Comparing Software Architectures for Coordination Languages

Marcello M. Bonsangue[1], Joost N. Kok[2], and Gianluigi Zavattaro[3]

[1] Centrum voor Wiskunde en Informatica, Amsterdam, The Netherlands
Marcello.Bonsangue@cwi.nl

[2] Leiden Institute of Advanced Computer Science, Leiden University, The Netherlands
joost@cs.leidenuniv.nl

[3] Department of Computer Science, University of Bologna, Italy
zavattar@cs.unibo.it

Abstract. We discuss three software architectures for coordination. All architectures are based on agents. Each agent has a local dataspace that contains shared distributed replicated data. The three architectures differ in the way agents communicate: either through an unordered broadcast, through an atomic broadcast, or through a synchronization among all agents. We first show how to represent both data-driven and control-oriented coordination languages in our model. Then we compare the behavior of the three architectures, under the assumption that the local dataspaces are either sets or multisets.

1 Introduction

A software architecture comprises processing elements, data elements, and rules that characterize the interaction among these components. An essential part of a software architecture is a collection of constraints on processing and data elements that ensure that the system satisfies a set of need requirements [14]. Examples of need requirements are the view processes can have on data and guarantees that certain data is received in the same order by all agents.

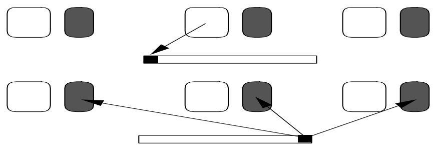
When building a software architecture for coordination languages, a designer is confronted with a number of choices, including the view that processes have on data (e.g. data is information, or data is a resource), the kind of coordination to be used (e.g. data-driven or control-oriented), the coordination primitives, and the protocol for implementing a broadcast of data.

We will consider three different classes of software architectures, differing in the broadcast mechanism used. In each architecture there are a number of agents which interact only by broadcasting data to all agents. Each agent consists of an active process together with a local memory used as a data repository.

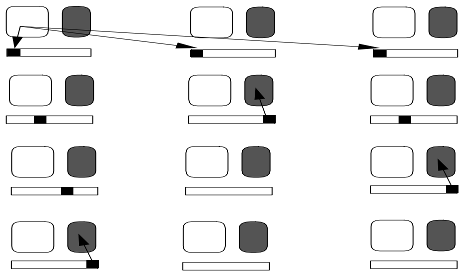
The processing element of each agent can, besides performing internal computations, executes some coordination operations in order to interact with the other agents. There are operations for inserting data in the local memory (*wrt*), for broadcasting data (*put*), for testing the presence of data in the memory (*rd*), and for the local (*get*) and global (*del*) consumption of data. There are several ways these operations can be implemented: we will consider blocking *rd*, *get*, and *del* operations, and non-blocking *wrt* and *put* operations. Furthermore, the *put* and *del* operations use the broadcast mechanism offered by the architecture.

The three classes of architectures we consider differ in their broadcasting mechanisms. The simplest broadcast mechanism to describe (and hardest to implement in a distributed system) is the synchronous one: when a data value is produced it is inserted immediately in the memory of each agent at the same time. Because there is no observable delay between the production and the receiving of data by all agents, we call this architecture *undelayed*.

In a second architecture, called *globally delayed*, agents communicate through an atomic broadcast: there can be a delay between the production of data and its actual reception by all agents, but all agents are guaranteed to receive the broadcast data value at the same time. We model this global delay using a shared queue of pending messages. When a data value is broadcast, it is inserted in this queue. Eventually it will be removed from the queue and stored in all the memories of the agents at the same time.



In our third architecture, that we call *locally delayed*, agents use an unordered broadcast, meaning that each agent may receive a broadcast data value at a different moment in time. We model this local delay by associating a queue of pending messages to each agent. When a data value is broadcast, it is inserted in all queues at the same time, but the transfer from each queue to the corresponding dataspace can happen for each agent at a different time.



These architectures are parameterized by the structure of the dataspaces, either multisets or sets. In the first case multiplicity of data is significant and hence data is interpreted as a resource. In the second case, multiplicity is insignificant and data is seen as information.

Several coordination languages can be modeled in one or more of these architectures. In this paper we model two simple data-driven coordination languages inspired by Linda [12] and Splice [5], respectively, and a control-oriented coordination language inspired by Manifold [4].

Then we compare the behavior of the three architectures, under the assumption that the local dataspaces are either sets or multisets. We give several equivalence results: informally, we say that two architectures are observationally equivalent if there exists an encoding of one into the other preserving the processing components and such that each step of one architecture can be simulated by the other one up to some unobservable communication steps, and vice-versa. We give also difference results by exhibiting processes that can produce a data value in one architecture but not in the other one. These relationships give useful insights to a designer when building a software architecture. For example it may be interesting to find an equivalent simple model of a distributed system, abstracting from the nature of broadcasting, or, conversely, one might want to implement a simple conceptual model in a distributed architecture.

The structure of the paper is as follows. The next section introduces the definitions common to all three architectures, while the specific description of the architectures is in Section 3. In Section 4 we model simple coordination languages within our architectures. The three architectures are compared in Section 5. In Section 6 we draw some conclusions and discuss related work. In an appendix we give some alternative definitions of the global delete operator.

2 Processing and data elements

Let Data, ranged over by $a, b, \ldots$, be a set of *data values*, and define $\mathit{Msg} = \mathit{Data} \cup \{\overline{a} \mid a \in \mathit{Data}\}$, ranged over by $x, y, \ldots$, to be a set of *messages* used by the agents of the architectures for their interactions. The message a denotes the request for insertion of the value a, while $\overline{a}$ represents the request for deletion of a. We use the convention that $\overline{\overline{x}} = x$ for $x \in \mathit{Msg}$. When produced, messages are associated to a sort used by the communication protocol to guarantee a common order in their reception among all agents. We assume the existence of an abstract set Sorts of *data sorts*, ranged over by $s, t, \ldots$, and define the set Σ of *broadcastable messages* as follows:

$$\Sigma = \{x{:}s \mid x \in \mathit{Msg}, s \in \mathit{Sorts}\}\,.$$

Two messages with different sorts (e.g. because broadcast by different agents) may be received by an agent in any order, while two messages with the same sort (e.g. because broadcast by the same process) will be received by any agent in the same order as they were produced.

