \in Obs^*, X \subseteq Label, \ P \stackrel{s}{\Longrightarrow} P', P' \stackrel{\tau}{\not\longrightarrow}, \forall \alpha \in X : P' \stackrel{\alpha}{\not\longrightarrow}\}$
where $P \stackrel{\alpha}{\not\longrightarrow}$ means that there is no P' such that $P \stackrel{\alpha}{\longrightarrow} P'$.

Definition 412 (*Failure equivalence* $\approx_F$)
$P \approx_F Q \ \stackrel{def}{\Longleftrightarrow} \ F[\![P]\!] = F[\![Q]\!]$

The failure equivalence is not a congruence. In fact:
$\tau.\underline{0} \approx_F \underline{0}$
but:
$a.\underline{0} + \tau.\underline{0} \not\approx_F a.\underline{0} + \underline{0}$
We introduce an equivalence which is proved (Ciancarini, Gorrieri, and Zavattaro, 1995) to be the fully abstract semantics w.r.t. the failure equivalence.

Definition 413 (*Failure congruence* $\simeq_F$)
$P \simeq_F Q \overset{def}{\Longleftrightarrow} (P \approx_F Q) \wedge (P \stackrel{\tau}{\not\longrightarrow} \text{ iff } Q \stackrel{\tau}{\not\longrightarrow})$

Proposition 414 (*Full abstractness of* $\simeq_F$ *w.r.t.* $\approx_F$)
$P \simeq_F Q$ iff $\forall \mathbf{C}[\,] : \mathbf{C}[P] \approx_F \mathbf{C}[Q]$

All the agents identified by the observational equality are equated by the failure congruence too (i.e. $\simeq \subseteq \simeq_F$). In other words the failure semantics is more abstract than bisimulation. The inverse relation is not true: the agents introduced in Example 49, which are not identified by bisimulation, are failure congruent:
$\hat{a}.(\hat{b}.P + \hat{c}.P) \simeq_F \hat{a}.\hat{b}.P + \hat{a}.\hat{c}.P$
Proposition 414 also states that $\simeq_F$ is a congruence. As for the observational equality, this allows to investigate about an axiomatic characterization for $\simeq_F$. The one that we propose is the adaptation to a standard equational proof system of the axiomatization for failure equivalence on synchronization trees (Brookes, 1983). The property $\simeq \subseteq \simeq_F$ allows to state that the set of axioms $\mathcal{A}$ introduced for the observational equality is sound for the failure congruence too. In Table 6 the characteristic axioms concerning the failure equivalence are introduced. The sound and complete axiomatization is defined as follows:
$\mathcal{B} \;=\; \mathcal{A} \setminus \{(14), (15), (16)\} \cup \{(17), (18), (19)\}$
It must be observed that the τ laws (14), (15), and (16) are not required even if they are sound for the failure congruence too.

Proposition 415 (*Soundness and completeness of* $\mathcal{B}$)
$P \simeq_F Q$ iff $\mathcal{B} \vdash P = Q$

4.3 Trace model

Definition 416 (*Trace set*)
$T[\![P]\!] \;=\; \{s \in Obs^{\star} \mid \exists P' : P \stackrel{s}{\Longrightarrow} P' \}$

Definition 417 (*Trace equivalence* $\approx_T$)
$P \approx_T Q \overset{def}{\Longleftrightarrow} T[\![P]\!] = T[\![Q]\!]$

It is easy to see that the trace semantics is more abstract than the failure, i.e., $\approx_F \subseteq \approx_T$. In fact, the trace set can be also defined in the following way:
$T[\![P]\!] \;=\; \{s \in Obs^{\star} \mid \exists X \subseteq Label : \; [s, X] \in F[\![P]\!]\}$
As for CCS, the trace semantics is too abstract because it does not distinguish deadlock. In fact, the trace equivalent agents:
$a.(b.\underline{0} + c.\underline{0}) \approx_T a.b.\underline{0} + a.c.\underline{0}$
show a different deadlock behavior, e.g., when composed in parallel with the process $\hat{a}.\hat{b}.\underline{0}$.

