Comparing Coordination Models and Architectures using Embeddings

Marcello M. Bonsangue

CWI
P.O. Box 94079, 1090 GB Amsterdam, The Netherlands

Joost N. Kok

Leiden Institute of Advanced Computer Science, Leiden University
P.O. Box 9512, 2300 RA Leiden, The Netherlands

Gianluigi Zavattaro [1]

Bologna University, Department of Computer Science
Mura Anteo Zamboni 7, 40127 Bologna, Italy

Abstract

We refine the notion of embedding in order to obtain a formal tool for the comparison of the relative expressive power of different languages, by taking into account also the intended architectures on which the software components described using those languages are executed. The new notion, called architectural embedding, is suitable for the comparison of different communication mechanisms, and gives rise to a natural notion of implementability. We will use this notion to present equivalence and difference results for several coordination models based on components that communicate either through an unordered broadcast, through an atomic broadcast, or through a synchronous broadcast.

Key words: Coordination models languages and architectures, data-driven and control-driven coordination, structured operational semantics, architectural embeddings.

[1] Corresponding author. E-mail: `zavattar@cs.unibo.it`

1 Introduction

In this paper we introduce the notion of *coordination architecture* as a class of configurations described in terms of the coordination actions of the active processes, the repository of the data each process can observe, and the mechanisms used to communicate data among processes. Coordination architectures are not intended to support a general or complete architectural style: they focus only on the specific issue of coordination. For example, processes are the only type of components specified by a coordination architecture. Furthermore, a coordination architecture provides only a predefinite interaction abstraction rather than a set of generic descriptions of protocols governing the software composition.

The introduction of the abstraction of coordination architecture besides the typical notion of coordination language [17] has two main motivations:

- At the one hand, coordination architectures deal only with the mechanisms adopted for the interaction among components, and abstract away from the linguistic support adopted for the description of these components. For this reason coordination architectures can be considered a common framework for the comparison of different coordination languages even inspired by different onthologies; for example, in Section 5 we show that the same coordination architecture can be seen as the basis for both Splice [6], a coordination language based on the notion of shared data space, and Manifold [4], inspired on the contrary by the event notification metaphor.
- On the other hand, coordination languages are designed in order to be portable and adaptable to different system platforms [17]; e.g., the same coordination language should be implementable on either a centralized memory system or a distributed system. Using our notion of coordination architectures this concept can be rephrased by saying that it is desirable for a coordination language to be implementable on different coordination architectures. In this scenario it could be of interest to know whether it is possible to move components described in a coordination language from one coordination architecture to another one without altering the behavior of the whole system. In this paper we present a notion of equivalence between coordination architectures which can be exploited to provide an answer to this question.

Moreover, the coordination architecture abstraction allows us to introduce a new interesting definition of *coordination model* which separates the linguistic aspects from the behavioral concerns. Formally, we define a coordination model $\mathcal{S}$ as a function $\mathcal{S}{:}L \to A$ which maps a coordination language L used to describe the components of the intended systems, to a coordination architecture A which describes the way the specified components interact.

In this paper we consider three styles of coordination architectures which are general enough to describe several well known coordination models as described in Section 5. Each architecture consists of a number of interacting processes together with local stores used as data repositories. Interaction takes place by broadcasting messages to all other processes.

The three styles of architectures we consider differ in their broadcasting mechanism.

(1) The simplest broadcast mechanism to describe is the synchronous one: there is no observable delay between the broadcast and the receiving of data in the local store of each process. We call this type of architectures *undelayed*.
(2) In a second type of architectures, called *globally delayed*, the broadcast is atomic, meaning that there can be a delay between the broadcast of a data and its actual reception in the local store of a process, but the local store of all processes are guaranteed to receive the broadcast data value at the same time.
(3) In a third type of architectures, that we call *locally delayed*, the broadcast is unordered: the local store of each process may receive a broadcast data value at a different moment of time.

For each type of architecture we consider two possible structures for the data repositories: *multiset* and *set*. 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.

Furthermore we parameterize our architectures on the collection of coordination actions that can be executed by an active process. We consider language primitives for producing and consuming data values and for testing for the presence or absence of data. The production and consumption of one datum can be either local or global. In the first case only the data repository associated to the process is modified, whereas in the latter case a message containing the request for insertion or deletion of the intended data is broadcast according to the broadcast mechanism of the given architecture. We denote by lo and ld the operations for local production and consumption of data, and by go and gd the operations for global production and consumption of data. We consider only local testing operations, as it seems not reasonable in a distributed environment to force a global test on all the data repositories of all processes. We denote by ta the test for absence, and by tp the test for presence of a given data. All these coordination actions are blocking with the exception of the primitives for production of data lo and go. Thus a process can always produce a datum and continue immediately with the execution of other statements.

In order to compare coordination architectures we adapt to our new setting the comparison method introduced by Shapiro [23] under the name of *embedding*, thus obtaining a new notion we call *architectural embedding*. The idea is to study, given a configuration of one architecture, whether it is possible to embed the components in this configuration in a configuration of the other architecture without altering the overall behavior. We say that two architectures are equivalent when it is possible to define such an embedding from any of the configurations of the first architecture to configurations of the second one and vice versa; on the contrary, if this is not the case, we say that the two architectures are different. The analysis of the equivalence between coordination architectures gives interesting insights concerning the basic features characterizing the various architectures. Moreover, a notion of implementability for coordination models directly follows from our analysis: let $\mathcal{S}{:}L \to A$ be a coordination model defined on a coordination architecture A and let A' be a coordination architecture equivalent to A, then the coordination model $\mathcal{S}$ can be implemented in terms of the architecture A' simply by exploiting the embedding of A into A'.

We perform an exhaustive comparison of the considered architectures, proving for each pair of architectures whether they are equivalent or different. The following table summarizes the equivalence and difference results that we prove. The table is split in three parts; the first part considers data repositories as sets, the second one data repositories as multisets, and the third one compares the choice between sets and multisets. The comparison is made by taking into account different groups of coordination primitives described in the first column of the table.

	set			*multi set*			
	$L-G$	$L-U$	$G-U$	$L-G$	$L-U$	$G-U$	$set-multi\ set$
tp, lo, go	$=$	$=$	$=$	$=$	$=$	$=$	$=$
tp, lo, go, ta	$\neq$	$\neq$	$\neq$	$\neq$	$\neq$	$\neq$	$=$
tp, lo, go, ld	$\neq$	$\neq$	$\neq$	$=$	$=$	$=$	$\neq$
tp, lo, go, gd	$\neq$	$\neq$	$\neq$	$\neq$	$\neq$	$\neq$	$\neq$

Here L stands for the locally delayed architectural style, G for the globally delayed one, and U for the undelayed one.

The results can be interpreted as follows: *(i)* in the absence of consuming operators (either local or global) and tests for the absence of data the three types of architectures are all equivalent and the choice between a set or a multiset structure of the data repositories does not make a difference; *(ii)* the addition of tests for the absence of data permits to distinguish among the three architectures but not between the choice of data repositories as sets or multisets; *(iii)* the presence of local consuming operators permits to distinguish among the three communication mechanisms we consider, but only if the data repositories are sets; and *(iv)* the presence of global consuming operators permits to distinguish all types of communication mechanism we consider regardless of the structure of the adopted data repositories.

1.1 Related work

This paper is a revised and extended version of [9] and [10]. In both papers equivalence and separation results are studied for several architectures, but without an explicit notion of architectural embedding as tool for comparison.

The use of embedding as a method for language comparison has been proposed by Shapiro [23] and refined by De Boer and Palamidessi [7]. There are several variations of the notion of embedding depending upon a set of conditions on the coder and on the decoder functions. Some of these conditions have been tailored for analyzing the expressiveness of coordination languages. For example, in [11], a number of different coordination languages is compared, all relying on the same architecture, similar to our undelayed one. In [12] different implementations of an output operator have been studied in the setting of the coordination language Linda [15]. In [25] the expressiveness of several negative test operators has been investigated for a coordination language embodying an undelayed architecture. All these works analyze the expressive power of two languages form the point of view of their basic operators and constructors, while our emphasis is more on the architectural properties of the model underlying the languages.

