Comparing coordination models based on shared distributed replicated data

Marcello M. Bonsangue and Joost N. Kok
Leiden Institute of Advanced Computer Science, Leiden University, The Netherlands
e-mail: `marcello@cs.leidenuniv.nl` `joost@cs.leidenuniv.nl`

Gianluigi Zavattaro
Department of Computer Science, Bologna University, Italy
e-mail: `zavattar@cs.unibo.it`

Keywords: Coordination models, software architectures, distributed dataspace, shared dataspace, transition system semantics.

Abstract

We present three models for a data-oriented coordination language and show several relationships between the models. We consider a distributed model and two centralized shared dataspace models, one preserving the order in which data is produced and a second one not necessarily preserving this order. For example, we show that if dataspaces are multisets then the distributed and the centralized systems are observationally equivalent for any program using read, write and delete operations. The same result does not hold if dataspaces are sets or databases.

1 Introduction

Coordination languages consist of a number of primitives that can be added to programming languages such that a number of programs, running in parallel, can coordinate their activities. Usually coordination is achieved using a number of primitives like read, write, delete and test operations that work on shared data. Synchronization and communication happen through these primitives. Read and delete operations are potentially blocking, while write operations always succeed, that is, a process can always execute a write statement and can continue immediately with the execution of other statements.

In practice, one often wants to distribute the dataspace. One of the possibilities is to duplicate the shared data among the different processes. Each process can then do locally read operations, while write operations correspond to broadcasting of data. In this paper we look at the consequences of duplicating the data among processes.

As starting point we consider three different system architectures. One is distributed while the other two are centralized. In the distributed system (denoted by D) each process is associated with an agent. Each agent has its own local dataspace for storing data. Communication between agents is established by a message passing mechanism using a protocol that does not guarantee that all processes receive data values at the same time and in the same order as they were produced.

In one of the centralized systems (denoted by O) there is one (non-replicated) dataspace shared among all processes. Communication between processes happens only through the shared dataspace, and the order in which data is produced is preserved. The other centralized system (denoted by U) is similar to the previous one, but the order in which data values are produced is not necessarily preserved.

The above three systems are parameterized by the structure of the dataspaces. More concretely, we will consider three ways of presenting data, thus obtaining in total nine different models. When multiplicity of data is significant because data is seen as resource then dataspaces are multisets (denoted by m). When multiplicity is insignificant because data is seen as information then dataspaces are sets (denoted by s). Finally, when data is sorted and the sort (index) is significant then we can see dataspaces as databases (denoted by db), in which at most one data value per sort can be present.

Each of the nine models can be coded by two letters: for example the centralized system with ordered writing, where the shared dataspace is a multiset, is coded by (O, m). Several well-known coordination languages can be abstractly classified in these models: LINDA [15, 13, 7] and its dialect studied in [12] correspond to (O, m), LINDA with an unordered *out* [7] and SPA without local delete [14] correspond to (U, m), SPLICE without local delete [3, 5] to (D, db), centralized and distributed database systems [10] correspond to (O, db) and (D, db), respectively, and concurrent constraint programming [17] and non-monotonic concurrent constraint programming [11] to (O, s).

In this paper we relate these models. There are two kinds of results. When we show an equivalence between two models, we prove it under a strong notion of observable equivalence obtained by requiring the existence of an encoding of one model into the other preserving all program constructs and such that each step in one model can be simulated by the other one in zero or more steps, and vice-versa. In this way we abstract from internal communication steps typical for one model but not fot the other. If we show that two models are distinct then we prove it under a weak notion of observable equivalence based on the fact that the same program can produce a data value in one model but not in the other model. The reason why we consider only production

	(D,m)	(O,m)	(U,m)	(D,s)	(O,s)	(U,s)	(D,db)	(O,db)	(U,db)
(D,m)		rwd	$rwdt$	rwt	rw	rwt	$\times$	$\times$	$\times$
(O,m)			rwd	rw	rwt	rw	$\times$	$\times$	$\times$
(U,m)				rwt	rw	rwt	$\times$	$\times$	$\times$
(D,s)					rw	rwt	$\times$	$\times$	$\times$
(O,s)						rw	$\times$	$\times$	$\times$
(U,s)							$\times$	$\times$	$\times$
(D,db)								$\times$	$\times$
(O,db)									$\times$
(U,db)									

Table 1: Overview of results

of data values to distinguish the models is because we are dealing with an asynchronous calculus: an observer cannot know if a broadcast value has already been received [1].

These relationships give insight in the design choices and the language constructs one has at its disposal when designing coordination languages. For a distributed system it may be interesting to find an equivalent centralized system allowing for easier reasoning about programs. Conversely, one might want to implement a centralized model by a distributed one.

Table 1 summarizes the results of this paper. The $\times$ symbol states that there is a program with read and write operations such that its operational semantics in the two systems are different. We use the symbols r, w, d, t for denoting programs that use read, write, delete and test-for-absence operators, respectively. In the table a string of those symbols denotes the maximal set of operators that a program may have such that its interpretations in the two systems are equivalent (up to internal steps).

For example, if dataspaces are multisets then, for any program P, the distributed system is equivalent to the centralized unordered system. However the distributed system is equivalent to the centralized ordered one only for programs not containing the testing-for-absence operator.

The use of a delete operator makes these three systems all different when we interpret dataspaces as sets. Furthermore, if we consider only programs without this operator, then there is no observable difference between systems where dataspaces are multisets and those where dataspaces are sets.

Finally, in case we interpret dataspaces as databases, then all systems are different from each other, and also from the other interpretations if we consider programs with only read and write operations.

The structure of the paper is as follows. In the next section we introduce the syntax of an abstract coordination language. In Section 3 we introduce the three different systems for our language by means of labeled transition systems. Then we study the relationships between the three systems when data is treated as resource (Section 4), as information (Section 5) and when the sorts of data are significant (Section 6). In Section 7 and 8 we give alternative definitions of the testing-for-absence operator and of the delete operator, respectively, and compare the three systems according to these new operators. In Section 9 we add a coordination primitive to our language for the dynamical creation of processes. Finally we draw some conclusions in Section 10.