5 CONCLUSION AND RELATED WORK

In this paper we have introduced a process algebraic framework which can be used to reason formally about generative communication. Our language extends CCS by introducing the message agents $\langle a \rangle$, the `read` prefix $\underline{a}$ and the `out` prefix $\hat{a}$ (which is used instead of the standard prefix $\overline{a}$). It can be thought that these extensions are only *syntactic sugar*, as $\langle a \rangle$, $\hat{a}$ and $\underline{a}$ could in principle be mapped to standard CCS by a translation like the following (suggested by an anonymous referee):

$$\begin{array}{lcl} [\![\langle a \rangle]\!] & = & \overline{a}.\underline{0} \\ [\![\hat{a}.P]\!] & = & \tau.([\![\langle a \rangle]\!]|P) \\ [\![\underline{a}.P]\!] & = & a.[\![\hat{a}.P]\!] \end{array}$$

This mapping models the reading of a message by means of the withdrawal and the consequent emission of such a message. This approach is not acceptable because there is an instant (between the withdrawal and the emission of the message) in which the state is not consistent; a process willing to read the message may not find it! Such as wrong situation cannot happen in our approach, where the reading of a message cannot prevent other agents to read it. This problem is solved by the following more accurate mapping which uses, however, one extra set of prefixes:

$$\begin{array}{lcl} [\![\langle a \rangle]\!] & = & rec\ x.(\overline{a'}.x + \overline{a}.\underline{0}) \\ [\![\hat{a}.P]\!] & = & \tau.([\![\langle a \rangle]\!]|P) \\ [\![\underline{a}.P]\!] & = & a'.P \end{array}$$

But some problems remain. The manager $rec\ x.(\overline{a'}.x + \overline{a}.\underline{0})$ of the message a is not able to give its contents to an arbitrary quantity of reading processes all at the same time, but it defines an order on them. Also in our language parallel processes which read the same message are not able to perform their operations simultaneously, but this is implicitly related to the fact that we have defined an *interleaving* semantics (i.e. a semantics in which parallel independent operations can be executed in every possible order but not simultaneously). In the case of *non interleaving* semantics the above mapping on CCS will be not appropriate, as it introduces unnecessary sequentializations on readers. Our language can be given a *step* semantics (Nielsen and Thiagarajan, 1984) only by adding the following rules:

$$\frac{P \xrightarrow{\alpha} P'}{P \overset{\{\!|\alpha|\!\}}{\twoheadrightarrow} P'} \qquad\qquad \frac{\forall i \in \{1 \ldots n\} : \alpha_i = \overline{\underline{a}}}{\langle a \rangle \overset{\{\!|\alpha_1, \alpha_2, \ldots, \alpha_n|\!\}}{\twoheadrightarrow} \langle a \rangle}$$

$$\frac{P \overset{\{\!|\alpha_1, \alpha_2, \ldots, \alpha_n|\!\}}{\twoheadrightarrow} P' \quad Q \overset{\{\!|\beta_1, \beta_2, \ldots, \beta_m|\!\}}{\twoheadrightarrow} Q' \quad \exists f : \{1 \ldots s\} \overset{1-1}{\longleftrightarrow} \{1 \ldots s\}\ s.t.\ \alpha_i = \overline{\beta_{f(i)}}}{\text{if } \ s = m = n \ \text{ then } \ P|Q \overset{\{\!|\tau|\!\}}{\twoheadrightarrow} P'|Q' \ \text{ else } \ P|Q \overset{\{\!|\alpha_{s+1}, \ldots, \alpha_n, \beta_{s+1}, \ldots, \beta_m|\!\}}{\twoheadrightarrow} P'|Q'}$$

where $\{\!|\alpha_1, \alpha_2, \ldots, \alpha_n|\!\}$ is used to represent the multiset which contains the elements $\alpha_1, \alpha_2, \ldots, \alpha_n$. The semantics that is obtained by adding the above rules allows two reading processes to access the same message simultaneously, i.e., $\langle a \rangle|\underline{a}.P|\underline{a}.Q \overset{\{\!|\tau|\!\}}{\twoheadrightarrow} \langle a \rangle|P|Q$.

We now analyze the originality of our framework with respect to other approaches to the formal analysis of the semantics of generative communication.

In (Ciancarini, Jensen, and Yankelevich, 1995) several frameworks such as CCS, Petri Nets and Chemical Abstract Machine (Berry and Boudol, 1992) are used as semantic domains for the coordination language Linda. That paper studies different possible implementations of generative communication in other computational models, but semantic equivalences (and their axiomatizations) are not presented.

In (De Nicola and Pugliese, 1995) observational equivalences based on testing (De Nicola and Hennessy, 1984) are applied to a language obtained by embedding the Linda primitives in a simple sequential host language. That is why that language is more complex than ours and axiomatic characterizations of the equivalences are not presented. Moreover, we are convinced that the properties of the generative communication mechanism are orthogonal to the features of the sequential host language: for instance, the most important property focused in (De Nicola and Pugliese, 1995), i.e. program `out(N,5); out(M,9)` is observationally undistinguishable from `out(M,9); out(N,5)`, is easily proved in our simpler framework by means of the general law $\hat{a}.\hat{b}.\underline{0} \simeq_F \hat{b}.\hat{a}.\underline{0}$.