The closest approaches to ours are taken in [1] and in [16], where languages which do not differ on their operators but only on their communication mechanisms are compared. 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 one used by the globally delayed architecture. Furthermore we prove that the equivalence holds because no global consuming operators are considered and because all data repositories have a multiset structure rather than a set structure.

In [16] fifty communication models for Message Sequence Charts are analyzed, and a hierarchy is defined according to a notion of implementability, defined by means of set of sequences of production, transmission and reception of messages. There are no operators for testing the presence or absence of data. Furthermore, the structure of the data repositories in all communication models is either a FIFO buffer or a multiset. In the latter case, our undelayed and globally delayed architectures can be mapped in two of their models, and our equivalence result between the two architectures coincides with their equivalence between the two respective communication models. All other models are incomparable to the architectures we considered.

1.2 Structure of the paper

The rest of the paper consists of five sections. Section 2 formally introduces our method of comparison. Section 3 deals with the modeling of the coordination architectures . Then in Section 4 we compare the different coordination architectures. Three coordination models based on existing languages (Linda, Splice and Manifold) are introduced in Section 5. Finally, in Section 6 we give conclusions and discuss future work.

2 The method of comparison: embedding

In this section we first describe how languages can be compared and then we propose an extension for the comparison of architectures.

A natural way to compare the expressive power of two languages is to study whether it is possible to translate all statements of one language into statements of the other language with the same observable behavior. In general, however, this method of comparison is too restrictive because it requires that the semantic domains of the two languages are same. This restriction can be relaxed by introducing an abstraction from the semantic domain of the second language to the domain of the first language. This relative comparison method has been introduced by Shapiro [23] under the name of *embedding*. Assume given two languages L_0 and L_1 together with their semantic functions $\mathcal{M}_0{:}L_0 \to O_0$ and $\mathcal{M}_1{:}L_1 \to O_1$. Then L_1 embeds L_0 if there exists a coder map $\mathcal{C}{:}L_0 \to L_1$ and a decoder map $\mathcal{D}{:}O_1 \to O_0$ such that the following diagram commutes:

$$\begin{tikzcd}
L_0 \arrow[r, "\mathcal{M}_0"] \arrow[d, "\mathcal{C}"'] & O_0 \\
L_1 \arrow[r, "\mathcal{M}_1"'] & O_1 \arrow[u, "\mathcal{D}"']
\end{tikzcd}$$

The notion of embedding is too weak if no restrictions are imposed on $\mathcal{C}$ and $\mathcal{D}$. In fact, if L_1 is a Turing complete language then in general it embeds any other language L_0. There is no general agreement on what restrictions should be required on the coder and decoder maps, since these may depend on the goal of the comparison between the two languages [7,21,25,11]. Commonly, the following properties are required:

(1) the coder should be compositional with respect to some of the operators of the language (e.g., the parallel composition operator),
(2) the decoder should preserve some predefined semantics (e.g., the behavior with respect to termination).

In general one proves the embedding of a language into another one by giving a translation of all the language operators. In this paper we refine the notion of language embedding by introducing the concept of coordination architecture into the notion of embedding in order to compare different ways a system can be composed.

As described in the introduction, we are interested in refining the notion of embedding to the more specific notion of architectural embedding. Before discussing this, we need to introduce the formal definition of coordination architecture.

We describe a coordination architecture A by the collection of all its configurations. Each configuration C consists of a multiset of active processes $Proc(C)$ and of structural elements needed for their communication, like data repositories and/or communication channels. We identify by $Init(A)$ the set of all initial configurations of the architecture A. The behavior of an architecture A is defined in terms of a semantic map $\mathcal{T}{:}A \to O$, where O is some suitable semantic domain.

A *coordination language* is the linguistic support for describing the run-time interactions specified in terms of a specific coordination architecture. A *coordination model* is a function $S : L \to A$ from a (coordination) language L to an initial configuration of a coordination architecture A. A coordination language focuses on the description of the interactions between the concurrent and distributed processes that have to cooperate or synchronize to achieve a common goal [3].

Let $\mathcal{S}{:}L \to A$ be a coordination model. In order to define the semantics of the coordination language L it is enough to have a semantic map $\mathcal{T}{:}A \to O$, where O is some suitable semantic domain. This approach to the semantics of a coordination language is modular since once we have fixed the semantics of a coordination architecture A, we then have a semantics for all coordination languages which embody A in their coordination model. Furthermore it allows for an extension of the language embedding as comparison method by taking into account the architectures underlying the languages.

Consider two coordination models $\mathcal{S}_0{:}L_0 \to A_0$ and $\mathcal{S}_1{:}L_1 \to A_1$ and assume given the semantic functions $\mathcal{T}_0{:}A_0 \to O_0$ and $\mathcal{T}_1{:}A_1 \to O_1$ for some suitable domains of observables O_0 and O_1. Then we say that the coordination model $\mathcal{S}_1{:}L_1 \to A_1$ *embeds* $\mathcal{S}_0{:}L_0 \to A_0$ if there exists a language coder $\mathcal{C}{:}L_0 \to L_1$, an architectural encoder $\mathcal{E}{:}A_0 \to A_1$ and a decoder $\mathcal{D}{:}O_1 \to O_0$ such that the following diagram commutes (i.e., all the paths from L_0 to A_1 are equivalent as also all paths from L_0 to O_0):

$$\begin{tikzcd}
L_0 \arrow[r, "\mathcal{S}_0"] \arrow[d, "\mathcal{C}"'] & A_0 \arrow[r, "\mathcal{T}_0"] \arrow[d, "\mathcal{E}"] & O_0 \\
L_1 \arrow[r, "\mathcal{S}_1"'] & A_1 \arrow[r, "\mathcal{T}_1"'] & O_1 \arrow[u, "\mathcal{D}"']
\end{tikzcd}$$

In order to use the above notion as a tool for the comparison of coordination models and architectures we add the following restrictions on the encoder and decoder functions:

(P1) The architectural encoder $\mathcal{E}{:}A_0 \to A_1$ should preserve the active processes and their distribution. More formally, we require that $Proc_0(C) = Proc_1(\mathcal{E}(C))$ for any configuration C of the architecture A_0, where $Proc_0$ and $Proc_1$ are two functions returning multisets of active processes from each configuration of the architectures A_0 and A_1, respectively. This requirement is justified by the fact that we are interested in comparing architectures only in terms of their system structure, like communication mechanisms and data repositories. In other words, we want to use architectural encodings to move the active components from the source architecture to the target one changing only the interaction mechanisms and not the component.

(P2) The architectural encoder $\mathcal{E}{:}A_0 \to A_1$ should map initial configuration of one architecture into initial configuration of the other. More formally, we require that $\mathcal{E}(C) \in Init_1(A_1)$ for all $C \in Init_0(A_0)$.

(P3) The decoder $\mathcal{D}{:}O_1 \to O_0$ should preserve the behavior of the original system with respect to a reasonable semantics, that is a semantics that distinguishes two systems whenever in a computation of one system it is possible to observe the production of one of some intended data values that cannot be observed in any computation of the other system. Formally, if $\{\surd_1, \ldots, \surd_n\}$ is the set of our intended values, we require that given $o \in O_1$, then, for any $i \in 1 \ldots n$, $\surd_i \in ov_1(o)$ if and only if $\surd_i \in ov_0(\mathcal{D}(o))$, where ov_0 and ov_1 are two functions extracting the observable values of each computation in O_0 and O_1, respectively.