2 The language

We start with the introduction of an abstract coordination language. Let *Data*, ranged over by $a, b, \ldots$, be a set of *data values.* We define a set *Prog* of *programs* by means of the grammar

$$P \quad ::= \quad S \mid \ S \parallel P\,,$$

where S is a *sequential statement* in *Stat*. The set of sequential statements is defined by the grammar

$$\begin{array}{lcl} S & ::= & 0 \ \mid \ \alpha.S \ \mid \ S+S \\ \alpha & ::= & \tau \ \mid \ rd(a) \ \mid \ put(a) \ \mid \ del(a) \ \mid \ tfa(a)\,, \end{array}$$

where $a \in Data$. We will consider three different communication mechanisms between sequential statements. First we present a distributed architecture, where each process is associated with an agent which embodies a dataspace for storing data. Communication between agents is established by a message passing mechanism using a protocol which distributes and replicates data across the communication network. Second we consider a centralized architecture, where processes communicate via a shared dataspace and the communication between processes and the dataspace is asynchronous. Finally we consider another centralized architecture, where, as before, processes communicate via a shared dataspace, but the communication between processes and dataspace is synchronous. Informally, the primitive actions of our language are as follows: τ denotes some internal computation, $rd(a)$ reads a data element without consuming it, $put(a)$ produces a new data element, $del(a)$ reads and consumes a data element, and $tfa(a)$ tests for the absence of a data element. While the execution of the *put* operation does not depend on a context, the *rd* and *del* operations block in case the requested data element is not present, and the *tfa* operation blocks in case the requested data element is present. The *tfa* operator can be used to encode non-blocking versions of the *rd* and *del* operators [7, 18]. For example, we can model a non-blocking read operator *rdp* in the style of Linda as follows:

$$rdp(a)?S_1\_S_2 \quad = \quad (rd(a).S_1) + (tfa(a).S_2)$$

$(D1)$	$[\tau.S, d, q] \xrightarrow{\tau} [S, d, q]$	$(D2)$	$\dfrac{in(d,a) = tt}{[rd(a).S, d, q] \xrightarrow{\tau} [S, d, q]}$
$(D3)$	$[put(a).S, d, q] \xrightarrow{a} [S, d, a \odot q]$	$(D4)$	$\dfrac{C \xrightarrow{a} C'}{[S, d, q] \parallel C \xrightarrow{a} [S, d, a \odot q] \parallel C'}$
$(D5)$	$\dfrac{in(d,a) = tt}{[del(a).S, d, q] \xrightarrow{\overline{a}} [S, d \oplus \overline{a}, q]}$	$(D6)$	$\dfrac{C \xrightarrow{\overline{a}} C' \quad in(d,a) = tt}{[S, d, q] \parallel C \xrightarrow{\overline{a}} [S, d \oplus \overline{a}, q] \parallel C'}$
$(D7)$	$\dfrac{in(d,a) = ff}{[tfa(a).S, d, q] \xrightarrow{\tilde{a}} [S, d, q]}$	$(D8)$	$\dfrac{C \xrightarrow{\tilde{a}} C' \quad in(d,a) = ff}{[S, d, q] \parallel C \xrightarrow{\tilde{a}} [S, d, q] \parallel C'}$
$(D9)$	$[S, d, q \odot \sigma] \xrightarrow{\tau} [S, d \oplus \sigma, q]$	$(D10)$	$\dfrac{C \xrightarrow{\tau} C'}{[S, d, q] \parallel C \xrightarrow{\tau} [S, d, q] \parallel C'}$
$(D11)$	$\dfrac{[S, d, q] \xrightarrow{\ell} [S', d', q'] \quad S \neq S'}{[S + S'', d, q] \xrightarrow{\ell} [S', d', q']}$		

Table 2: Axioms and rules for the distributed system

If the value a is present in the (local) dataspace then the statement S_1 is executed, otherwise the statement S_2 is executed.

To model communication between processes we let

$$\Sigma = Data \cup \{\overline{a} \mid a \in Data\}$$

be a set of *messages*, ranged over by σ. The message $\overline{a}$ represents the request for deletion of the element a. We use the convention that $\overline{\overline{a}} = a$ for $a \in Data$. We assume that Σ comes equipped with a reflexive and symmetric relation

$$\bowtie \subseteq \Sigma \times \Sigma$$

relating messages with the same type of information, for example produced by the same process. We say that two messages have the same *sort* if they are related by $\bowtie$. This relation plays a role in the broadcasting of messages. We assume that reception of messages respects the order in which they were broadcast when they are of the same sort. Thus, two messages with different sorts (e.g. broadcast by different processes) may be received by another process in any order, while two messages with the same sort (e.g. broadcast by the same process) are received by any process in the same order as they were produced.

Following [4], we model broadcasting using *queues of pending messages*. They are congruence classes of finite strings in the monoid $(\Sigma^\star, \cdot, \varepsilon)$ modulo the least congruence such that, for all $\sigma_1, \sigma_2 \in \Sigma$

$$\sigma_1 \cdot \sigma_2 = \sigma_2 \cdot \sigma_1 \qquad \text{if } \sigma_1 \not\bowtie \sigma_2 .$$

We denote by DQ the set of all queues, with q as typical element. We write $\odot$ for string concatenation $\cdot$ modulo the above congruence, and denote a congruence class containing a single element by the element itself. Hence σ is the congruence class containing the one-element string $\sigma \in \Sigma$ and ε the congruence class containing the empty string.

When received, messages are stored in *dataspaces*. The structure of a dataspace strongly depends on how data is treated by the processes. For example, a multiset structure of the dataspace corresponds to the view of data as resource (multiplicity of data values is significant), while a set structure of the dataspace implies that data represents information (multiplicity of data values is insignificant). A refined structure of the latter is when the sort of the data becomes significant (no values of the same sort can be in a dataspace). We call this structure database, inspired by the fact that in a database each data element is an entity uniquely determined by its sort (also called index). We will define our three systems independently from the structure of the dataspaces used. Therefore we assume the existence of an abstract set DS of dataspaces, ranged over by d, which comes together with a predicate in for asserting the presence or absence of a message in a dataspace, and an operation $\oplus$ for inserting messages in a dataspace:

$$in\colon DS \times \Sigma \to \{tt, ff\} \quad \text{and} \quad \oplus\colon DS \times \Sigma \to DS .$$

We assume that $\perp \in DS$, where $\perp \notin \Sigma$ is intended to denote the empty dataspace. Hence we require $in(\perp, \sigma) = ff$ for all $\sigma \in \Sigma$. After the definition of the three systems we will consider concrete interpretations of dataspaces: as multisets, as sets and as databases.

3 Operational semantics

In this section we introduce three different models for our language based on transition systems.