The behavior of each process is described by a synchronization tree: let Proc, ranged over by $P, Q, \ldots$, be the set of all synchronization trees defined by the grammar

$$P ::= 0 \mid \sum_{i \in I} \alpha_i.P_i$$
$$\alpha ::= \tau \mid \mathit{rd}(a) \mid \mathit{put}(a{:}s) \mid \mathit{wrt}(a) \mid \mathit{del}(a{:}s) \mid \mathit{get}(a)$$

where $a \in \mathit{Data}$, $s \in \mathit{Sorts}$ and I is a non-empty (possibly infinite) index set. Informally, τ denotes some internal activity, $\mathit{rd}(a)$ tests for the presence of an occurrence of the value a in the local dataspace without consuming it, $put(a{:}s)$ emits a new instance of the value a that is broadcast to all the agents as the message $a{:}s$, $wrt(a)$ introduces a new instance of value a in the local dataspace, $del(a{:}s)$ broadcasts the message $\overline{a}{:}s$ indicating the request for deletion of one instance of value a, $get(a)$ consumes a local occurrence of value a. The put and del actions associate to data the sorts used during their broadcast.

Following [3, 7], we model broadcasting of messages using *queues of pending messages*. A queue q is a partially commuting string defined as a congruence class of finite strings in the monoid $(\Sigma^\star, \cdot, \varepsilon)$ modulo the least congruence such that, for all $x{:}s, y{:}t \in \Sigma$,

$$x{:}s \cdot y{:}t = y{:}t \cdot x{:}s \ \text{ if } s \neq t\,.$$

Commutation of messages of different sorts allows for a compact formalization of several broadcasting mechanisms. We let DQ be the set of all queues of pending messages, and write $\odot$ for the string concatenation $\cdot$ modulo the above congruence. Also, we denote a congruence class containing a single element by the element itself. Hence $x{:}s$ is the congruence class containing the one-element string $x{:}s \in \Sigma$ and ε the congruence class containing the empty string.

When received, messages are stored in *dataspaces*. The set of all dataspaces is denoted by DS and ranged over by d. We will consider two kinds of dataspaces: multisets and sets. In the first case we let $DS = Msg \rightarrow \mathbb{N}$ while in the second case $DS = Msg \rightarrow \{0, 1\}$. For $d \in DS$ and $x \in Msg$ we say $in(d, x) = tt$ if $d(x) > 0$. Also, we define the insertion of $x \in Msg$ into a dataspace $d \in DS$ as follows:

$$d \oplus x = \begin{cases} d[x/d(x)+1] & \text{if } d(x)+1 \in cod(d) \text{ and } in(d, \overline{x}) \neq tt \\ d[\overline{x}/d(\overline{x})-1] & \text{if } in(d, \overline{x}) = tt \\ d & \text{otherwise} \end{cases}$$

where, for any function $f\colon X \rightarrow Y$ we denote by $cod(f)$ its codomain Y, and, for $x \in X$ and $y \in Y$, we denote by $f[x/y]$ the function mapping x to y and acting as f otherwise. The above operation is defined for both interpretations of a dataspace as set or multiset (the condition on the codomain of the dataspace makes the distinction here), and for adding and removing data values from a dataspace. Informally, a data value a is inserted into a dataspace only if no request for deletion $\overline{a}$ is present. Otherwise a is not inserted and $\overline{a}$ is removed from the dataspace. Conversely, a data value a from a dataspace is removed when the message $\overline{a}$ arrives. In case the message $\overline{a}$ arrives and the value a is not present in the dataspace then $\overline{a}$ is stored into the dataspace.

3 Three software architectures

3.1 The locally delayed architecture

The set $Conf_L$ of possible configurations of the agents in the locally delayed architecture is defined by the grammar

$$C ::= [P, d, q] \ \mid \ C \parallel C\,.$$

where $\|$ is a commutative and associative parallel composition operator. Here P is a process and $d \in DS$ a local dataspace (together forming an agent), and $q \in DQ$ is a queue containing the messages already produced by some agent in the system but not yet received by (P, d). The dynamical evolution of the agents is specified by a transition relation $\longrightarrow_L$, defined as the least relation in $Conf_L \times (\{\tau\} \cup \Sigma) \times Conf_L$ satisfying the following axioms and rules:

$$
\begin{array}{|lll|}
\hline
(L1) & [\tau.P, d, q] \xrightarrow{\tau} [P, d, q] & \\
(L2) & [rd(a).P, d, q] \xrightarrow{\tau} [P, d, q] & \text{if } in(d, a) = tt \\
(L3) & [put(a{:}s).P, d, q] \xrightarrow{a:s} [P, d, a{:}s \odot q] & \\
(L4) & [wrt(a).P, d, q] \xrightarrow{\tau} [P, d \oplus a, q] & \\
(L5) & [del(a{:}s).P, d, q] \xrightarrow{\overline{a}:s} [P, d, \overline{a}{:}s \odot q] & \text{if } in(d, a) = tt \\
(L6) & [get(a).P, d, q] \xrightarrow{\tau} [P, d \oplus \overline{a}, q] & \text{if } in(d, a) = tt \\
(L7) & [P, d, q \odot x{:}s] \xrightarrow{\tau} [P, d \oplus x, q] & \\
(L8) & \dfrac{[\alpha_k.P_k, d, q] \xrightarrow{\ell} [P', d', q'] \quad k \in I}{[\sum_I \alpha_i.P_i, d, q] \xrightarrow{\ell} [P', d', q']} & \ell \in \{\tau\} \cup \Sigma \\
(L9) & \dfrac{C \xrightarrow{\tau} C'}{[P, d, q] \parallel C \xrightarrow{\tau} [P, d, q] \parallel C'} & \\
(L10) & \dfrac{C \xrightarrow{x:s} C'}{[P, d, q] \parallel C \xrightarrow{x:s} [P, d, x{:}s \odot q] \parallel C'} & x{:}s \in \Sigma \\
\hline
\end{array}
$$

The conditions in $(L2)$, $(L5)$, and $(L6)$ say that the *rd*, *del*, and *get* operators are blocking. Rules $(L3)$ and $(L5)$ in combination with $(L10)$ show that the effect of the *put* and *del* operations is global to all agents, while that of the *rd*, *wrt*, and *get* operations is local to the agent executing the operations (specified by $(L2)$, $(L3)$, $(L6)$ and $(L9)$). The receiving of a message is specified by $(L7)$.