Note that no requirements are imposed on the coder map $\mathcal{C}{:}L_0 \to L_1$, as these may depend on the purpose of the language comparison. We say that an embedding between two coordination models is *architectural* if the encoder and decoder functions satisfy the properties listed above.

Given a coordination model $\mathcal{S}_0{:}L_0 \to A_0$ (with the given semantic function $\mathcal{T}_0{:}A_0 \to O_0$) and an architecture A_1 (with its given semantic function $\mathcal{T}_1{:}A_1 \to O_1$), we say that $\mathcal{S}_1{:}L_0 \to A_1$ is an *implementation* of the coordination model $\mathcal{S}_0$ on the architecture A_1 if there exists an architectural embedding of $\mathcal{S}_0$ into $\mathcal{S}_1$ with the identity as coder function. It is interesting to observe that $\mathcal{S}_0$ has an implementation on the architecture A_1 if and only if there exists a *universal embedding* of the architecture A_0 into A_1, defined as an encoder $\mathcal{E}{:}A_0 \to A_1$ and a decoder $\mathcal{D}{:}O_1 \to O_0$ satisfying the above three properties and such that $\mathcal{D} \circ \mathcal{T}_1 \circ \mathcal{E} = \mathcal{T}_0$. Indeed, if there exists such an universal embedding between A_0 and A_1, then the implementation $\mathcal{S}_1{:}L_0 \to A_1$ is simply defined as $\mathcal{S}_1 = \mathcal{E} \circ \mathcal{S}_0$, that is, the functional composition of the encoder function $\mathcal{E}$ of the universal embedding and the given function $\mathcal{S}_0$. On the other hand, given an implementation of $\mathcal{S}_1$ on the architecture A_1, the universal embedding of A_0 into A_1 is given by the encoder and decoder functions of the corresponding architectural embedding of $\mathcal{S}_0$ into $\mathcal{S}_1$.

2.1 The operational semantics

The behavior of the configurations is described by means of rooted transition systems $(C, r, \longrightarrow)$. The nodes C are the configurations of the considered architecture, the root node r is an element of the initial configurations, and the transition $\longrightarrow$ specifies how the architecture evolves: a transition $c \xrightarrow[\ell]{o} c'$ states that the configuration c of an architecture may evolve to a configuration c' by producing an observable o and an effect ℓ. The observables model the data that the system makes available to the environment, whereas the effects specify the information needed for modeling the process interaction. We assume that only the broadcast of messages is observable, and write τ on top of a transition when it produces no observables. Similarly, we write τ at the bottom of a transition when it produces no effect, for example because the transition specifies an evolution of the architecture that does not involve any interaction. For simplicity, we will abbreviate $c \xrightarrow[\ell]{\tau} c'$ as $c \xrightarrow[\ell]{} c'$, $c \xrightarrow[\tau]{o} c'$ as $c \xrightarrow{o} c'$, and $c \xrightarrow[\tau]{\tau} c'$ as $c \longrightarrow c'$. In the following we use $c \Longrightarrow c'$ to denote a possibly empty path of unobservable transitions from the node c to the node c'; formally, $c \Longrightarrow c'$ iff either $c = c'$ or there exist $c_1, \ldots, c_n$ and $\ell_1, \ldots, \ell_{n+1}$ (possibly equal to τ) such that $c \xrightarrow[\ell_1]{} c_1 \xrightarrow[\ell_2]{} \cdots \xrightarrow[\ell_n]{} c_n \xrightarrow[\ell_{n+1}]{} c'$.

Having in mind that the only observable transitions are those with a label different from τ on top, we say that two rooted transition systems are equivalent if every possible observable transition in the one system corresponds with an equivalent transition in the other (as for usual bisimulation equivalence), apart from some arbitrary long sequences of unobservable transitions that are allowed to precede or follow, and furthermore every unobservable transition corresponds to an arbitrary long (possibly empty) of unobservable transitions. Thus we use as semantic domains the collection of rooted transition systems modulo weak bisimulation [19,20]:

Definition 1 *Let $(C_1, r_1, \longrightarrow_1)$ and $(C_2, r_2, \longrightarrow_2)$ be two rooted transition systems. We say that a relation $\mathcal{R} \subseteq C_1 \times C_2$ is a* weak bisimulation *if for each $(c_1, c_2) \in \mathcal{R}$ we have that*

(1) if $c_1 \xrightarrow[\ell_1]{o}_1 c_1'$ then either $o = \tau$ and $(c_1', c_2) \in \mathcal{R}$, or there exists an effect ℓ_2 and a path $c_2 \Longrightarrow_2 c \xrightarrow[\ell_2]{o}_2 c' \Longrightarrow_2 c_2'$ such that $(c_1', c_2') \in \mathcal{R}$ and
(2) if $c_2 \xrightarrow[\ell_2]{o}_2 c_2'$ then either $o = \tau$ and $(c_1, c_2') \in \mathcal{R}$, or there exists an effect ℓ_1 and a path $c_1 \Longrightarrow_1 c \xrightarrow[\ell_1]{o}_1 c' \Longrightarrow_1 c_1'$ such that $(c_1', c_2') \in \mathcal{R}$.

We say that the two rooted transition systems are weakly bisimilar *if there exists a weak bisimulation $\mathcal{R} \subseteq C_1 \times C_2$ such that $(r_1, r_2) \in \mathcal{R}$.*

We denote by $TS/\approx$ the class of all rooted transition systems modulo weak bisimulation. According to this semantic domain, the existence of an universal embedding of an architecture A_0 into another architecture A_1 intuitively means that an observer is not capable to distinguish whenever a program is executed according the communication mechanism of A_0 or A_1. Conversely, the non-existence of such an universal embedding means that there is a program that if it executes according the communication mechanism of A_0 it produces a datum that cannot be observed when the same program is executed using the communication mechanism of the other architecture. Of course, it may be the case that this program cannot be written in a specific coordination model $\mathcal{S}{:}L \to A_0$, and thus it is still possible that $\mathcal{S}$ can be implemented by the architecture A_1.

3 Modeling Coordination Architectures

The three styles of coordination architectures we consider are uniformly described by a collection of configurations, describing the architectural components, and by an operational semantics, defining the behavior of the components. In each architecture there are a number of active processes which interact only by broadcasting data. Each process is associated with a local memory used as a data repository. We first present a syntax for the description of the process components of the architecture, and introduce some basic building blocks needed for the specification of its behavior.

3.1 Processes and data repositories

Let *Data*, ranged over by $a, b, \ldots$, be a set of *data values* that we assume will be used by the active processes for their interactions. We consider two basic types of interactions for each datum a: the request for its insertion in a data repository (denoted by the message a), and the request for its deletion from a data repository (denoted by the massage $\bar{a}$). Messages are thus elements of the set

$$Msg = \{a, \bar{a} \mid a \in Data\}\,,$$

ranged over by $m, m', \ldots$. We use the convention that $\bar{\bar{m}} = m$ for $m \in Msg$. A data repository is a structured collection of elements taken from this set. In this paper we consider two simple structures: multiset and set. We denote by $DS_\mu = Msg \to \mathbb{N}$ the set of all data repositories with a multiset structure, and by $DS_\sigma = Msg \to \{0, 1\}$ the set of all data repositories with a set structure. In the following we use d to range over both the sets DS_μ and DS_σ; given a set d with a slight abuse of notation we sometimes use d to denote the corresponding multiset containing one occurrence for each datum contained in d.