3.1 The distributed system

In the distributed system we associate an agent to each sequential component of a program. An *agent* is a triple $[S, d, q]$ consisting of statement S, a dataspace $d \in DS$ and a queue of pending messages $q \in DQ$. The set $Conf_D$ of configurations of the distributed system is defined by the following grammar:

$$C \quad ::= \quad [S, d, q] \ \mid \ [S, d, q] \parallel C ,$$

where $S \in Stat$, $d \in DS$ and $q \in DQ$. Communication between agents is established by a broadcast mechanism: if

$(U1)$	$\langle \tau.S, d, q\rangle \longrightarrow \langle S, d, q\rangle$	$(U2)$	$\dfrac{in(d,a) = tt}{\langle rd(a).S, d, q\rangle \longrightarrow \langle S, d, q\rangle}$
$(U3)$	$\langle put(a).S, d, q\rangle \longrightarrow \langle S, d, a \odot q\rangle$	$(U4)$	$\langle P, d, q \odot \sigma\rangle \longrightarrow \langle P, d \oplus \sigma, q\rangle$
$(U5)$	$\dfrac{in(d,a) = tt}{\langle del(a).S, d, q\rangle \longrightarrow \langle S, d \oplus \overline{a}, q\rangle}$	$(U6)$	$\dfrac{in(d,a) = \mathit{ff}}{\langle tfa(a).S, d, q\rangle \longrightarrow \langle S, d, q\rangle}$
$(U7)$	$\dfrac{\langle S, d, q\rangle \longrightarrow \langle S', d', q'\rangle \quad S \neq S'}{\langle S + S'', d, q\rangle \longrightarrow \langle S', d', q'\rangle}$	$(U8)$	$\dfrac{\langle P, d, q\rangle \longrightarrow \langle P', d', q'\rangle}{\langle S \parallel P, d, q\rangle \longrightarrow \langle S \parallel P', d', q'\rangle}$

Table 3: Axioms and rules for the unordered system

an agent receives a data value from another agent, it stores the value in its local dataspace. Conversely, if a process produces a data element then its agent will forward a copy of the element to all other agents. There is no synchronization among the agents, thus each agent can receive the data element at a different moment in time. Furthermore, values of different sorts need not to be received by all agents in the same order.

The semantics of agents is presented in two steps. First we define a labeled transition relations which captures the dynamical behavior of agents. It uses labels from the set

$$\Lambda \quad = \quad \{\tau\} \cup \{a, \overline{a}, \tilde{a} \mid a \in Data\},$$

ranged over by ℓ. The symbols τ, $\overline{a}$ and $\tilde{a}$ are assumed to be not in $Data$. The labels in Λ are only necessary for a structural definition of the transition relation. In a second step we abstract from these labels and define another transition relation describing the reduction steps of the system.

The labeled transition relation $\longrightarrow_D$ is defined as the least relation in $Conf_D \times \Lambda \times Conf_D$ satisfying the axioms and rules given in Table 2, where the symmetric rules for the '+' and the '$\parallel$' constructs are omitted.

Rule $(D2)$ says that the *rd* action is blocking. It can be performed only if the data is present in the local dataspace. Rule $(D10)$ says that this action is local. The *put* action is not blocking (Axiom $(D3)$), and its effect is global: every agent receives the sent value in its queue of pending messages (Rule $(D4)$). The *del* action is also blocking (Rule $(D5)$) and its effect is to delete a value in every local dataspace (Rule $(D6)$). Another blocking operation is *tfa* (Rule $(D7)$) which succeeds only if the value requested is not present in the dataspaces of all agents in the system (Rule $(D8)$). Note that both *del* and *tfa* require a synchronization among all agents in the system. Axiom $(D9)$ shows that if a message is pending then it can be stored in the local dataspace. This action is local (Rule $(D10)$). Rule $(D11)$ is the usual one for nondeterministic choice. The condition $S \neq S'$ in its premise is necessary to ensure that the choice does not take place when a message is received (Axiom $(D9)$).

The labels in the transition relation $\longrightarrow_D$ have been introduced for a structural definition of its specification. However, when comparing the above distributed system with the other two centralized systems, we are only interested in the reduction steps. Therefore we introduce a transition relation $\longrightarrow\!\!\!\!\!\rightarrow_D \subseteq Conf_D \times Conf_D$ which abstracts from the labels in Λ. For all $C, C' \in Conf_D$, we define

$$C \longrightarrow\!\!\!\!\!\rightarrow_D C' \quad \text{if and only if} \quad \exists \ell \in \Lambda \,.\, C \stackrel{\ell}{\longrightarrow}_D C'. \qquad (1)$$

The transition relation $\longrightarrow\!\!\!\!\!\rightarrow_D$ is 'too strong' to be used for a comparison when we want to abstract from certain internal communication steps typical for one system but not for the other one. We define its *weak* correspondent as the reflexive and transitive closure of $\longrightarrow\!\!\!\!\!\rightarrow_D$, that is $\Longrightarrow_D = (\longrightarrow_D)^*$.

We say that $C \in Conf_D$ is an *initial configuration* of the distributed system if the dataspace of each agent in C contains no values (i.e. it is $\perp$) and the queue of each agent in C contains no pending messages (i.e. it equals ε). A program $P \in Prog$ is said to be *executed* in a configuration C of the distributed system if each sequential component of P is the statement of an agent in C, that is, P is executed in C if we can assign to each sequential statement of P a dataspace and a queue of pending messages in C. Finally, a configuration $C \in Conf_D$ is *reachable* if there exists a program P executed in an initial configuration C_0 such that $C_0 \Longrightarrow_D C$.

3.2 The centralized unordered system

Next we consider a centralized architecture. In this system each sequential component of a program communicates via a shared dataspace, where all data values that are produced are stored. The communication with the shared dataspace is asynchronous. Values do not become immediately visible to all processes once they are produced, but they may be delayed for some unpredictable time. As a consequence, the order in which data values of different sorts are produced is not necessarily preserved.

Let $Conf_U = Prog \times DS \times DQ$ be the set of all configurations of the centralized unordered system. Its dynamical behavior is described by means of a transition relation $\longrightarrow\!\!\!\!\!\rightarrow_U$, defined as the least relation in $Conf_U \times Conf_U$ satisfying the axioms and rules given in Table 3 (the symmetric rules for the '+' and the '$\parallel$' constructs are omitted).

The *rd*, *del* and *tfa* actions are blocking (Rules $(U2)$, $(U5)$ and $(U6)$, respectively). Their execution depends on the content of the shared dataspace and requires a direct synchronization with the shared dataspace. The only coordination action which is not blocking is *put* (Axiom $(U3)$). Axiom $(U4)$ shows that between the production of a value and its introduction in the shared dataspace there may be an unpredictable delay, while Rule $(U8)$ says that process communicate only through the shared dataspace. Finally, Rule $(U7)$ is for the nondeterministic choice, using a condition in the premise similar to the one for the previous system.

In order to abstract from certain internal communication steps typical for one system but not for the other one, we define a weaker transition relation $\Longrightarrow_U$ as the reflexive and

$(O1)$	$\langle \tau.S, d\rangle \longrightarrow \langle S, d\rangle$	$(O2)$	$\dfrac{in(d,a) = tt}{\langle rd(a).S, d\rangle \longrightarrow \langle S, d\rangle}$
$(O3)$	$\langle put(a).S, d\rangle \longrightarrow \langle S, d \oplus a\rangle$		
$(O4)$	$\dfrac{in(d,a) = tt}{\langle del(a).S, d\rangle \longrightarrow \langle S, d \oplus \overline{a}\rangle}$	$(O5)$	$\dfrac{in(d,a) = \mathit{ff}}{\langle tfa(a).S, d\rangle \longrightarrow \langle S, d\rangle}$
$(O6)$	$\dfrac{\langle S, d\rangle \longrightarrow \langle S', d'\rangle}{\langle S + S'', d\rangle \longrightarrow \langle S', d'\rangle}$	$(O7)$	$\dfrac{\langle P, d\rangle \longrightarrow \langle P', d'\rangle}{\langle S \parallel P, d\rangle \longrightarrow \langle S \parallel P', d'\rangle}$

Table 4: Axioms and rules for the ordered system

transitive closure of $\twoheadrightarrow_U$, that is $\Longrightarrow_U = (\twoheadrightarrow_U)^*$.