The labels τ, $a{:}s$ and $\overline{a}{:}s$ in the above specification have been introduced for a structural definition of the transition relation $\longrightarrow_L$. When comparing the architectures we are interested in the reduction steps that the system can execute. To this aim we define a reduction relation $\longrightarrow\!\!\!\!\!\rightarrow_L \subseteq Conf_L \times Conf_L$ abstracting from the labels used by the transition relation $\longrightarrow_L$:

$$C \longrightarrow\!\!\!\!\!\rightarrow_L C \;\text{ if and only if }\; \exists \ell \in \{\tau\} \cup \Sigma \,.\, C \xrightarrow{\ell}_L C' \,.$$

We define also a weaker reduction relation $\Longrightarrow_L$ as the reflexive and transitive closure of $\longrightarrow\!\!\!\!\!\rightarrow_L$, that is $\Longrightarrow_L = (\longrightarrow\!\!\!\!\!\rightarrow_L)^*$.

3.2 The globally delayed architecture

The set of configurations of the agents in this architecture is given by

$$Conf_G = \{(A, q) \mid A \in Agents, q \in DQ\}\,,$$

where *Agents* is defined by the grammar:

$$A ::= [P, d] \mid A \parallel A\,,$$

where $\parallel$ is used here as a commutative and associative parallel composition operator for agents. As usual P is a process and $d \in DS$ is a local dataspace. The difference with the configuration of the previous architecture is that here all agents share the same queue of pending messages.

Observe that $\parallel$ is overloaded because it is used for both agents and configurations; this does not introduce confusion because the operator is usually determined by the context.

The behavior of the agents in the globally delayed architecture is specified by means of the transition relation $\longrightarrow_G$, defined as the least relation in $\mathit{Conf}_G \times (\{\tau\} \cup \mathit{Msg}) \times \mathit{Conf}_G$ satisfying the following axioms and rules:

$$\boxed{\begin{array}{lll}
(G1) & [\tau.P, d], q \xrightarrow{\tau} [P, d], q & \\
(G2) & [rd(a).P, d], q \xrightarrow{\tau} [P, d], q & \text{if } in(d, a) = tt \\
(G3) & [put(a{:}s).P, d], q \xrightarrow{\tau} [P, d], a{:}s \odot q & \\
(G4) & [wrt(a).P, d], q \xrightarrow{\tau} [P, d \oplus a], q & \\
(G5) & [del(a{:}s).P, d], q \xrightarrow{\tau} [P, d], \overline{a}{:}s \odot q & \text{if } in(d, a) = tt \\
(G6) & [get(a).P, d], q \xrightarrow{\tau} [P, d \oplus \overline{a}], q & \text{if } in(d, a) = tt \\
(G7) & [P, d], q \odot x{:}s \xrightarrow{x} [P, d \oplus x], q & \\
(G8) & \dfrac{[\alpha_k.P_k, d], q \xrightarrow{\ell} [P', d'], q' \quad k \in I}{[\sum_I \alpha_i.P_i, d], q \xrightarrow{\ell} [P', d'], q'} & \ell \in \{\tau\} \cup \mathit{Msg} \\
(G9) & \dfrac{A, q \xrightarrow{\tau} A', q'}{([P, d] \parallel A), q \xrightarrow{\tau} ([P, d] \parallel A'), q'} & \\
(G10) & \dfrac{A, q \xrightarrow{x} A', q'}{([P, d] \parallel A), q \xrightarrow{x} ([P, d \oplus x] \parallel A'), q} & x \in \mathit{Msg}
\end{array}}$$

Here $(G3)$ shows that when a data item is broadcast it is not immediately visible to all agents. The fact that eventually they will receive a message at the same time is modeled by $(G7)$ together with $(G10)$. As in the previous architecture, the *rd*, *del* and *get* operators are blocking ($(G2)$, $(G5)$ and $(G6)$, respectively), the *rd*, *wrt* and *get* operations are local ($(G2)$, $(G5)$ and $(G4)$ together with $(G8)$), and the *put* and *del* operations have a global effect obtained through the broadcasting of a message ($(G3)$ and $(G5)$ together with $(G10)$).

As we have done for the previous architecture, we define a reduction relation $\twoheadrightarrow_G \subseteq \mathit{Conf}_G \times \mathit{Conf}_G$ and its weaker correspondent $\Longrightarrow_G$:

$$\begin{array}{l}
C \twoheadrightarrow_G C' \;\text{ if and only if }\; \exists \ell \in \{\tau\} \cup \mathit{Msg}\,.\, C \xrightarrow{\ell}_G C'\,, \\
\Longrightarrow_G \;=\; (\twoheadrightarrow_G)^*\,.
\end{array}$$