Our formalism can be also compared with other frameworks for asynchronous communication. In fact, standard asynchronous communication can be obtained by eliminating the `read` prefix $\underline{a}$ from our language. Moreover, even if generative communication uses only one single data structure as a communication medium, it is not difficult to model several channels in our formalism. It is enough to add the information related to the channel in which the message is inserted to the messages' identification name (e.g. (t, a) represents a message a sent along channel t).

In (De Boer, Klop, Palamidessi, and Rutten, 1991) a compositional semantic model, based on sequences of pairs of states, is defined for a general asynchronous language. In that framework, refusal information is not required in order to describe deadlock because a trace-like model is sufficient. That is why the authors said that failure semantics "fails" in the asynchronous case. This apparent inconsistency with our approach is essentially due to a basic technical difference between the two frameworks: the state of the communication medium is included in our agents (each single sent message a is denoted by $\langle a \rangle$), whereas in their approach it is considered external to the terms of the language.

More recently (De Boer, Klop, and Palamidessi, 1992) an "encapsulation operator" is introduced to model asynchronous communication in ACP. The "encapsulation operator" is used as a store for the messages which have been sent along a certain channel. A first basic difference with our approach is that the order of transmission of the messages is a determinating factor. If the channel is a queue, the authors suggest that this distinction is useful when messages are sent along the same channel, as in a queue the sending order influences the reading order. However, in their approach the distinction is kept also when the sending order should influence in no way the reading order, e.g., when the messages are sent along different channels. In fact, $t{\uparrow}a.s{\uparrow}b.P$ (corresponding to the term $\widehat{(t, a)}.\widehat{(s, b)}.P$ of our language) is not equivalent to $s \uparrow b.t \uparrow a.P$ (corresponding to $\widehat{(s, b)}.\widehat{(t, a)}.P$). This is due to the fact that only in our framework the execution of an `out` operation is not visible until the sent message is read. Moreover, standard failure equivalence does not correctly describes deadlock in that formalism. In fact, the agent $t{\uparrow} a.s{\uparrow}b.P + t{\uparrow}a.s{\uparrow}c.P$ (corresponding to $\widehat{(t, a)}.\widehat{(s, b)}.P + \widehat{(t, a)}.\widehat{(s, c)}.P$) and $t{\uparrow}a.(s{\uparrow}b.P + s{\uparrow}c.P)$ (corresponding to $\widehat{(t, a)}.(\widehat{(s, b)}.P + \widehat{(s, c)}.P)$) are not standard failure equivalent even if they can be considered observationally undistinguishable (see Example 49). In order to solve this problem special-purpose refusal sets are defined: the sending ("intended output") operations are not introduced in the set X of each failure $[s, X]$. In our opinion, a formalism in which the standard equivalences correctly describes the properties of asynchronous communication is more suitable w.r.t. other frameworks in which the equivalences must be adapted in order to capture the intended meaning.

In two papers (Honda and Tokoro, 1991) and (Boudol, 1992), asynchronous communication is embedded in the π-calculus (Milner, Parrow, and Walker, 1992) using a representation of the sent messages very similar to ours, but an explicit prefix for the `out` operation is not defined. In fact, the process P which sends the message a and becomes P', is directly represented by means of the parallel composition of the agent representing the message a and P'. In our calculus, this means that $\hat{a}.P'$ should be considered structurally equivalent to $\langle a\rangle|P'$. In our opinion this structural equivalence gives rise to some problems when the choice composition $+$ is considered (the above are choice-free languages). In fact, the agents $P = \hat{a}.\underline{0} + \hat{b}.\underline{0}$ and $Q = \langle a\rangle + \langle b\rangle$ (which should be considered structurally equivalent) are used in our framework to represent different situations. In the first case, the agent P sends either the message a or the message b and the choice is internal to P. In the second case, the messages have been already sent, but only one of them can be read and the choice is left to the reading process.