3.3 The centralized ordered system

Finally, we consider a second centralized architecture consisting of sequential programs running in parallel and sharing a dataspace. The difference with the previous system is that the communication with the shared dataspace is assumed to be synchronous, so that data elements are introduced directly in the dataspace at the moment of their production.

Let $Conf_O = Prog \times DS$ be the set of all possible configurations of the centralized ordered system. The behavior of programs in this system is given by the transition relation $\twoheadrightarrow_O$, defined as the least relation in $Conf_O \times Conf_O$ satisfying the axioms and rules given in Table 4, where the symmetric rules for the '$+$' and the '$\parallel$' constructs are omitted.

The ordered system is similar to the unordered system, the only difference being the Axiom $(O3)$ instead of $(U3)$ together with $(U4)$ which show that the insertion of a value in the shared dataspace happens when the value is produced.

As for the other two systems, we define a weaker transition relation $\Longrightarrow_O$ as the reflexive and transitive closure of $\twoheadrightarrow_O$, that is $\Longrightarrow_O = (\twoheadrightarrow_O)^*$.

4 Data as resource

If processes treat data as resource, multiplicity of values becomes significant. Hence dataspaces are multisets. Thus, in this section, we define $DS = \Sigma \to \mathbb{N}$,

$$in(d,\sigma) = tt \Leftrightarrow d(\sigma) > 0 \quad \text{and}$$
$$d \oplus \sigma \;=\; \begin{cases} d[\sigma/(d(\sigma)+1)] & \text{if } in(d,\overline{\sigma}) = \mathit{ff} \\ d[\overline{\sigma}/(d(\overline{\sigma})-1)] & \text{if } in(d,\overline{\sigma}) = tt\,, \end{cases}$$

where, for any function f, we use the notation $f[x/y]$ for denoting the function mapping x to y and otherwise acting as f. A data value a is inserted into a dataspace only if no deletion message $\overline{a}$ is stored in the dataspace. Otherwise the message is removed from the data space. Conversely, a data value a is removed from the dataspace if a message for its deletion arrives. In case the message arrives and the data is not present in the dataspace then the message is stored into the dataspace.

First we show that the distributed system and the centralized unordered system are both observationally different from the centralized ordered one if we use tfa operators. Consider the program

$$P \;=\; put(a).tfa(a).put(b).0\,. \tag{2}$$

The above program executed in the ordered system with a dataspace containing no a cannot produce the value b, while the same program may terminate its execution in any of the other two systems with the same dataspace because it may take some time before a is stored in the dataspace.

Next we show that the distributed system and the centralized unordered system are equivalent. First we need to introduce some notation.

Given two multisets $d_1, d_2 \in DS$ their least upper bound with respect to multiset inclusion is defined, for $\sigma \in \Sigma$, by

$$d_1 \sqcup d_2(\sigma) \;=\; \max\{d_1(\sigma), d_2(\sigma)\}\,.$$

We need also to define the greatest lower bound of two queues of pending messages. We say that a queue $q_1 \in DQ$ is a prefix of a queue $q \in DQ$, denoted by $q_1 \sqsubseteq q$, if there exists a queue q_2 such that $q_1 \odot q_2 = q$. For example, if $a_1 \bowtie a_2$ and $a_1 \not\bowtie b$ then $b \sqsubseteq a_1 \odot b \odot a_2$ because $b \odot a_1 \odot a_2 = a_1 \odot b \odot a_2$. The $\sqsubseteq$ is a partial order on DQ such that every two queues q_1 and q_2 in DQ have a greatest lower bound, denoted by $q_1 \sqcap q_2$. If $\bowtie$ is a transitive relation, then $q_1 \sqcap q_2$ is the set of the longest prefixes of the same sort of the strings in q_1 and q_2. For example, let $a_1 \bowtie a_2$, $b_1 \bowtie b_2$ and $a_1 \not\bowtie b_1$. If $q_1 = a_1 \odot b_1 \odot b_2$, $q_2 = b_1 \odot a_1 \odot a_2$ and $q_3 = a_1 \odot b_2$ then $q_1 \sqcap q_2 = a_1 \odot b_1$ and $q_1 \sqcap q_3 = a_1$.

We are now ready for defining inductively an encoding $\mathcal{E}_{DU}$ mapping configurations of the distributed system to corresponding configurations of the centralized unordered system:

$$\begin{aligned} \mathcal{E}_{DU}([S,d,q]) &= \langle S, d, q\rangle \\ \mathcal{E}_{DU}([S,d,q] \parallel C) &= \langle S \parallel P_C, d \sqcup d_C, q \sqcap q_C\rangle\,, \end{aligned}$$

where $\langle P_C, d_C, q_C\rangle = \mathcal{E}_{DU}(C)$. Thus we construct a shared dataspace as the local dataspace of the agent in C which has stored the largest amount of data, and we construct a queue of pending messages by taking the smallest queue of pending messages among all agents in C.

The following theorem states the observable equivalence between the distributed and the centralized unordered system.

Theorem 4.1 *Let $C \in Conf_D$ be a reachable configuration of the distributed system. The following holds:*

1. *for all $C' \in Conf_D$, if $C \twoheadrightarrow_D C'$ then $\mathcal{E}_{DU}(C) \Longrightarrow_U C''$ with $\mathcal{E}_{DU}(C') = C''$;*

2. *for all $C' \in Conf_U$, if $\mathcal{E}_{DU}(C) \longrightarrow\!\!\!\!\!\rightarrow_U C'$ then $C \Longrightarrow_D C''$, with $\mathcal{E}_{DU}(C'') = C'$.*

The operations modifying dataspaces that we consider in this paper are all global. Hence, if a data value is in the dataspace or in the queue of one agent, then it must be also present either in the dataspace or in the queue of any other agent. That is, agents in the system must have a consistent view of the same distributed replicated data values. This explains the condition on the reachability of C in the above theorem. For example, the configuration

$$[del(a).0, \{\!|a|\!\}, \varepsilon] \parallel [0, \perp, a]$$

is reachable but the configuration

$$[del(a).0, \{\!|a|\!\}, \varepsilon] \parallel [0, \perp, \varepsilon]$$

is not because the rightmost agent will never receive the value a in its dataspace even if, according to the leftmost agent, it has already been produced.

We obtain as corollary of the above theorem that for each program $P \in Prog$, executed in the initial configuration of the distributed system is weak bisimilar ([16]) to the configuration $\langle P, \perp, \varepsilon\rangle$ of the unordered system.