3.3 The undelayed architecture

The behavior of the agents in the undelayed architecture is specified by the transition relation $\longrightarrow_U$, that is the least relation in $\mathit{Agents} \times (\{\tau\} \cup \mathit{Msg}) \times \mathit{Agents}$ satisfying the following axioms and rules:

$$
\begin{array}{|lr|}
\hline
(U1)\ [\tau.P, d] \xrightarrow{\tau} [P, d] & \\
(U2)\ [\mathit{rd}(a).P, d] \xrightarrow{\tau} [P, d] & \text{if } \mathit{in}(d, a) = \mathit{tt} \\
(U3)\ [\mathit{put}(a{:}s).P, d] \xrightarrow{a} [P, d \oplus a] & \\
(U4)\ [\mathit{wrt}(a).P, d] \xrightarrow{\tau} [P, d \oplus a] & \\
(U5)\ [\mathit{del}(a{:}s).P, d] \xrightarrow{\overline{a}} [P, d \oplus \overline{a}] & \text{if } \mathit{in}(d, a) = \mathit{tt} \\
(U6)\ [\mathit{get}(a).P, d] \xrightarrow{\tau} [P, d \oplus \overline{a}] & \text{if } \mathit{in}(d, a) = \mathit{tt} \\
(U7)\ \dfrac{[\alpha_k.P_k, d] \xrightarrow{\ell} [P', d'] \quad k \in I}{[\sum_I \alpha_i.P_i, d] \xrightarrow{\ell} [P', d']} & \ell \in \{\tau\} \cup \mathit{Msg} \\
(U8)\ \dfrac{A \xrightarrow{\tau} A'}{[P, d] \parallel A \xrightarrow{\tau} [P, d] \parallel A'} & \\
(U9)\ \dfrac{A \xrightarrow{x} A'}{[P, d] \parallel A \xrightarrow{x} [P, d \oplus x] \parallel A'} & x \in \mathit{Msg} \\
\hline
\end{array}
$$

The synchronous behavior of the *put* and *del* operations is modeled by $(U3)$ and $(U5)$ together with $(U9)$. All other operations are local (the remaining axioms together with $(U8)$). As in the previous architectures, the *rd*, *del*, and *get* operations are blocking, while the *put* and the *wrt* operations are not. Observe that in the undelayed architecture sorts do not play any role. Indeed, no queue of pending messages is present and hence sorts could be safely eliminated from the *del* and *put* operations. Nevertheless, they are present for consistency with the previous architectures.

As before, we define a reduction relation $\longrightarrow\!\!\!\!\!\rightarrow_U \subseteq \mathit{Agents} \times \mathit{Agents}$ and its weaker correspondent $\Longrightarrow_U$:

$$
\begin{aligned}
& A \longrightarrow\!\!\!\!\!\rightarrow_U A' \ \text{ if and only if } \ \exists \ell \in \{\tau\} \cup \mathit{Msg} \,.\, A \xrightarrow{\ell}_U A' \,, \\
& \Longrightarrow_U \ = \ (\longrightarrow\!\!\!\!\!\rightarrow_U)^* \,.
\end{aligned}
$$

4 Modeling coordination languages

In this section we give examples about how to model coordination in our architectures.

4.1 A shared data-driven language

We start with a coordination language inspired by one of the abstract models of Linda [12] presented in [11]. It consists of operations acting on a shared dataspace. Processes in parallel can introduce values, test for their presence, and consume them via the following syntax:

$$S ::= end \mid \alpha.S \mid S + S$$
$$\alpha ::= out(a) \mid rd(a) \mid in(a)$$

where a is an element of an abstract set Val of values.

Consider $Data = Val$ and let $Sort = Msg$. We model the above language by means of the undelayed architecture, where we consider the local dataspaces as distributed consistent copies of the same shared dataspace. It is not hard to see that the local dataspaces are kept consistent because we are dealing with a synchronous broadcast, and no local operations like *wrt* or *get* are considered. Given a statement S we define its behavior in isolation as the synchronization tree $[\![S]\!]$. It is given inductively as follows:

$$\begin{array}{ll} [\![end]\!] \quad = 0 & [\![out(a).S]\!] = put(a{:}a).[\![S]\!] \\ [\![rd(a).S]\!] = rd(a).[\![S]\!] & [\![in(a).S]\!] \;\; = del(a{:}\overline{a}).[\![S]\!] \\ [\![S_1 + S_2]\!] = [\![S_1]\!] + [\![S_2]\!] & \end{array}$$

We could interpret the above language also in the globally delayed architecture using the synchronous *del* operator as specified in the Appendix. In this case, the way we assign sorts to each message to be broadcasted implies that the order in which data is received is insignificant with respect to the order of its production.

4.2 A distributed data-driven language

Next we model another data-driven coordination language inspired by the abstract model of Splice [5] as presented in [6]. The main difference with the previous language is that the local dataspace of each agent need not to be a consistent copy of a shared one.

Let Val be a countable set of *values*, Var a set of data variables, and Idx a set of *data indexes*. We define the syntax of our distributed data-driven coordination language as follows:

$$\alpha ::= out(\iota, v) \mid rd(\iota, v) \mid in(\iota, v)$$
$$S ::= end \mid \alpha.S \mid S + S$$

where $v \in Val \cup Var$ and $\iota \in Idx$. The language consists of operations for the manipulation of structured data $\langle \iota, v \rangle$, where ι is an index and v a value. Intuitively the coordination primitives are as for the previous language, but *in* acts locally on the store. The index is used to retrieve data, and data variables match with any data item. For example, if $v \in Var$ then $in(\iota, v)$ will consume one of the values with index ι from the local dataspace of the agent executing it. Data is broadcast using a message passing protocol which guarantees that values with the same index are received by any agent in the same order as they where produced. Furthermore, it is required that there is at most one agent producing values with a given index.