We write $\mathbf{0}$ for the empty (multi) set, that is, $\mathbf{0}(m) = 0$ for every message $m \in Msg$. We define the predicate *in* testing if a message m is in the data repository d by

$$in(d, m) \equiv d(x) > 0\,,$$

and the function $\oplus$ for inserting the message m into the data repository d by

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

where, for any function $f\colon X \to Y$ 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 data repositories with a set or multiset structure (the condition on the codomain of d makes the distinction here), and for both adding and removing values from a data repository. Informally, a data value a is inserted into a data repository only if no request for deletion $\bar{a}$ is present. Otherwise a is not inserted and $\bar{a}$ is removed from the data repository. Conversely, a data value a from a data repository is removed when the message $\bar{a}$ arrives. In case the message $\bar{a}$ arrives and the value a is not present in the data repository then $\bar{a}$ is stored into the data repository.

When produced, messages are associated to a sort used by the communication protocol to guarantee a common order in their reception among all processes. For example an architecture may use a protocol that guarantees two messages to be received by any process in the same order they were produced only if they are both produced by the same component. We assume the existence of an abstract set *Sorts* of *data sorts*, ranged over by $s, t, \ldots$, and define the set Σ of *broadcast-able messages* as follows:

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

Intuitively, two messages with same sort will be received by any agent in the same order as they were produced. Formally this is achieved by using *queues of pending messages*. A queue q is a partially commutative string defined as a congruence class of finite strings in the monoid $(\Sigma^\star, \cdot, \varepsilon)$ modulo the least congruence such that, for all $m{:}s, m'{:}s' \in \Sigma$,

$$m{:}s \cdot m'{:}s' = m'{:}s' \cdot m{:}s \text{ if } s \neq s'\,.$$

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 $m{:}s$ is the congruence class containing the one-element string $m{:}s \in \Sigma$ and ε the congruence class containing the empty string.

The way sorts are associated to data is an architectural issue and therefore should be transparent at the level of the coordination model. They are formalisms to abstract from the implementation of a specific type of broadcast. For example one architecture may use a broadcast algorithm that guarantees that all messages broadcast by the same source are received in the same order they were produced. This type of broadcast can be specified in our formalism by assigning the same sort to all data broadcast by a process, and different sorts to data broadcast by different processes. Another architecture may guarantee that all processes receive data in the same order they were produced. This type of broadcast corresponds to associating to each data the same sort, regardless of the process producing it.

The behavior of each process in isolation is syntactically described by a synchronization tree labeled by a collection of coordination actions. We have adopted this representation of processes as we would like to abstract away from the syntax of the different languages by observing the behavior of programs only. The coordination actions that we consider are either internal, local or global. Local actions only consider the data repository of their own process, and do not produce any message. Global actions produce messages that are broadcast in order to act on remote data repositories too. Because of the broadcast, global actions will require the data value to be broadcasted and its sort to be used during the communication protocol. We start our analysis by taking into account only three basic coordination primitives: the local and global output operations *lo* and *go*, and the local test for presence *tp*. The other primitives will be introduced later.

Formally, a process is a term of the following grammar:

$$
\begin{aligned}
P &::= 0 \mid \sum_I \alpha_i.P_i \\
\alpha &::= \tau \mid tp(a) \mid lo(a) \mid go(a{:}s)\,,
\end{aligned}
$$

where $a \in \mathit{Data}$, $s \in \mathit{Sorts}$, and I is a non-empty (possibly infinite) index set. The term 0 denotes the inactive process, and it is usually omitted for the sake of simplicity. Further we consider the usual action prefixing, and choice operators. We denote the collection of all process by the set $\mathit{Process}$, ranged over by P, Q. We do not treat recursion in this paper, but it seems rather straightforward to add it a later stage.

Informally, the meaning of the prefixes is as follows: τ denotes some internal activity, $tp(a)$ tests for the presence of an occurrence of the value a in the local data repository without consuming it, $lo(a)$ introduces a new instance of value a in the local data space, and $go(a{:}s)$ emits a new instance of the value a that is broadcast to all the components as the message $a{:}s$. The tp action is blocking, meaning that it is executed only if the required data is present.

The actions lo and go do not depend on the actual content of the local data repository and can always be executed. In Section 4 we will extend the set of prefixes with coordination actions for locally testing for the absence of a value, for consuming a local occurrence of value, and for broadcasting a request of the deletion of a value.

The sort associated to the data value in the go primitive is used to model specific types of broadcast. We can abstract from this architectural issue by means of the function prc on processes, defined inductively as follows:

$$
\begin{aligned}
prc(0) &= 0 \\
prc(\textstyle\sum_I \alpha_i.P_i) &= \textstyle\sum_I prc(\alpha_i.P_i) \\
prc(\tau.P) &= \tau.prc(P) \\
prc(tp(a).P) &= tp(a).prc(P) \\
prc(lo(a).P) &= lo(a).prc(P) \\
prc(go(a{:}s).P) &= go(a).prc(P)
\end{aligned}
$$

This function will be used when defining the multiset of active processes of a configuration of an architecture.

In the next three subsections we introduce three styles of coordination architectures. For each of the three styles, we consider two instantiations, one in which data spaces are sets, and one in which they are multisets. In this way we obtain six different coordination models. In order to abstract away from the choice between sets or multisets, we use a general index γ which ranges over the set $\{\sigma, \mu\}$ where σ represents sets and μ represents multisets.

We present, for each architecture A, the collection of all its configurations $Conf(A)$, a set $Init(A)$ of initial configurations, a function mapping each configuration $C \in Conf(A)$ to the multiset of its active processes $Proc_A(C)$, and a transition system specification which defines a general labelled transition system with states taken from $Conf(A)$. The behavior of an initial configuration $C \in Init(A)$ is given by the rooted transition system $TS_A(C)$ obtained by selecting the part of the general system reachable from the root C. Thus, the semantics of the architecture A is defined as the function $\mathcal{T}_A$ mapping a configuration $C \in Init(A)$ to the equivalence class (with respect to weak bisimulation) containing the rooted transition system $TS_A(C)$.

In the description of property (P3) we have adopted a function ov which extracts the observable values produced during a computation. Formally, let C be the considered configuration and let $TS_A(C)$ be the rooted transition system describing its behavior; the corresponding observable values $ov(TS_A(C))$

are simply the observable labels present in the transition system $TS_A(C)$.

As the bisimulation relation is defined in terms of the observable labels only, and provided that they are the same for each of the defined transition systems, we have that the domain of the transition systems up to weak bisimulation is the same for each architecture. For this reason, we will omit the index A writing $TS(C)$ instead of $TS_A(C)$.

3.2 The locally delayed architectures L_γ

In the *locally delayed architectures* communication between processes is established by broadcasting messages using a protocol that does not guarantee that all processes receive data values at the same time they were produced.

The set of configurations $\mathit{Conf}(L_\gamma)$ of a locally delayed architecture is defined by the grammar

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

where $P \in \mathit{Process}$ is a process, and $d \in DS_\gamma$ is its associated data repository. Each process P in a configuration C is associated to a queue $q \in DQ$ containing the messages already produced by some process but not yet received by P. The operator $\parallel$ denotes the parallel composition of the processes that compose the actual configuration of the architecture. Its intended meaning is to be a a commutative and associative operator. Formally this is achieved by means of a structural congruence $\equiv_{L_\gamma}$ defined as the least congruence on $\mathit{Conf}(L_\gamma)$ such that

$$C_1 \parallel C_2 \equiv_{L_\gamma} C_2 \parallel C_1 \quad \text{and} \quad C_1 \parallel (C_2 \parallel C_3) \equiv_{L_\gamma} (C_1 \parallel C_2) \parallel C_3\,.$$

In the following we will reason up to the structural congruences defined for each architecture; in other words, we do not make any distinction between C and C' if they are structural congruent.

A configuration is initial if no value is present in all data repositories and there are no pending messages. Thus $\mathit{Init}(L_\gamma)$ is the subset of $\mathit{Conf}(L_\gamma)$ defined by the grammar

$$C ::= [P, \mathbf{0}, \varepsilon] \mid C \parallel C\,,$$

where $P \in \mathit{Process}$.