A framework more similar to ours is proposed in (Cleaveland and Yankelevich, 1994) where a CCS with value passing and asynchronous communication is presented. Even in that paper the `out` operations are executed by means of local (i.e. τ labeled) transitions. But, differently from our language, the message remains related to the process from which it has been sent. In fact, in our framework a sent message is simply composed in parallel with the sending process (i.e. $\hat{a}.P \xrightarrow{\tau} \langle a\rangle|P$), while in their approach a message is composed with its sending process by means of the auxiliary operator $\lhd$ in the following way: $\hat{a}.P \xrightarrow{\tau} \langle a\rangle \lhd P$. Moreover, the messages can not be read by the processes from which they have been sent. For these reasons, this approach is not suitable to model *generative communication*, where a message is equally accessible to all processes (also the process which sent it) and it is bound to none (neither its generating process). Furthermore, no semantic equivalences are defined on their formalism in order to identify terms with different operational behavior which can be considered semantically equivalent.

Acknowledgements

This research was partially supported by EC BRA n. 9102 COORDINATION. We thank Catuscia Palamidessi for her helpful suggestions on a preliminary version of the paper and the anonymous referees for their accurate and stimulating comments.

REFERENCES

Bergstra, J.A. and Klop, J.W. (1986) *Process algebra: specification and verification in bisimulation semantics*. CWI Monographs. North-Holland.

Berry, G. and Boudol, G. (1992) The chemical abstract machine. *Theoretical Computer Science*, 96:217-248.

Brogi, A. and Ciancarini, P. (1991) The concurrent language Shared Prolog. *ACM Transactions on Programming Languages and Systems*, 13(1):99-123.

Boudol, G. (1992) Asynchrony and the π calculus. Technical Report 1702, INRIA Sophia-Antipolis, France.

Brookes, S.D. (1983) On the relationship of CCS and CSP, in *ICALP'83*, volume 154 of *LNCS*, pages 83-96. Springer Verlag.

Carriero, N. and Gelernter, D. (1986) The S/Net's Linda kernel. *ACM Transactions on Computer Systems*, pages 110-129.

Carriero, N., Gelernter, D. and Zuck,L. (1995) Bauhaus Linda, in *Object-Based Models and Languages for Concurrent Systems*, volume 924 of *LNCS*, pages 66-76. Springer Verlag.

Ciancarini, P., Gorrieri, R. and Zavattaro, G. (1995) Generative Communication in Process Algebra. Technical Report 95-16, University of Bologna, Italy.

Ciancarini, P., Jensen, K.K. and Yankelevich, D. (1995) On the Operational Semantics of a Coordination Language, in *Object-Based Models and Languages for Concurrent Systems*, volume 924 of *LNCS*, pages 77-106. Springer Verlag.

Cleaveland, R. and Yankelevich, D. (1994) An Operational Framework for Value-Passing Processes, in *POPL'94*.

De Boer, F.S., Klop, J.W. and Palamidessi, C. (1992) Asynchronous communication in process algebra, in *LICS'92*, pages 137-159. IEEE Computer Society Press.

De Boer, F.S., Kok, J.N., Palamidessi, C. and Rutten, J.J.M.M. (1991) The Failure of Failures in a Paradigm for Asynchronous Communication, in *Concur'91*, volume 527 of *LNCS*, pages 111-126. Springer Verlag.

De Boer, F.S. and Palamidessi, C. (1990) On the Asynchronous Nature of Communication in Concurrent Logic Languages: a Fully Abstract Model based on Sequences, in *Concur'90*, volume 458 of *LNCS*, pages 99-114. Springer Verlag.

De Nicola, R. and Hennessy, M.C.B. (1984) Testing Equivalences for Processes. *Theoretical Computer Science*, 34:83-133.

De Nicola, R. and Pugliese, R. (1995) An Observational Semantics for Linda, in *STRICT'95*, series *Workshop in Computing*, pages 129-143. Springer Verlag.

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

Gelernter, D. and Carriero, N. (1992) Coordination Languages and their Significance. *Communications of the ACM*, 35(2):97-107.

Hoare, C.A.R. (1978) Communicating Sequential Processes. *Communications of the ACM*, 21:666-677.

Honda, K. and Tokoro,M. (1991) An Object Calculus for Asynchronous Communication, in *ECOOP'91*, volume 512 of *LNCS*, pages 133-147. Springer Verlag.

Milner, R. (1989) *Communication and Concurrency*. Prentice-Hall.

Milner, R., Parrow, J. and Walker, D. (1992) A Calculus of Mobile Processes. *Information and Computation*, 100(1):1-77.

Nielsen, M. and Thiagarajan, P.S. (1984) Degrees of non-determinism and concurrency: a Petri net view, in *5th Conference on Foundation of Software Technology and Thoeretical Computer Science*, volume 181 of *LNCS*, pages 89-118. Springer Verlag.