The above broadcasting mechanism fits well with those used by the locally and globally delayed architectures. In order to model the language in those architectures, let $Data = Idx \times Val$ and $Sorts = Idx$. Given a statement S we define its behavior in isolation as a synchronization tree $[\![S]\!]$. It is given inductively as follows:

$$
\begin{array}{lll}
[\![end]\!] & = 0 & \\
[\![out(\iota, u).S]\!] & = put(\langle \iota, u \rangle{:}\iota).[\![S]\!] & u \in Val \\
[\![rd(\iota, u).S]\!] & = rd(\langle \iota, u \rangle).[\![S]\!] & u \in Val \\
[\![rd(\iota, v).S]\!] & = \sum_{u \in Val} rd(\langle \iota, u \rangle).[\![S[u/v]]\!] & v \in Var \\
[\![in(\iota, u).S]\!] & = get(\langle \iota, u \rangle).[\![S]\!] & u \in Val \\
[\![in(\iota, v).S]\!] & = \sum_{u \in Val} get(\langle \iota, u \rangle).[\![S[u/v]]\!] & v \in Var \\
[\![S_1 + S_2]\!] & = [\![S_1]\!] + [\![S_2]\!] &
\end{array}
$$

where $S[u/v]$ is the statement obtained by replacing all occurrences in S of the variable $v \in Var$ by the value $u \in Val$. The summation used for the rd and in operations denotes the openness of the system to the environment, which has to determine which branch will be chosen.

4.3 A control-oriented language

We conclude this section by modeling a language inspired to the event mechanism of the control-oriented language Manifold [4], of which the operational semantics is given in [3].

Let Evn be a set of events and Pid be a set of process identifiers. We define the syntax of our control-oriented coordination language as follows:

$$
\begin{array}{lll}
G & ::= & ep?S \mid G + G \\
ep & ::= & (e, p) \mid (*e, *p) \mid (*e, p) \mid (e, *p) \\
S & ::= & end \mid G \mid raise(e).S \mid post(e).S
\end{array}
$$

where $e \in Evn$ and $p \in Pid$. A system consists of several agents with a local store and with a unique process identifier. The behavior of each process is driven by reactions to broadcasted events. Event and process identifiers prefixed by a $*$ are formal parameters to be substituted by actual events and process identifiers, respectively, when a reaction takes place. Raising events means broadcasting them to all agents in the system using a protocol that guarantees that events raised by the same process are received by all agents in the same order as they were raised. Posting events is sending events only to the process itself.

We can use the locally delayed architecture to model the above language. Define $Data = Events \times Pid$ and let $Sort = Pid$. For each $q \in Pid$ and statement G we define the behavior of the agent q executing G as the synchronization tree $[\![G]\!](q)$, defined inductively as follows:

$$
\begin{array}{ll}
[\![(e,p)?S]\!](q) & = get((e,p)).[\![S]\!]_q \\
[\![(*e,*p)?S]\!](q) & = \sum_{e' \in Evn, p' \in Pid} get((e',p')).[\![S[e'/e][p'/p]]\!]_q \\
[\![(*e,p)?S]\!](q) & = \sum_{e' \in Evn} get((e',p)).[\![S[e'/e]]\!]_q \\
[\![(e,*p)?S]\!](q) & = \sum_{p' \in Pid} get((e,p')).[\![S[p'/p]]\!]_q \\
[\![G_1 + G_2]\!](q) & = [\![G_1]\!](q) + [\![G_2]\!](q)
\end{array}
$$

and for each statement S and $q \in PId$, $[\![S]\!]_q$ is defined by

$$\begin{array}{llll} [\![end]\!]_q & = 0 & [\![G]\!]_q & = [\![G]\!](q) \\ [\![raise(e).S]\!]_q & = put((e,q){:}q).[\![S]\!]_q & [\![post(e).S]\!]_q & = wrt((e,q)).[\![S]\!]_q\,. \end{array}$$

Note that events raised by different agents will produce data of different sorts. Hence the order in which data is sent by an agent is preserved.

5 Comparing the architectures

When comparing two architectures we have to deal with two possible kinds of results: either we prove that they are observationally equivalent or that they are observationally different. When proving an equivalence result, we will consider a strong notion of equivalence, and we use no assumptions on the sorts of the broadcast messages. Furthermore we make no assumptions about the initial state of the agents. In this way the equivalence results hold also for weaker notions of observation and for other instances of the sort mechanism. Informally, we say that two architectures are *observationally equivalent* if there exists an encoding of one into the other preserving the processing components, and such that each reduction step of one architecture can be simulated by the other one in zero or more steps, and vice-versa.

On the other hand, when we prove a difference result, we use a weak notion of observation: two architectures are observationally different if there exists an agent executing a process that can produce a data value in one architecture but not in the other one. Sometimes difference results rely on some assumptions about the sorts of the broadcast messages and assumptions about the initial content of the local dataspaces. These assumptions are justified by the protocol used by some broadcast mechanisms for many realistic software architectures. In this way, the difference results we present hold also for stronger notions of observation and for other instances of the sort mechanism.

5.1 Comparison without consuming data

We start by proving that if we consider processes described by synchronization trees without local or global delete operations *get* and *del*, then the above three architectures are all equivalent. Furthermore the equivalences do not depend on the choice between dataspaces as sets or multisets, that is, the multiplicity of data is not important. These results holds because we are dealing with dataspaces that monotonically grow (as no delete operation is permitted).

First we need to introduce some notation. For $A \in Agents$ and $q \in DQ$, we denote by $A \Leftarrow q$ the collection of agents obtained after all values in the queue q have been flushed in all local dataspaces of the agents in A. We define it by induction on the structure of A and q:

$$\begin{array}{ll} [P,d] \Leftarrow \varepsilon & = [P,d] \\ [P,d] \Leftarrow (q \odot x{:}s) & = [P, d \oplus x] \Leftarrow q \\ (A \parallel A) \Leftarrow q & = (A \Leftarrow q) \parallel (A \Leftarrow q)\,. \end{array}$$

The set DQ of queues can be turned into a meet-semilattice by defining a prefix order as follows: $q_1 \sqsubseteq q_2$ if and only if there exists $q \in DQ$ such that $q_1 \odot q = q_2$ [13]. If every broadcastable message has the same sort then the above order coincides with the usual prefix ordering, while if they have all a different sort then the order coincides with the usual multiset inclusion ordering. For q_1 and q_2 in DQ, we denote by $q_1 \sqcap q_2$ their greatest lower bound.