The program in (2) containing the *tfa* operator distinguishes the centralized ordered system from the centralized unordered one. Next we prove that if we consider only programs that do not use this operator, then the two systems are observationally equivalent.

First we define an encoding $\mathcal{E}_{UO}$ mapping configurations of the centralized unordered system into configurations of the centralized ordered one. For each configuration $\langle P, d, q\rangle$ of the unordered system we define the corresponding configuration $\mathcal{E}_{UO}(\langle P, d, q\rangle)$ in the ordered system by induction on the length of the queue q:

$$\begin{aligned} \mathcal{E}_{UO}(\langle P, d, \varepsilon\rangle) &= \langle P, d\rangle \\ \mathcal{E}_{UO}(\langle P, d, (q \odot \sigma)\rangle) &= \mathcal{E}_{UO}(\langle P, (d \oplus \sigma), q\rangle\,. \end{aligned}$$

In other words, we flush all messages of the queue in the shared dataspace. Note that even if $q \odot \sigma = q' \odot \sigma'$, the above encoding is a function because q and q' are queues of finite length and $(d \oplus \sigma) \oplus \sigma' = (d \oplus \sigma') \oplus \sigma$ for any multiset $d \in DS$.

We have the following equivalence result.

Theorem 4.2 *Let $C = \langle P, d, q\rangle \in Conf_U$ be such that the program P does not contain the 'tfa' operator. The following holds:*

1. *for all $C' \in Conf_U$, if $C \longrightarrow\!\!\!\!\!\rightarrow_U C'$ then $\mathcal{E}_{UO}(C) \Longrightarrow_O C''$ with $\mathcal{E}_{UO}(C') = C''$;*

2. *for all $C' \in Conf_O$, if $\mathcal{E}_{UO}(C) \longrightarrow\!\!\!\!\!\rightarrow_O C'$ then $C \stackrel{\ell}{\Longrightarrow}_U C''$ with $\mathcal{E}_{UO}(C'') = C'$.*

An immediate consequence of the above theorem and Theorem 4.1 is that the distributed system and the centralized ordered one are observationally equivalent for programs not containing the *tfa* operator. Furthermore, if P is a program of that kind then $\langle P, \perp, \varepsilon\rangle$ and $\langle P, \perp\rangle$ are weak bisimilar.

5 Data as information

When data represents information then its multiplicity is insignificant. Hence dataspaces are sets. In this case we define DS as the set of all subsets of Σ,

$$\begin{array}{ll} in(d,\sigma){=}tt \Leftrightarrow \sigma \in d & \text{and} \\ d \oplus \sigma \;\; = \begin{cases} d \cup \{\sigma\} & \text{if } in(d, \overline{\sigma}) = f\!f \\ d \setminus \{\overline{\sigma}\} & \text{if } in(d, \overline{\sigma}) = tt. \end{cases} \end{array}$$

As for the case of dataspaces as multisets, a data value a is inserted into a dataspace only if no request for deletion $\overline{a}$ is stored in the dataspace. Otherwise the message is removed from the dataspace. Conversely, a value a is removed from the dataspace if a message for its deletion arrives. In case the message arrives and the value is not present in the dataspace then the message is stored in the dataspace.

The program given in (2) shows that if we use the *tfa* operator then the distributed and the centralized unordered systems are both observationally different from the centralized ordered system when dataspaces are sets.

Next we give a program with the *del* operator that shows that both the distributed system and the centralized unordered system are observationally different from the centralized ordered system. Take the program

$$P \;\; = \;\; put(a).put(a).del(a).del(a).put(b).0\,. \qquad (3)$$

If P is executed in the ordered system then it cannot produce the value b because it blocks at the second delete as there is no a in the dataspace. However, if P is executed in the centralized unordered system or in the distributed system it can either block at the second delete or it can happen that the second a is received by the dataspace only after the first one already has been deleted, and thus P can terminate its execution by producing the value b.

The same program P shows also an observational distinction between the three systems with dataspaces as multisets and those with dataspaces as sets. Indeed, regardless of the system, if dataspaces are multisets then P will always produce the value b and will never block at the second delete.

The next program shows that the distributed system is also different from the centralized unordered system when we allow programs with the *del* operator. Let $P = S_1 \parallel S_2$, where

$$\begin{array}{l} S_1 = put(a_1).put(a_1).put(a_2).rd(a_2).del(a_1).put(b).0 \\ S_2 = rd(b).rd(a_1).put(c).0\,, \end{array} \qquad (4)$$

and $a_1 \bowtie a_2$ (for example because the system uses a communication protocol which guarantees that data from the same producer is received by all processes in the same order as it was produced). If P is executed in the centralized unordered system with a dataspace containing no a_2 and b, then S_2 blocks in the execution of its $rd(a_1)$ operation because both the instances of a_1 have reached the shared dataspace at the moment in which program S_1 performs its $del(a_1)$ operation. Hence the value c cannot be produced. On the other hand, if P is executed in the distributed system with dataspaces containing no a_2 and b, then the agent executing S_2 may not block in the execution of its $rd(a_1)$ operation, for example when $del(a_1)$ is executed by S_1 when only one instance of a_1 has been stored in the dataspace of S_2. Hence P can produce the value c.

The programs in (3) and (4) show that in presence of the *del* operator, the three systems are all observationally different from each other when dataspaces are sets. It is interesting to observe that in the absence of the *del* operator there is no observable difference between dataspaces as multisets or as sets. To prove this fact we need to introduce three flattening functions (one for each system) which transform terms with multisets into terms with sets. For $S \in Stat$, $P \in Prog$, $q \in DQ$ and a multiset $d \in DS$ define

$$\begin{array}{lcl} flat_D([S,d,q]) & = & [S,\overline{d},q] \\ flat_D([S,d,q] \parallel C) & = & [S,\overline{d},q] \parallel flat_D(C) \\ flat_U(\langle P,d,q\rangle) & = & \langle P,\overline{d},q\rangle \\ flat_O(\langle P,d\rangle) & = & \langle P,\overline{d}\rangle\,, \end{array}$$

where $\overline{d} = \{\sigma \mid d(\sigma) > 0\}$ is the set associated to d.

Theorem 5.1 *Let $C \in Conf_D$ be a configuration of the distributed system such that it does not contain the 'del' operator, and the dataspace of each agent in C is a multiset. The following holds:*

1. *for all $C' \in Conf_D$ such that all its dataspaces are multisets, if $C \longrightarrow\!\!\!\!\!\rightarrow_D C'$ then $flat_D(C) \longrightarrow\!\!\!\!\!\rightarrow_D flat_D(C')$;*
2. *for all $C' \in Conf_D$ such that all its dataspaces are sets, if $flat_D(C) \longrightarrow_D C'$ then $C \longrightarrow_D C''$ for $flat_D(C'') = C'$.*

A similar theorem holds also for the centralized ordered system and the centralized unordered system, using the flattening $flat_O$ and $flat_U$, respectively. It follows from Theorems 4.1 and 4.2 that if dataspaces are sets then the distributed system is observationally equivalent to the centralized unordered one for programs not using the *del* operator, and all three systems are observationally equivalent for programs not using the *del* and *tfa* operators.