The multiset of active process $Proc_{L_\gamma}(C)$ of a configuration $C \in Conf(L_\gamma)$ is defined as expected:

$$
\begin{aligned}
Proc_{L_\gamma}([P,d,q]) &= \{\!|prc(P)|\!\} \\
Proc_{L_\gamma}(C_1 \parallel C_2) &= Proc_{L_\gamma}(C_1) \uplus Proc_{L_\gamma}(C_2)
\end{aligned}
$$

Here $\{\!|prc(P)|\!\}$ denotes the singleton multiset containing the term obtained by abstracting from P the architectural information it encodes, and $\uplus$ denotes the usual multiset union.

The transition system specification of the locally delayed architecture is given by the following axioms and rules:

(L1) $[\tau.P, d, q] \longrightarrow [P, d, q]$

(L2) $[tp(a).P, d, q] \longrightarrow [P, d, q]$ if $in(d, a) = tt$

(L3) $[lo(a).P, d, q] \longrightarrow [P, d \oplus a, q]$

(L4) $[go(a{:}s).P, d, q] \xrightarrow[a:s]{a} [P, d, a{:}s \odot q]$

(L5) $[P, d, q \odot m{:}s] \longrightarrow [P, d \oplus m, q]$

(L6) $\dfrac{[\alpha_k.P_k, d, q] \xrightarrow[\ell]{o} [P_k, d', q']}{[\sum_I \alpha_i.P_i, d, q] \xrightarrow[\ell]{o} [P_k, d', q']}$ if $k \in I$

(L7) $\dfrac{C \xrightarrow{o} C'}{[P, d, q] \parallel C \xrightarrow{o} [P, d, q] \parallel C'}$

(L8) $\dfrac{C \xrightarrow[m:s]{o} C'}{[P, d, q] \parallel C \xrightarrow[m:s]{o} [P, d, m{:}s \odot q] \parallel C'}$

(L9) $\dfrac{C \equiv_{L_\gamma} D \quad C \xrightarrow[\ell]{o} C' \quad C' \equiv_{L_\gamma} D'}{D \xrightarrow[\ell]{o} D'}$

The first four axioms describe the behavior of the primitive actions. The side condition in axiom (L2) reflects the fact that the *tp* action may block. Axiom (L5) describes the receiving of a messages by a single process. Finally, the other rules are the usual for compound processes, with the exception of the rules (L8) that specifies the interaction among processes: when a message is broadcast its effect is global to all processes.

At each transition, effect and observations are similar. The only difference is that the observation of a broadcast is the message produced, while its effect can also depend on the sort of the message. This because different processes may receive messages with different sorts in a different order. Notice that the observer has no knowledge about the process executing the broadcast.

3.3 The globally delayed architectures G_γ

In the *globally delayed architectures* processes communicate through an atomic broadcast that guarantees that all processes receive data values at the same time.

The set of configurations of a globally delayed architecture is defined by the set

$$Conf(G_\gamma) = \{(A, q) \mid A \in Agents_\gamma, q \in DQ\},$$

where $Agents_\gamma$ is a set defined by the grammar

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

Here $P \in Process$ is a process, and $d \in DS_\gamma$ is its associated data repository. The difference with the configuration of the previous architectures is that here all processes share the same queue of pending messages. As before, the operator $\parallel$ is used to compose processes in parallel. It is a commutative and associative operator as specified by the structural congruence $\equiv_{G_\gamma}$, that is the least congruence on $Agents_\gamma$ such that

$$A_1 \parallel A_2 \equiv_{G_\gamma} A_2 \parallel A_1 \quad \text{and} \quad A_1 \parallel (A_2 \parallel A_3) \equiv_{G_\gamma} (A_1 \parallel A_2) \parallel A_3 .$$

A configuration (A, q) is initial if $q = \varepsilon$ and A is an agent defined by the grammar

$$A ::= [P, \mathbf{0}] \mid A \parallel A ,$$

where $P \in Process$. Thus no value is present in all data repositories and there are no pending messages.

The multiset of active processes $Proc_{G_\gamma}(C)$ of a configuration $C \in Conf(G_\gamma)$

is defined as for the locally delayed architectures:

$$\begin{aligned} Proc_{G_\gamma}([P,d],q) &= \{\!|prc(P)|\!\} \\ Proc_{G_\gamma}(A_1 \parallel A_2, q) &= Proc_{G_\gamma}(A_1,q) \uplus Proc_{G_\gamma}(A_2,q). \end{aligned}$$

Finally, the transition system specification of the globally delayed architecture is given by the following axioms and rules:

(G1) $[\tau.P,d],q \longrightarrow [P,d],q$

(G2) $[tp(a).P,d],q \longrightarrow [P,d],q \quad \text{if } in(d,a)=tt$

(G3) $[lo(a).P,d],q \longrightarrow [P,d\oplus a],q$

(G4) $[go(a{:}s).P,d],q \xrightarrow{a} [P,d],a{:}s \odot q$

(G5) $[P,d],q \odot m{:}s \xrightarrow[m]{} [P,d\oplus m],q$

(G6) $\dfrac{[\alpha_k.P_k,d],q \xrightarrow[\ell]{o} [P_k,d'],q'}{[\sum_I \alpha_i.P_i,d],q \xrightarrow[\ell]{o} [P_k,d'],q'} \quad \text{if } k \in I$

(G7) $\dfrac{A,q \xrightarrow{o} A',q'}{[P,d] \parallel A,q \xrightarrow{o} [P,d] \parallel A',q'}$

(G8) $\dfrac{A,q \xrightarrow[m]{o} A',q'}{[P,d] \parallel A,q \xrightarrow[m]{o} [P,d\oplus m] \parallel A',q'}$

(G9) $\dfrac{A \equiv_{G_\gamma} B \quad A,q \xrightarrow[\ell]{o} A',q' \quad A' \equiv_{G_\gamma} B'}{B,q \xrightarrow[\ell]{o} B',q'}$

The axiom (G4) shows that when a data item is broadcast then it is not immediately visible to all agents. The fact that they eventually will receive a message at the same time is modeled by (G5) together with (G8). As in the previous architecture, the *tp* operator is blocking (expressed by the side condition of axiom (G2)), the *lo* operation is local (axiom (G3)) and the *go* operation has a global effect obtained through the broadcasting of a message.

Notice that the broadcast of a message has effect on the other processes only when the message is actually delivered. For uniformity with the other two architectures, we assume that the observations take place when the message is produced.

3.4 The undelayed architectures U_γ

Finally we consider the *undelayed architectures*. Communication happens via a synchronization among all active processes that guarantees they all receive data values at the same time at which they were produced.

The set $Conf(U_\gamma)$ of configurations of an undelayed architecture is defined by the grammar

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

Here $P \in Process$ is a process, and $d \in DS_\gamma$ is its associated data repository. The difference with the configurations of the previous two architectures is that here there are no queues of pending messages. As before, the operator $\parallel$ is used to compose processes in parallel. It is a commutative and associative operator specified by the structural congruence $\equiv_{U_\gamma}$, that is the least congruence on *Agents* such that

$$C_1 \parallel C_2 \equiv_{U_\gamma} C_2 \parallel C_1 \quad \text{and} \quad C_1 \parallel (C_2 \parallel C_3) \equiv_{U_\gamma} (C_1 \parallel C_2) \parallel C_3\,.$$

A configuration is initial if it is generated by the grammar

$$C ::= [P, \mathbf{0}] \mid C \parallel C\,,$$

where $P \in Process$. Thus no value is present in all data repositories.