Next we define an encoding $\mathcal{E}_{LG}\colon \mathit{Conf}_L \to \mathit{Conf}_G$ mapping configurations of the locally delayed architectures into corresponding configurations of the globally delayed architectures:

$$
\begin{aligned}
\mathcal{E}_{LG}([P, d, q]) &= [P, d], q \\
\mathcal{E}_{LG}([P, d, q] \parallel C) &= (([P, d] \Leftarrow q_P) \parallel (A' \Leftarrow q_C)), q \sqcap q'
\end{aligned}
$$

where $A', q' = \mathcal{E}_{LG}(C)$, $(q \sqcap q') \odot q_P = q$, and $(q \sqcap q') \odot q_C = q'$. In other words, we construct a shared queue as the smallest queue among all agents, and flush in the dataspace of each agent the messages that are in its queue but not in the shared one. This encoding is used to prove the following equivalence result.

Theorem 1. *For any configuration $C \in \mathit{Conf}_L$ of the locally delayed architecture with agents containing processes without 'del' and 'get' operators, the following holds:*

1. *for all $C' \in \mathit{Conf}_L$, if $C \longrightarrow\!\!\!\!\!\rightarrow_L C'$ then $\mathcal{E}_{LG}(C) \Longrightarrow_G \mathcal{E}_{LG}(C')$,*
2. *for all $C' \in \mathit{Conf}_G$, if $\mathcal{E}_{LG}(C) \longrightarrow\!\!\!\!\!\rightarrow_G C'$ then there exists $C'' \in \mathit{Conf}_L$ such that $C \Longrightarrow_L C''$ and $\mathcal{E}_{LG}(C'') = C'$.*

To prove the equivalence between the globally delayed and the undelayed architectures, we define, for $A \in \mathit{Agents}$ and $q \in DQ$ the following encoding:

$$
\mathcal{E}_{GU}(A, q) = A \Leftarrow q\,.
$$

Hence we eliminate the shared queue by flushing its pending messages in each local dataspace of A.

Theorem 2. *For any configuration $C \in \mathit{Conf}_G$ of the globally delayed architecture with agents containing processes without 'del' and 'get' operators, the following holds:*

1. *for all $C' \in \mathit{Conf}_G$, if $C \longrightarrow\!\!\!\!\!\rightarrow_G C'$ then $\mathcal{E}_{GU}(C) \Longrightarrow_U \mathcal{E}_{GU}(C')$,*
2. *for all $A \in \mathit{Agents}$, if $\mathcal{E}_{GU}(C) \longrightarrow\!\!\!\!\!\rightarrow_U A$ then there exists $C' \in \mathit{Conf}_G$ such that $C \Longrightarrow_G C'$ and $\mathcal{E}_{GU}(A) = C'$.*

Finally, we show that in the presence of only *wrt*, *put* and *rd* operators the multiplicity of data is insignificant. We show it only for the locally delayed architecture, but similar results hold also for the other two architectures. For a configuration $C \in \mathit{Conf}_L$ with dataspaces interpreted as multisets, define a flattening function flat_L that returns a configuration obtained by replacing each dataspace d with its associated set $\overline{d}$, where $\overline{d}(x) = 1$ if and only if $d(x) > 0$.

Theorem 3. *Let $C \in Conf_L$ be a configuration of the locally delayed architecture containing agents without 'del' and 'get' operations, and such that each dataspace in C is interpreted as multiset. The following holds:*

1. *for all $C' \in Conf_L$, if $C \longrightarrow\!\!\!\!\!\rightarrow_L C'$ then $flat(C) \longrightarrow\!\!\!\!\!\rightarrow_L flat(C')$.*
2. *for all $C' \in Conf_L$, if $flat(C) \longrightarrow\!\!\!\!\!\rightarrow_L C'$ then $C \longrightarrow\!\!\!\!\!\rightarrow_L C''$ with $C' = flat(C'')$.*

5.2 Consuming data, locally

Next we allow agents to contain processes with the local delete operator *get*. First we observe that the equivalence in Theorem 3 does not hold anymore. Consider an agent executing the process

$$P = wrt(a).wrt(a).get(a).get(a).wrt(b).0\,.$$

If the structure of the dataspace is a set then an agent executing P in any of our three architectures can never produce a b because it blocks at the second $get(a)$. However, if the dataspace is a multiset then the agent will always insert b in its store. Note that the difference holds also if we replace the *wrt* operations in P by *put* operations.

If dataspaces are sets then the locally delayed architecture is different from the globally delayed. Consider the following processes:

$$P = put(a{:}s).put(a{:}s).put(b{:}s).get(c).get(a).get(a).wrt(d).0$$
$$Q = get(b).put(c{:}t).0\,,$$

where $t \neq s$ (for example because the architecture uses a broadcast mechanism which guarantees that data from the same agent is received by all other agents in the same order as it was produced). Assume P is executed by an agent with a dataspace containing no a and c, while Q is executed by an agent with a dataspace containing no b. Because $t \neq s$, the agent executing P in the locally delayed system may receive c from the agent executing Q before the second a will be stored in its local dataspace. Hence the operation $wrt(d)$ may succeed. This is not the case if the two agents are executed in the globally delayed architecture because when c is broadcast, all agents in the system have already received the value b and hence also the two a's. Because of the set-structure, only one $get(a)$ may now be executed by the process P. The same result holds even if we do not allow for an agent to delete data not produced by itself, since we can replace $get(b)$ by $rd(b)$ in the process Q, and $get(c)$ by $rd(c)$ in the process P.

If dataspaces are multisets then we still have an equivalence between the locally delayed and the globally delayed architectures.