6 Data as entity

An entity can be uniquely identified and distinguished from other entities. In this section we consider data as entity identified by its sort. Thus the sort of data is significant: at most one value per data sort may be present in a dataspace. This implies that dataspaces are special kinds of sets. Furthermore, the operation of insertion of messages is different from the one used in the case of data as information. We define DS to be the set of all subsets of Σ containing no pair of messages of the same sort, and

$$in(d,\sigma)=tt \Leftrightarrow \sigma \in d \quad \text{and}$$
$$d \oplus \sigma = \begin{cases} (d \setminus \{\sigma' \mid \sigma' \bowtie \sigma\}) \cup \{\sigma\} & \text{if } in(d,\overline{\sigma}) = ff \\ d \setminus \{\overline{\sigma}\} & \text{if } in(d,\overline{\sigma}) = tt. \end{cases}$$

When a message is inserted in a dataspace it overwrites all other messages of the same sort already present in the dataspace. In this way, at most one message for each sort is present in the dataspace also after insertion of a new message.

First we show that the interpretation of dataspaces as databases is observationally different from the set and multiset interpretations. Consider the program

$$P \;=\; put(a_1).put(a_2).rd(a_1).put(b).0\,, \tag{5}$$

where $a_1 \neq a_2$ and $a_1 \bowtie a_2$. Regardless of the system in which we execute the program P, if dataspaces are multisets initially containing no a_1 then P can always produce b and terminate. The same holds if dataspaces are sets. However, when dataspaces are databases then P may block in the execution of its $rd(a_1)$ (regardless of the system in which P is executed) as a_2 may have already overwritten a_1.

Note that if dataspaces are databases and we execute the above program P in the ordered system then it will always block when reading a_1, while in the other two systems it may not block because a_2 may not yet be stored in the dataspace when a_1 is read. Thus, the centralized ordered system is observationally different from the other two systems when dataspaces are databases. The same result holds also if we substitute $del(a_1)$ for $rd(a_1)$ in P.

Next we give a program that makes the distributed system observationally different from the centralized unordered one when we interpret dataspaces as databases. Let $P = (S_1 \parallel S_2) \parallel S_3$ where

$$\begin{array}{l} S_1 = put(a_1).put(a_2).0 \\ S_2 = rd(a_1).rd(a_2).put(b).0 \\ S_3 = rd(b).rd(a_1).rd(a_2).put(c).0\,, \end{array} \tag{6}$$

and $a_1 \neq a_2$, $a_1 \bowtie a_2$, $a_1 \not\bowtie b$ and $a_2 \not\bowtie b$ (for example due to a communication protocol which guarantees that only data values from the same producer are received in the same order as they were produced). If P is executed in the centralized unordered system with a dataspace initially containing no b, a_1 and a_2, then it cannot produce the value c because b is sent to the dataspace only after a_2 has already overwritten a_1. Hence S_3 cannot read first b and then a_1 from the dataspace and block forever. However, S_3 may produce the value c if P is executed in the distributed system, with dataspaces initially containing no b, a_1 and a_2, because $a_2 \not\bowtie b$, and hence the agent of S_3 can receive first b and then a_2, having the possibility to read a_1 before a_2 overwrites it.

7 Testing for the absence of data

In Section 3.1 we modeled the *tfa* operation in the distributed system as a global blocking operation that succeeds only if the value requested is not present in the dataspaces of all agents. Another possibility is to model it as a local operation that requires no synchronization among the agents in the system. In this case the *tfa* operator becomes exactly the *dual* of the *rd* operation. Formally we need only to replace Rule $(D7)$ by the following rule:

$$(D7') \quad \frac{in(d,a) = ff}{[tfa(a).S,d,q] \xrightarrow{\tau} [S,d,q]}$$

Note that no transition is labeled by $\tilde{a}$ and hence the Rule $(D8)$ could safely be omitted. In this way the $tfa(a)$ operation executed by an agent A is local, and it may succeed even if another agent A' has already stored the value a in its local dataspace.

If we use the above local *tfa*, regardless of the structure of the dataspaces (multisets, sets or databases), the distributed system, the centralized unordered system, and the centralized ordered system are all observationally different.

To show the difference between the ordered system and the other two systems, consider the program P given in (2). If P is executed in the ordered system with a dataspace containing no a, then it cannot produce the value b, while the same program may terminate its execution in any of the other two systems with the same dataspace due to a possible delay in storing the value a in the dataspace.

Consider now the program $P = (S_1 \parallel S_2) \parallel S_3$, where

$$\begin{array}{l} S_1 = put(a).0 \\ S_2 = rd(a).put(b).0 \\ S_3 = rd(b).tfa(a).put(c).0\,, \end{array} \tag{7}$$

and $a \not\bowtie b$ (for example because the system uses a communication protocol which guarantees that data from the same producer is received by all processes in the same order as it was produced). If P is executed in a configuration of the distributed system with dataspaces not containing a or b, then when the agent associated to S_3 receives b it does not need to have already received a, hence, after executing the local *tfa* it may produce c. However if P is executed in the centralized unordered system with a shared dataspace containing no a and b, when the value b is present in the shared dataspace so is also a. Hence the statement S_3 blocks without producing the value c.

No overwriting or multiplicity of data values is involved in the above two programs, hence it does not matter if dataspaces are sets, multisets or databases.

8 Deleting data

The delete operation for the distributed system that we described in Section 3 is a blocking operation that requires synchronization among all agents. In this section we give an alternative definition of a synchronous global blocking delete operation for the distributed system and study its relationships with respect to synchronous and asynchronous global blocking delete operations for the centralized systems.

8.1 Another synchronous '*del*' for the distributed system

First we consider a synchronous delete for the distributed system that does not need to wait for the presence of the required value in all local dataspaces of the agents in the system. It is enough to replace Rule $(D6)$ in Section 3.1 by the following rule

$$(D6') \quad \frac{A \xrightarrow{\overline{a}} A'}{[S, d, q] \parallel A \xrightarrow{\overline{a}} [S, d \oplus \overline{a}, q] \parallel A'}$$

This rule, together with Rule $(D5)$ say that *del* is a blocking operation that synchronizes with all agents in the system leaving a message for the deletion of the requested value. The value need not to be present in the local dataspace of each agent, and it is deleted as soon as it is received.

The programs in (3) and (4) show that if we use Rule $(D6')$ in place of $(D6)$ and dataspaces are sets, then the distributed system, the centralized unordered system, and the centralized ordered system are all observationally different.

If dataspaces are multisets then program (2) can be used to show that the distributed system specified by substituting Rule $(D6')$ for Rule $(D6)$, is, in general, observationally different from the ordered system. However, the equivalence between the distributed system and the centralized unordered system is still valid.