The multiset of active processes $Proc_{U_\gamma}(C)$ of a configuration $C \in Conf(U_\gamma)$ is defined as for the other two architectures:

$$\begin{aligned} Proc_{U_\gamma}([P, d]) \quad &= \{\!|\, prc(P) \,|\!\} \\ Proc_{U_\gamma}(C_1 \parallel C_2) &= Proc_{U_\gamma}(C_1) \uplus Proc_{U_\gamma}(C_2). \end{aligned}$$

Finally, the transition system specification of the undelayed architecture is given by the following axioms and rules:

(U1) $[\tau.P, d] \longrightarrow [P, d]$

(U2) $[tp(a).P, d] \longrightarrow [P, d]$ if $in(d, a) = tt$

(U3) $[lo(a).P, d] \longrightarrow [P, d \oplus a]$

(U4) $[go(a{:}s).P, d] \xrightarrow[a]{a} [P, d \oplus a]$

(U5) $\dfrac{[\alpha_k.P_k, d] \xrightarrow[\ell]{o} [P_k, d']}{[\sum_I \alpha_i.P_i, d] \xrightarrow[\ell]{o} [P_k, d']}$ if $k \in I$

(U6) $\dfrac{C \xrightarrow{o} C'}{[P, d] \parallel C \xrightarrow{o} [P, d] \parallel C'}$

(U7) $\dfrac{C \xrightarrow[m]{o} C'}{[P, d] \parallel C \xrightarrow[m]{o} [P, d \oplus m] \parallel C'}$

(U8) $\dfrac{C \equiv_{U_\gamma} D \quad C \xrightarrow[\ell]{o} C' \quad C' \equiv_{U_\gamma} D'}{D \xrightarrow[\ell]{o} D'}$

The synchronous behavior of the *go* operation is modeled by the axiom (U4) together with the rule (U7). All other operations are local (the remaining axioms together with $(U6)$). As in the previous architectures, the *tp* operation is blocking, while the *lo* and the *go* operations are not.

In the undelayed architectures there is no difference between observables and effects. This is because in the broadcast of a message, the production coincides with the delivering of the messages, and the sort associated to the message does not play any role, and can safely be omitted.

4 Comparing Coordination Architectures

In this section we compare the coordination architectures that were introduced in the previous section. Given a pair of architectures, we investigate the possibility to define a universal embedding of the first architecture in the second one, and vice versa. If this is possible, we say that the two architectures are *equivalent*, otherwise we say that they are *different*. This kind of analysis permits to investigate the specific features of the coordination architectures independently of the coordination models which embed them. Moreover, given two equivalent coordination architectures, we can state that each coordination model defined using one of them, can be implemented also in the other one.

An interesting general observation is related to the possibility to compose the universal embeddings. As an example, consider the existence of three architectures A_0, A_1 and A_2. Suppose now the existence of two universal embeddings, the first from A_0 to A_1 and the second from A_1 to A_2. It is easy to see that the functional composition of the encoding and decoding functions define a universal embedding from A_0 to A_2. Given this observation, we can conclude that our relation of equivalence among architectures is transitive, thus it is an equivalence relation (it is trivially also reflexive and commutative).

The first result that we present in this section is that, if we consider only the basic coordination primitives, the six architectures are all equivalent. After, we independently introduce the other three operators (test for absence, local, and global delete) and we investigate if the equivalence results continue to hold or not.

4.1 Comparison with the basic operators

We start by considering the six coordination architectures as introduced in the previous section, and we prove that they are all equivalent. This allows us to conclude that the local read $tp(a)$, local output $lo(a)$, and global output $go(a{:}s)$ operations do not permit to distinguish among the different characteristics of the considered architectures.

Intuitively, the three styles of coordination architecture cannot be discriminated because the unique operation able to test the actual state of the data spaces, the $tp(a)$ primitive, is blocking and is not able to observe the different delays characterizing the three considered broadcasts. On the other hand, the multiplicity of data has no importance because, in the absence of consumption operators, it is not possible to observe the presence of multiple occurrences of the same datum.

In order to prove these results, we need to introduce some notation. Let $d \in DS_\gamma$ and $q \in DQ$ such that no data $\bar{a}$ (representing deletion) is present neither in d nor in q. It is easy to see that in the absence of messages of kind $\bar{a}$, the $\oplus$ operator is associative and commutative, i.e., $(d' \oplus a) \oplus b$ is equal to $(d' \oplus b) \oplus a$ for any data space d' and data values a and b. Given this observation, the following two definitions are well formed even if q denotes an equivalence class and not a fixed queue.

We denote by $d \leftarrow q$ the data space obtained after all the values in the queue q have been flushed in the data space d. We define it as follows:

$$
\begin{aligned}
d \leftarrow \varepsilon &= d \\
d \leftarrow q \odot m{:}s &= (d \oplus m) \leftarrow q
\end{aligned}
$$

Consider now $A \in Agents_\gamma$ and $q \in DQ$, we denote by $A \Leftarrow q$ the agent obtained after that all values in the queue q have been flushed in all the local data spaces of A. We define it by induction on the structure of A:

$$
\begin{aligned}
[P, d] \Leftarrow q \quad &= [P, d \leftarrow q] \\
(A_1 \parallel A_2) \Leftarrow q &= A_1 \Leftarrow q \parallel A_2 \Leftarrow q\,.
\end{aligned}
$$

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 $q_1 \odot q = q_2$ [18]. 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.

We also introduce a flattening operation which, given a multiset, produces a set containing elements which are present in the initial multiset. Formally, given the multiset d, its flattening is denoted by $\bar{d}$, where $\bar{d}(m) = 1$ if and only if $d(m) > 0$.

We divide our analysis in three parts: we first compare the locally and the globally delayed styles, then the globally delayed and the undelayed, and finally we compare sets and multisets.

4.1.1 Locally delayed is equivalent to globally delayed

We now consider the locally and the globally delayed styles of coordination architectures, without making any assumptions on the kind of data space (either set or multiset). Formally, we consider the architectures L_γ and G_γ without making any assumption on γ, which could be either σ or μ. We first define a universal embedding of G_γ in L_γ, then we consider the opposite embedding.

In order to embed the globally in the locally delayed architecture, we define the encoder $\mathcal{E}_{GL}$: $Conf(G_\gamma) \to Conf(L_\gamma)$ inductively as follows:

$$
\begin{aligned}
\mathcal{E}_{GL}([P, d], q) \quad &= [P, d, q] \\
\mathcal{E}_{GL}([P, d] \parallel A, q) &= [P, d, q] \parallel \mathcal{E}_{GL}(A, q)\,.
\end{aligned}
$$

This encoder simply replicates the shared queue as local queue for each process in the configuration. Furthermore, we take as decoder function $\mathcal{D}_{GL}$: $TS/\approx \to TS/\approx$ the identity.

Conversely, we can encode the locally in the globally delayed architectures by using the function $\mathcal{E}_{LG}$: $Conf(L_\gamma) \to Conf(G_\gamma)$ that is inductively defined by:

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

where $A', q' = \mathcal{E}_{LG}(C)$, $(q \sqcap q') \odot q_P = q$, and $(q \sqcap q') \odot q_C = q'$. In this case, we construct a shared queue as the greatest lower bound among all the local queues, and we flush in the data space of each agent the messages that are in in the local queue but not in the shared one. As in the previous case, we take as decoder $\mathcal{D}_{LG}$: $TS/\approx \to TS/\approx$ the identity function.

It is not difficult to see that both the embeddings satisfy (P1), (P2), and (P3). It remains to prove that the corresponding general diagram correctly commutes in both the cases. This is a consequence of two more general results that we will present in two theorems, stating that the rooted transition system of a configuration (taken from one of the two considered architectures) is the same (modulo weak bisimulation) as the rooted transition system of its encoding. The proofs of the theorems are reported in Appendix A.