Theorem 4. *If dataspaces are multisets then for any configuration $C \in Conf_L$ of the locally delayed architecture with agents containing processes without the 'del' operator, the following holds:*

1. *for all $C' \in Conf_L$, if $C \longrightarrow\!\!\!\!\!\rightarrow_L C'$ then $\mathcal{E}_{LG}(C) \Longrightarrow_G \mathcal{E}_{LG}(C')$,*

2. *for all* $C' \in Conf_G$*, if* $\mathcal{E}_{LG}(C) \longrightarrow\!\!\!\!\!\rightarrow_G C'$ *then there exists* $C'' \in Conf_L$ *such that* $C \Longrightarrow_L C''$ *and* $\mathcal{E}_{LG}(C'') = C'$.

If dataspaces are sets, then both the locally and the globally delayed architectures are different from the undelayed one, without any assumption on the sorts of the broadcastable messages. Consider the process

$$P = put(a{:}s).put(a{:}t).get(a).get(a).wrt(b).0\,.$$

An agent executing P with a dataspace containing no a in the undelayed architecture cannot produce b, as there is no a present in its dataspace after the execution of the first $get(a)$. This is not the case for the other two architectures, due to the delay in receiving the message $a{:}t$.

Finally, if dataspaces are multisets and agents do not use the asynchronous global delete operator *del* then the three architectures are all equivalent.

Theorem 5. *If dataspaces are multisets then for any configuration* $C \in Conf_G$ *of the globally delayed architecture with agents containing processes without the 'del' operator the following holds:*

1. *for all* $C' \in Conf_G$*, if* $C \longrightarrow\!\!\!\!\!\rightarrow_G C'$ *then* $\mathcal{E}_{GU}(C) \Longrightarrow_U \mathcal{E}_{GU}(C')$*,*
2. *for all* $A \in Agents$*, if* $\mathcal{E}_{GU}(C) \longrightarrow\!\!\!\!\!\rightarrow_U A$ *then there exists* $C' \in Conf_G$ *such that* $C \Longrightarrow_G C'$ *and* $\mathcal{E}_{GU}(C') = A$.

5.3 Consuming data, globally

Finally we allow agents to contain processes with the asynchronous operator *del* for the global consumption of data.

Regardless of the interpretation of dataspaces as sets or multisets, the undelayed architecture is different from the other two architectures. Consider the process

$$P = put(a{:}s).del(a{:}t).del(a{:}r).wrt(b)\,, \tag{1}$$

executed by an agent with a dataspace containing no a. In the undelayed architecture this agent cannot produce a b, while in the other two architectures this may happen due to the delay in receiving the first message for the deletion of a.

Also the locally delayed architecture is different from the globally delayed one, regardless of the structure of the dataspaces (set or multiset). Consider the following three processes:

$$\begin{aligned} P &= put(a{:}s).del(a{:}t).put(b{:}t).0 \\ Q &= rd(b).put(c{:}r).0 \\ R &= rd(c).rd(a).wrt(d).0\,, \end{aligned}$$

where $t \neq r$. Assume P is executed by an agent with a dataspace containing no a, Q by an agent with a dataspace containing no b, and R by an agent with a dataspace containing no a and c. When c is broadcast in the globally delayed system, the value a has already been stored in all dataspaces, and consumed from them because b is guaranteed to be received after $\overline{a}$. Hence the agent executing R cannot produce d. In the locally delayed architecture, d may be produced by the agent executing R, because c may be stored in its dataspace before b and $\overline{a}$. The above difference remains if we substitute *get* actions for *rd* actions.

6 Conclusions and related work

We have presented three different architectures for coordination languages with primitives for producing data (locally or globally), for consuming data (locally or globally), and for testing the presence of data (locally). We have characterized equivalences and differences among the architectures by pointing out the maximal set of coordination primitives under which the architectures are equivalent. The next table gives an overview of the results in this paper. It is split in two parts, depending on the structure of dataspaces (set or multiset).

	set			*multiset*		
	$L-G$	$L-U$	$G-U$	$L-G$	$L-U$	$G-U$
rd, put, wrt	$=$	$=$	$=$	$=$	$=$	$=$
rd, put, wrt, get	$\neq$	$\neq$	$\neq$	$=$	$=$	$=$
rd, put, wrt, get, del	$\neq$	$\neq$	$\neq$	$\neq$	$\neq$	$\neq$
rd, put, wrt, get, del_L	$\neq$	$\neq$	$\neq$	$\neq$	$\neq$	$\neq$
rd, put, wrt, get, del_G	$\neq$	$\neq$	$\neq$	$\neq$	$\neq$	$=$

Here L stands for the locally delayed architecture, G for the globally delayed and U for the undelayed one; del_L is the global delete specified in the Appendix by replacing $(L5)$ with $(L5')$, and del_G is the global delete specified in the Appendix by replacing $(G5)$ with $(G5')$ (regardless whether we replace $(L5)$ by $(L5')$).

The result we present here continues the program started by the authors in [7]. In that paper the authors studied a synchronous global delete operator and negative tests for the locally delayed architecture and also for two other architectures based on a shared dataspace. In this paper, we concentrate on operators that are either local or realized via broadcast mechanisms. Moreover, here we separate the description of the software components from the specification of the software architectures. This is a necessary step towards a formal comparison among architectures.

In [1] two possible implementations for the broadcast mechanism of the coordination language LO [2] are presented; the first one corresponds to the broadcast used in our undelayed architecture while the second coincides with that of the locally delayed one. The equivalence between the two implementations is shown by proving that they are both correct implementations of the broadcasting mechanism of LO. We strengthened this equivalence result by presenting a third equivalent broadcast mechanism (the globally delayed one). Furthermore we prove that the equivalence holds because no global consuming operators are considered and because dataspaces have a multiset structure rather than a set structure.

In [9] different implementations of an output operator have been studied in the setting of Linda. In particular, implementations are considered that are similar to our undelayed and globally delayed with the synchronous global delete specified in the Appendix. A formal comparison between the above two implementations is presented in [10] where a simple Linda calculus is proven to be Turing powerful only under the undelayed implementation but not under the globally delayed one. The difference stems from the presence in Linda of operators that permit to test for the absence of data.