Theorem 8.1 *Let $C \in Conf_D$ be a reachable configuration of the distributed system with a 'del' operator specified by replacing $(D6)$ with $(D6')$. The following holds:*

1. *for all $C' \in Conf_D$, if $C \longrightarrow\!\!\!\!\!\rightarrow_D C'$ then $\mathcal{E}_{DU}(C) \Longrightarrow_U C''$ with $\mathcal{E}_{DU}(C') = C''$;*
2. *for all $C' \in Conf_U$, if $\mathcal{E}_{DU}(C) \longrightarrow\!\!\!\!\!\rightarrow_U C'$ then $C \Longrightarrow_D C''$, with $\mathcal{E}_{DU}(C'') = C'$.*

By the above theorem and Theorem 4.2, it follows that that the distributed system with a '*del*' operator specified by replacing $(D6)$ with $(D6')$ and the centralized ordered system are observationally equivalent for programs not containing the *tfa* operator.

Furthermore, we can also use Theorem 4.1 in combination with Theorem 8.1 to conclude that the distributed system with a '*del*' operator specified by $(D6)$ is observationally equivalent to the distributed system with the alternative '*del*' specified by $(D6')$.

8.2 An asynchronous '*del*' for the unordered system

Next we consider a blocking *asynchronous* delete for the unordered system, which sends a request for deletion without synchronizing with the shared dataspace. We only need to replace Rule $(U5)$ in Section 3.2 by the following one:

$$(U5') \quad \frac{in(d, a) = tt}{\langle del(a).S, d, q\rangle \longrightarrow \langle S, d, \overline{a} \odot q\rangle}$$

The premise of Rule $(U5')$ says that *del* is blocking: it takes place only if the requested value is present in the shared dataspace. However, deletion of the value is not immediate, but it may have an unpredictable delay.

The unordered system with the asynchronous delete operator specified by substituting $(U5')$ for $(U5)$, is different from the ordered system, regardless of the structure of the shared dataspace (multiset, set or database). Consider the program

$$P \quad = \quad put(a).del(a).del(a).put(b).0\,. \qquad (8)$$

If P is executed in the ordered system with a dataspace containing no a then it cannot produce the value b because it blocks forever when executing the second $del(a)$. This may not be the case if P is executed in the unordered system with the same dataspace, because of the delay in receiving the message for the deletion of the value a.

We conclude by comparing the distributed system and unordered system with an asynchronous delete operator. We have two possibilities for the distributed system: either we consider the synchronous delete operator specified by Rule $(D6)$ or its alternative definition obtained by using Rule $(D6')$ instead of Rule $(D6)$. In both cases the two systems are observationally different regardless whether the dataspaces are multisets, sets or databases. Consider the above program P. When it is executed in the distributed system with dataspaces containing no a then it cannot produce the value b because it blocks at the second $del(a)$.

The two *del* operations we have considered for the distributed system have a synchronous behavior with respect to the dataspace of the agent executing it, and also with respect to the dataspaces of all other agents. By inserting the message for deletion $\overline{a}$ in the queue rather than in the local dataspace in Rule $(D6')$ we specify a global blocking delete operator with a synchronous behavior with respect to the dataspace of the agent executing it, but with an asynchronous behavior with respect to the dataspaces of all other agents. We can also specify a completely asynchronous global blocking delete operator by replacing Rule $(D5)$ with a rule similar to the above $(U5')$. Both these delete operators for the distributed system, and the comparison with the other two centralized systems are studied by the authors in [6].

9 Process creation

In this section we discuss another coordination primitive which adds new parallel processes to the system. The syntax of our language presented in Section 2 is changed by adding the coordination primitive

$$new(S)$$

to the prefixes α of the sequential statement in *Stat*, where S is a statement in *Stat*. We do not allow statements of the form $new(S_1).S_2 + S_3$ in which the *new* operator appears as prefix in one of the branches of a non-deterministic statement (we require this restriction only for simplicity, because then we do not need to change our rules for the + construct). Intuitively, the statement $new(S_1).S_2$ creates a new process S_1 to be executed in parallel with S_2. As for the *put* operation, its execution does not depend on the content of the dataspaces.

We model the new coordination primitive in the centralized ordered system by adding the following axiom to the specification in Section 3.3:

$$\boxed{(O8) \quad \langle new(S_1).S_2, d\rangle \longrightarrow \langle S_1 \parallel S_2, d\rangle}$$

When the new process is created it has immediately access to all data values that have been produced so far and have not yet been deleted.

Similarly, we model the behavior of the *new* operator in the centralized unordered system by adding the following axiom to the specification in Section 3.2:

$$\boxed{(U9) \quad \langle new(S_1).S_2, d, q\rangle \longrightarrow \langle S_1 \parallel S_2, d, q\rangle}$$

The difference with the previous system is that here the new process when created does not need to have immediately access to the all data values produced so far and that have not yet been deleted, due to the delay of the dataspace in receiving some of the data produced.

For example, let $P = put(a).new(S).del(a).0$ and $Q = put(a).del(a).new(S).0$ for $S = rd(a).put(b).0$. If we execute P and Q in one of the two centralized systems with a dataspace containing no a then P may produce value b, while Q cannot because a is not in the dataspace when S is created.

All results presented in this paper on the comparison between the two centralized systems hold also when we add the *new* operator as specified above.

We have several possibilities for the specification of the *new* operator for the distributed system. For example, the newly created process may be associated to an agent with an empty dataspace and with no pending messages, as specified by the axiom below:

$$\boxed{(D12) \quad [new(S_1).S_2, d, q] \xrightarrow{\tau} [S_1, \perp, \varepsilon] \parallel [S_2, d, q]}$$

The new process has no access to the data values already produced before its creation. The program

$$P \quad = \quad put(a).new(rd(a).put(b).0).0$$

shows that in this case the distributed system is different from both the two centralized systems, regardless whether dataspaces are multisets, sets, or databases. In the two centralized systems the value b will always be produced, but in the distributed system it will never be produced. The same program also shows that if the dataspace initially does not contain the value a, then the distinction remains even if we replace in the above Axiom $(D12)$ the agent $[S_1, \perp, \varepsilon]$ by either $[S_1, \perp, q]$ or $[S_1, d, \varepsilon]$. In both cases the value b may not be produced, depending on whether the value a is in the dataspace of the agent creator at the moment of creation.

The next axiom specifies a *new* operator for the distributed system in a such way that the newly created process has the same dataspace as that of its agent creator and the same pending messages:

$$\boxed{(D12') \quad [new(S_1).S_2, d, q] \xrightarrow{\tau} [S_1, d, q] \parallel [S_2, d, q]}$$

The above operator could be implemented, for example, by requiring that is the responsibility of an agent to send messages to all processes it creates.

All results presented in this paper on the comparison between the distributed system and the other two centralized systems hold also when we add the *new* operator as specified by the Axiom $(D12')$.