Theorem 2 *Let $C \in Conf(G_\gamma)$; its transition system $(Conf(G_\gamma), C, \longrightarrow)$ is weakly bisimilar to the rooted transition system $(Conf(L_\gamma), \mathcal{E}_{GL}(C), \longrightarrow)$.*

Theorem 3 *Let $C \in Conf(L_\gamma)$; its transition system $(Conf(L_\gamma), C, \longrightarrow)$ is weakly bisimilar to the rooted transition system $(Conf(G_\gamma), \mathcal{E}_{LG}(C), \longrightarrow)$.*

4.1.2 Globally delayed is equivalent to undelayed

We now consider the globally delayed and the undelayed styles of architectures. Also in this case we do not make any assumptions on the kind of adopted data space (set or multiset). Formally, we consider the architectures G_γ and U_γ without making any assumption on γ, which could be either σ or μ.

The embedding of the undelayed in the globally delayed architecture is based on an encoding $\mathcal{E}_{UG}$: $Conf(U_\gamma) \to Conf(G_\gamma)$ which simply adds an empty common queue

$$\mathcal{E}_{UG}(A) = A, \varepsilon\,.$$

The corresponding decoder $\mathcal{D}_{UG}$: $TS/\approx \to TS/\approx$ is the identity function.

The opposite encoder $\mathcal{E}_{GU}$: $Conf(G_\gamma) \to Conf(U_\gamma)$ flushes the data in the common queue in each local data space:

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

Also in this case, the decoder $\mathcal{D}_{GU}\colon TS/\approx \to TS/\approx$ is the identity.

Both the embeddings satisfy the three considered properties. As in the previous section, we show that they also make the general diagram commute by presenting two more general theorems stating that the rooted transition system of a configuration is the same (modulo weak bisimulation) as the rooted transition system of its corresponding encoding. The proof of the theorems is reported in Appendix B.

Theorem 4 *Let $C \in Conf(U_\gamma)$; its transition system $(Conf(U_\gamma), C, \longrightarrow)$ is weakly bisimilar to the rooted transition system $(Conf(G_\gamma), \mathcal{E}_{UG}(C), \longrightarrow)$.*

Theorem 5 *Let $C \in Conf(G_\gamma)$; its transition system $(Conf(G_\gamma), C, \longrightarrow)$ is weakly bisimilar to the rooted transition system $(Conf(U_\gamma), \mathcal{E}_{GU}(C), \longrightarrow)$.*

4.1.3 Set is equivalent to multiset

Finally, we show that the choice between data spaces as sets or multisets is insignificant; more precisely, given an architectural style, the version with data spaces as sets and the version with data spaces as multisets are equivalent. We show this only for the locally delayed architecture, but the same approach can be simply applied to the other two kinds of architectures.

We first present a universal embedding of the locally delayed architecture with multisets in that with sets. The encoder $\mathcal{E}_{MS}\colon Conf(L_\mu) \longrightarrow Conf(L_\sigma)$ simply applies a flattening operation on all the local data spaces:

$$\begin{aligned}
\mathcal{E}_{MS}([P, d, q]) &= [P, \bar{d}, q] \\
\mathcal{E}_{MS}([P, d, q] \parallel C) &= [P, \bar{d}, q] \parallel \mathcal{E}_{MS}(C)
\end{aligned}$$

Also in this case the decoder $\mathcal{D}_{MS}\colon TS/\approx \longrightarrow TS/\approx$ is the identity function.

The opposite encoding $\mathcal{E}_{SM}\colon Conf(L_\sigma) \longrightarrow Conf(L_\mu)$ interprets the data spaces as multisets and leaves the rest unchanged. The decoder $\mathcal{D}_{SM}\colon TS/\approx \longrightarrow TS/\approx$ is still the identity function.

Also in this case the equivalence result is a consequence of two theorems (the proofs are reported in Appendix C).

Theorem 6 *Let $C \in Conf(L_\sigma)$; its transition system $(Conf(L_\sigma), C, \longrightarrow)$ is weakly bisimilar to the rooted transition system $(Conf(L_\mu), \mathcal{E}_{SM}(C), \longrightarrow)$.*

Theorem 7 *Let $C \in Conf(L_\mu)$; its transition system $(Conf(L_\mu), C, \longrightarrow)$ is weakly bisimilar to the rooted transition system $(Conf(L_\sigma), \mathcal{E}_{MS}(C), \longrightarrow)$.*

4.2 Comparison with test for absence

In this section we introduce a further primitive that is able to test for the absence of data in the local data space. We extend the syntax of processes by introducing a new prefix $ta(a)$ representing the test-for-absence of datum a

$$\alpha ::= \ \ldots \ | \ ta(a)\,.$$

The definition of the function *prc* which removes the sort information from processes is extended adding

$$prc(ta(a).P) \ = \ ta(a).prc(P)$$

Also the transition system specifications should be extended in order to deal with the new operator. We add the following three axioms to the specification for the locally, globally, and undelayed architectures, respectively.

(L10) $[ta(a).P, d, q] \longrightarrow [P, d, q]$ if $in(d, a) \neq tt$

(G10) $[ta(a).P, d], q \longrightarrow [P, d], q$ if $in(d, a) \neq tt$

(U9) $[ta(a).P, d] \longrightarrow [P, d]$ if $in(d, a) \neq tt$

Given the possibility to test the absence of data, most of the equivalence results proved in the previous section do not hold any more. The unique one which still holds is the impossibility to observe the multiplicity of data; only the ability to consume data permits to discriminate the choice between sets and multisets.

In order to prove a difference result between two architectures, we proceed by contraposition. We assume the possibility to universally embed one architecture in the other one; after, we present a configuration of one of the two architectures which has a different reasonable semantics with respect to its encoding.

4.2.1 Globally delayed is different from undelayed

We start by proving that there exists no universal embedding of the globally delayed architecture into the undelayed architecture. In order to prove this we use a technique adopted to prove also the other discriminating results reported in the remainder of the paper. We first assume by contraposition the existence of a universal embedding from one architecture to the other one, and then we show that there exists a particular initial configuration of the source architecture which is surely mapped on a configuration with a different behavior. In this first case we report the proof in details; in the subsequent proofs we only describe the discriminating initial configuration.

We do not make any assumption on the kind of adopted data spaces, thus γ can be either μ or σ.

Consider now the following processes

$$
\begin{aligned}
&P = go(a{:}s).go(b{:}t)\,,\\
&Q = tp(b).ta(a).go(\surd{:}u)\,.
\end{aligned}
$$

The initial configuration $C_G \in Conf(G_\gamma)$ comprising these two processes is

$$C_G = [P,\emptyset] \parallel [Q,\emptyset], \varepsilon\,.$$

It is easy to see that this configuration has the ability to produce the special datum $\surd$, namely $\surd \in ov(TS_{G_\gamma}(C_G))$. The production of $\surd$ can happen because the message $b{:}t$ may be delivered to the data space of Q before the message $a{:}s$ is delivered (they have different sorts, thus they can commute inside the common queue).

Let $\mathcal{E}$ be the encoding associated to the considered universal embedding, and let $C_U = \mathcal{E}(C_G)$. By property (P1) the active processes should be preserved by the encoding, thus the active components of C_U will be the same as P and Q, up to the use of different sorts. By property (P2) the configuration C_U is initial, thus it has empty data spaces. For this reason, the second process (corresponding to Q) is blocked until the first process (corresponding to P) produces datum b. At this moment, as the architecture is undelayed, the previously emitted a is already available in the local data space of the second process. As no consumption of a may be performed, this process is blocked trying to perform the operation $ta(a)$; thus the datum $\surd$ cannot be produced, namely $\surd \notin ov(TS_{U_\gamma}(C_U))$.

Hence, the property (P3) is not satisfied for the minimal set $\{\surd\}$ of intended values; for this reason the considered embedding cannot be universal.

4.2.2 Locally delayed is different from globally delayed

We now prove that there exists no universal embedding of the locally in the globally delayed architecture; again we do not make any assumption on the kind of adopted data spaces: γ can be either μ or σ.