We leave for future work the investigation of negative test operators [8, 15] in combination with the locally and the asynchronous delete operators studied in this paper. In [7] some results have already been obtained for negative tests together with globally synchronous delete operators. We also plan to investigate other operators like one that spawns new agents, other forms of consumption operations (like an asynchronous version of the local delete), and also other forms of broadcast, such as the one that preserves the causal dependencies.

References

1. J.-M. Andreoli, L. Leth, R. Pareschi, and B. Thomsen. True Concurrency Semantics for a Linear Logic Programming Language with Broadcast Communication. In *Proc. of TAPSOFT'93*, volume 668 of *LNCS*, pages 182–198, Springer, 1993.
2. J.-M. Andreoli and R. Pareschi. Linear objects: Logical processes with built-in inheritance. *New Generation Computing*, 9(3+4):445–473, 1991.
3. F. Arbab, J.W. de Bakker, M.M. Bonsangue, J.J.M.M. Rutten, A. Scutellá, and G. Zavattaro. A transition system semantics for the control-driven coordination language MANIFOLD. To appear in *Theoretical Computer Science*, 1999.
4. F. Arbab, I. Herman, and P. Spilling. An overview of Manifold and its implementation. *Concurrency: Practice and Experience*, 5(1):23–70, 1993.
5. M. Boasson. Control systems software. In *IEEE Transactions on Automatic Control* 38:7, pages 1094–1107, 1993.
6. M.M. Bonsangue, J.N. Kok, M. Boasson, and E. de Jong. A software architecture for distributed control systems and its transition system semantics. In *Proc. of ACM Symp. on Applied Computing (SAC'98)*, pages 159–168. ACM press, 1998.
7. M.M. Bonsangue, J.N. Kok, and G. Zavattaro. Comparing coordination models based on shared distributed replicated data. In *Proc. of ACM Symp. on Applied Computing (SAC'99)*. ACM press, 1999.
8. A. Brogi and J.-M. Jacquet. On the expressiveness of Linda-like concurrent languages In I. Castellani and C. Palamidessi editors, *Proc. of Express'98*, volume 16(2) of *Electronic Notes in Theoretical Computer Science*, 1998.
9. N. Busi, R. Gorrieri, and G. Zavattaro. Comparing Three Semantics for Linda-like Languages. To appear in *Theoretical Computer Science*, 1999.
10. N. Busi, R. Gorrieri, and G. Zavattaro. On the Expressiveness of Linda Coordination Primitives. To appear in *Information and Computation*, 1999.
11. N. Busi, R. Gorrieri, and G. Zavattaro. A Process Algebraic View of Linda Coordination Primitives. In *Theoretical Computer Science*, 192(2): 167–199, 1998.
12. N. Carriero and D. Gelernter. Linda in context. In *Communications of the ACM* 32:4, pages 444–458, 1989.

13. A. Mazurkiewicz. Trace theory. In W. Brauer et al., editors, *Petri Nets, Applications, and Relationship to other models of Concurrency*, volume 255 of LNCS, pages 279–324, Springer-Verlag, 1987.
14. D.E. Perry and A. L. Wolf. Foundations for the Study of Software Architecture. In *Software Engineering Notes*, ACM SIGSOFT vol. 17:4, pages 40–52, 1992.
15. G. Zavattaro. Towards a Hierarchy for Negative Test Operators for Generative Communication. In I. Castellani and C. Palamidessi editors, *Proc. of Express'98*, volume 16(2) of *Electronic Notes in Theoretical Computer Science*, 1998.

A Alternative global delete operators

The *del* operation we have considered has an asynchronous behavior with respect to both the dataspace of the agent executing it, and also with respect to the dataspaces of all other agents. We could specify a more synchronous behavior of *del* by considering the following rules $(L5')$ and $(G5')$, instead of $(L5)$ and $(G5)$, respectively.

$$\boxed{\begin{array}{l} (L5')\ [del(a{:}s).P, d, q] \xrightarrow{\overline{a}:s} [P, d \oplus \overline{a}, q] \text{ if } in(d,a) = tt \\ (G5')\ [del(a{:}s).P, d], q \xrightarrow{\overline{a}:s} [P, d \oplus \overline{a}], q \text{ if } in(d,a) = tt \end{array}}$$

Rule $(L5')$ ensures synchrony with respect to the local dataspace in the locally delayed architecture. For the globally delayed architecture, rule $(G5')$ ensures (with rule $(G10)$) that an instance of a is removed from the local dataspace and also from the space of the other agents (if available) in one atomic step.

If we consider the new rule $(L5')$ for the locally delayed architecture, it is not hard to prove that the three architectures are still different, regardless of the interpretation of the dataspaces as sets or multisets.

More interesting is the analysis of the globally delayed architecture with the new synchronous global delete. Indeed, even if it remains different with respect to the locally delayed architecture (regardless of the use of the rule $(L5)$ or $(L5')$) we obtain an equivalence result.

Theorem 6. *If dataspaces are multisets then for any configuration $C \in Conf_G$ of the globally delayed architecture with a 'del' operator specified by replacing $(G5)$ with $(G5')$ the following holds:*

1. *for all $C' \in Conf_G$, if $C \longrightarrow\!\!\!\!\rightarrow_G C'$ then $\mathcal{E}_{GU}(C) \Longrightarrow_U \mathcal{E}_{GU}(C')$,*
2. *for all $A \in Agents$, if $\mathcal{E}_{GU}(C) \longrightarrow\!\!\!\!\rightarrow_U A$ then there exists $C' \in Conf_G$ such that $C \Longrightarrow_G C'$ and $\mathcal{E}_{GU}(A) = C'$.*

The same result does not hold if dataspaces are sets. Consider the program

$$P = put(a{:}s_1).put(a{:}s_2).del(a{:}s_3).del(a{:}s_4).wrt(b)\,.$$

If P is executed by an agent with no a in its dataspace then, in the undelayed architecture, it cannot produce b. On the other hand, in the globally delayed architecture, P can produce b due to the delay in receiving the second instance of the datum a.