Note that initially the agent creating a process has to synchronize with the agent associated to the process created in order to transfer the content of its dataspace. Another possibility would be to send this content asynchronously. Below we show how to model this asynchronous transfer, without studying any further the consequences of this alternative definition of the *new* operator.

For $d \in DS$ and $q \in DQ$ we define the set of queues $d \propto q \subseteq DQ$ as follows:

$$\begin{aligned} &\perp \propto q = \{q\} \\ &d \propto q = \bigcup\{(d \oplus \overline{\sigma}) \propto (q \odot \sigma) \mid \sigma \in \Sigma \ \& \ in(d, \sigma) = tt\} \end{aligned}$$

If $q = \varepsilon$ then a queue in $d \propto q$ denotes a possible way to send all messages in the dataspace d. Otherwise it represents the sending of all messages in q after a possible way of sending all messages in the dataspace d.

The *new* operation for the distributed system with an asynchronous transfer of the content of the dataspace from the agent creator to the agent created, is specified by the following axiom:

$$\boxed{(D12'') \quad \frac{q_1 \in d \propto q_2}{[new(S_1).S_2, d, q_2] \xrightarrow{\tau} [S_1, \perp, q_1] \parallel [S_2, d, q_2]}}$$

Note that in this way the agent creating the new process may proceed immediately after the creation without waiting until the agent of the created process has received all values of its dataspace.

10 Conclusions

We have presented a framework for the comparison of distributed and centralized architectures for coordination languages. The main results are summarized in Table 1. Alternative definitions of some of the coordination primitives of the three architectures are given and their influence on the relationships are studied.

The analysis of these relationships is useful when one wants to have insight in the design choices for coordination languages. In particular, the designer usually wants to have an abstract model which is at the same time simple and tractable, and also implementable on different architectures with different degrees of distribution. In other words, one might want to reason considering a centralized architecture, even if the current implementation is inherently distributed. The framework we have introduced is useful in order to formally prove the equivalences between the abstract model and its implementations.

As future work, we plan to consider other systems, like a distributed system with a causal broadcast mechanism, and several other extensions of the language, like other operators for dynamical process creation. In [6] we have already investigated other interpretations of delete operators, including an asynchronous global blocking delete and a local blocking delete, like the one used in the coordination language SPLICE (formally described in [5]).

Due to space limitations, we have not included recursion in our language. All results presented in this paper hold also in presence of a recursion operator $recX.S$, where X is a program variable and S is a closed guarded statement, with the usual semantics [2].

We have not compared the relative expressive power of the three systems in this paper by, e.g., investigating how to implement coordination primitives in one system by composed statements in the other system. In [8] the expressiveness of two different models for the LINDA coordination language, corresponding to the ordered and the unorderd models considered here, has been compared. The language adopted in [8] (which extends the language considered here by comprising also recursion) is proven to be Turing powerful under the ordered system, while this is not the case for the unordered model because the termination problem is decidable.

Acknowledgments: We want to thank Paul Dechering and Edwin de Jong for clarifying discussions concerning different views of distributed systems, and Henk Goeman and all the members of the Amsterdam Coordination Group for their comments on a preliminary version of this paper.

References

[1] R. Amadio, I. Castellani, and D. Sangiorgi. On bisimulations for the asynchronous π-calculus. *Theoretical Computer Science*, 195(2):291-324, 1998.

[2] J. de Bakker and D. Scott. A theory of programs. Notes IBM seminar, Vienna, 1969. Appeared in *J.W. de Bakker, 25 jaar semantiek*, pages 1–30, CWI, Amsterdam, 1989.

[3] M. Boasson. Control systems software. *IEEE Transactions on Automatic Control* 38:7, pages 1094–1107, 1993.

[4] M. Bonsangue, F. Arbab, J. de Bakker, J. Rutten, A. Scutellá, and G. Zavattaro. A transition system semantics for the control-driven coordination language MANIFOLD. Technical Report, CWI, Amsterdam, 1998.

[5] M. Bonsangue, J. Kok, M. Boasson, and E. de Jong. A software architecture for distributed control systems and its transition system semantics. In *Proc. SAC'98*, pages 159–168. ACM press, 1998

[6] M.M. Bonsangue, J.N. Kok, and G. Zavattaro. Comparing software architecture for coordination languages. Unpublished note, 1998.

[7] N. Busi, R. Gorrieri, and G. Zavattaro. Three semantics of the output operation for generative communication. In *Proc. COORDINATION'97*, volume 1282 of LNCS, pages 205–219. Springer-Verlag, 1997.

[8] N. Busi, R. Gorrieri, and G. Zavattaro. On the Turing equivalence of Linda coordination primitives. In *Proc. of Express'97*, volume 6 of Electronic Notes in Theoretical Computer Science, 1997.

[9] N. Busi, R. Gorrieri, and G. Zavattaro. A process algebraic view of Linda coordination primitives. *Theoretical Computer Science*, 192(2):167–199, 1998.

[10] P. Bernstein, V. Hadzilacos, and N. Goodman. *Concurrency control and recovery in database systems.* Addison-Wesley, 1987.

[11] F.S. de Boer, J.N. Kok, C. Palamidessi, and J.J.M.M. Rutten. Non-monotonic concurrent constraint programming. In *Proc. Int. Logic Programming Symposium*, pages 315–334. The MIT press, 1993.

[12] A. Brogi and J.-M. Jacquet. Modeling coordination via asynchronous communication. In *Proc. COORDINATION'97*, volume 1282 of LNCS, pages 238–255. Springer-Verlag, 1997.

[13] N. Carriero and D. Gelernter. Linda in context. In *Communications of the ACM* 32:4, pages 444–458, 1989.

[14] P. Dechering, R. Groenboom, E. de Jong, and J.T. Udding. Formalization of a software architecture for embedded systems: a process algebra for SPLICE. In *Proc. FMERail* June, 1998.

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

[16] R. Milner. Lectures on a calculus for communicating systems. In *Seminar on Concurrency*, volume 197 of LNCS, pages 197–220, Springer-Verlag, 1985.

[17] V. Saraswat. *Concurrent constraint programming.* The MIT Press, 1993.

[18] G. Zavattaro. Towards an hierarchy for negative test operators for generative communication. In *Proc. of Express'98*, volume 16(2) of Electronic Notes in Theoretical Computer Science, 1998.

Biographical notes

Marcello Bonsangue is a post doctoral researcher in Computer Science at Leiden University. His research interests include operational, denotational, and axiomatic semantics of programming languages, object-oriented and distributed systems, modal logic, program refinement, mathematical structures for semantics, and topology.

Joost Kok is Professor of Computer Science at Leiden University. His research interests include intelligent data analysis, algorithms for optimization and prediction, neural networks, evolutionary computation, program development, and semantics of parallel and distributed systems.

Gianluigi Zavattaro is a Ph.D. student in Computer Science at Bologna University. His research interests include concurrency theory, semantics and expressiveness of coordination languages, and models for object-oriented distributed programming.