In this case, it is enough to consider a configuration composed of three processes:

$$
\begin{aligned}
&P = go(a{:}s)\,,\\
&Q = tp(a).go(b{:}t)\,,\\
&R = tp(b).ta(a).go(\surd{:}u)\,.
\end{aligned}
$$

These processes are similar to the ones adopted above, with the difference that the data a and b are emitted by two different processes. It is easy to see that these processes are able to produce the datum $\surd$. This is because under the locally delayed architecture the processes may receive messages in different order; in particular, R could receive in its data space the message $b{:}t$ before $a{:}s$ (they are broadcast with two different sorts). On the other hand, this cannot happen under the globally delayed architecture because the message $a{:}s$ is surely received by R before message $b{:}t$.

4.2.3 Set is equivalent to multiset

The proof of the equivalence result between sets and multisets presented for the basic primitives, can be easily extended in order to deal with the test-for-absence operator.

The definition of the embeddings is the same as for the basic primitives. We do not report here the propositions and theorems that formally prove the equivalence results because they are the same as those presented in the previous section. Intuitively, this equivalence result holds because the condition that it is possible to test on the data spaces if the number of occurrences of a particular datum is equal or greater than 0. These conditions are invariants between each multiset d and its flattened version $\bar{d}$.

4.3 Comparison with local consumption

We now introduce a further primitive used to consume data in the local data space. Also in this case, it is enough to add a new prefix

$$\alpha ::= \ \ldots \ | \ ld(a)\,.$$

The definition of the function *prc* which removes the sort information from processes is extended adding

$$prc(ld(a).P) \ = \ ld(a).prc(P)$$

The new axioms specifying the behavior of the $ld(a)$ operator are:

(L11) $[ld(a).P, d, q] \longrightarrow [P, d \oplus \bar{a}, q]$ if $in(d, a) = tt$

(G11) $[ld(a).P, d], q \longrightarrow [P, d \oplus \bar{a}], q$ if $in(d, a) = tt$

(U10) $[ld(a).P, d] \longrightarrow [P, d \oplus \bar{a}]$ if $in(d, a) = tt$

Next we consider processes containing the basic primitives plus local deletion. The presence of the new operator permits to distinguish between data spaces as sets or multisets; furthermore, also the three styles of architectures become different, but only in the case of data spaces as sets.

We do not report the proofs of the equivalence results as they are essentially the same as those described above. Intuitively, one of the reasons for which the equivalences continue to hold only under the multiset approach, and not under the set approach, is that the associativity and commutativity of the operator $\oplus$ is important. In the presence of the data of kind $\bar{a}$ representing deletion, these properties hold only if we consider multisets. As an example, the two data spaces $(\{a\} \oplus a) \oplus \bar{a}$ and $(\{a\} \oplus \bar{a}) \oplus a$ are the same if we consider multisets but not if we consider sets.

4.3.1 Set is different from multiset

First we observe that the equivalence between sets and multisets does not hold anymore.

Consider the following process

$$P = lo(a).lo(a).ld(a).ld(a).go(\surd{:}s)\,.$$

If the structure of the data space is a multiset, an initial configuration (of one of the three architectures) composed of this process has the ability to produce the datum $\surd$. Otherwise, if the data space is a set, it is not possible to perform a sequence composed of two subsequent local delete operations on the same kind of datum (in the case, as we suppose, that no other instances of the datum can be emitted by other processes in the environment). On the other hand, if the data space is a set it is not possible for the process P to perform both the consumption operations, thus $\surd$ cannot be observed.

4.3.2 Globally delayed is different from undelayed (under sets)

If data spaces are sets, then the globally delayed architecture is different from the undelayed. Consider the following processes

$$\begin{aligned}
&P = go(a{:}s).go(a{:}t).go(b{:}u)\,,\\
&Q = tp(b).ld(a).ld(a).go(\surd{:}v)\,.
\end{aligned}$$

Under the globally delayed architecture $\surd$ may be produced because the messages can be received in an order different with respect to their production; thus, datum b may be received by Q before the data a. On the other hand, this cannot happen under the undelayed architecture.

4.3.3 Locally delayed is different form globally delayed (under sets)

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

$$\begin{aligned}
&P = go(a{:}s).go(a{:}t)\,,\\
&Q = ld(a).ld(a).go(b{:}u)\,,\\
&R = tp(b).ld(a).ld(a).go(\surd{:}v)\,.
\end{aligned}$$

Also in this case, it is not difficult to see that $\surd$ can be observed only under the locally delayed architecture because the third process R may receive the messages produced by P in a different order with respect to the second process Q.

4.4 Comparison with global consumption

Finally we give the possibility to globally consume data by broadcasting a request for deletion. This is obtained by introducing a new prefix

$$\alpha \ ::= \ \ldots \ \mid \ gd(a{:}s)\,.$$

The definition of the function *prc* which removes the sort information from processes is extended adding

$$prc(gd(a{:}s).P) \;=\; gd(a).prc(P)$$

The new axioms specifying the behavior of the $gd(a{:}s)$ operator are:

(L12) $[gd(a{:}s).P, d, q] \xrightarrow[\bar{a}:s]{\bar{a}} [P, d, \bar{a}{:}s \odot q]$ if $in(d, a) = tt$

(G12) $[gd(a{:}s).P, d], q \xrightarrow{\bar{a}} [P, d], \bar{a}{:}s \odot q$ if $in(d, a) = tt$

(U11) $[gd(a{:}s).P, d] \xrightarrow[\bar{a}]{\bar{a}} [P, d \oplus \bar{a}]$ if $in(d, a) = tt$

The presence of this operator permits us to distinguish among the six considered architectures.

4.4.1 Globally delayed is different from undelayed

We first show a particular configuration of the globally delayed architecture that has no satisfactory encoding in the undelayed architectures. We do not make any assumption on the kind of data spaces (which could be either sets or multisets). Consider the following processes

$$P = go(a{:}s).gd(a{:}t).go(b{:}u)\,,$$
$$Q = tp(b).tp(a).go(\surd{:}v)\,.$$

In this case it is not difficult to see that $\surd$ may be produced only under the globally delayed architecture as under this architecture the message $\bar{a}{:}t$ may be received by Q after message $b{:}u$.

4.4.2 Locally is different from globally delayed

In order to prove that locally is different from globally delayed (under both the set and the multiset approaches), we have to slightly change our proof technique by using two distinct observable messages $\surd_1$ and $\surd_2$. Consider the processes

$$P = go(a{:}s).gd(a{:}t).go(b{:}t)\,,$$
$$Q = tp(a).tp(b).go(c{:}u).tp(a).go(\surd_1, v)\,,$$

$$R = tp(c).tp(a).go(\surd_2{:}z)\,.$$

Observe that the global delete of a and the global output of b are executed using the same sort t. Thus, the two messages will be received according to the order of emission by all the processes.

If we embed the above processes in an initial configuration of the locally delayed architecture, we have that $\surd_2$ may be produced while $\surd_1$ cannot. The data value $\surd_1$ cannot be produced because at the moment Q tests the presence of b, the datum a has been already consumed in its data space (because the message $\bar{a}{:}t$ is delivered before the message $b{:}t$). On the other hand, $\surd_2$ may be produced because in the locally delayed architecture the processes may receive messages in different orders; e.g., R could receive the messages regarding the data value a after the message $c{:}u$ produced by Q.

Suppose that we can embed the above processes (even with different sorts) in an initial configuration of the globally delayed architecture. As sorts may be changed by the encoding, we have to deal with two different cases: the two broadcast operations which adopt the sort t continue to use the same sort, or two different sorts are adopted. In the first case, we have that the message $\surd_2$ cannot be produced because at the moment the third process tests the presence of c, the datum a has been already introduced and is also consumed. In the second case, we have that the message $\surd_1$ may be produced because the global delete operation could be delayed.

Thus, we can conclude that the embedding cannot preserve any reasonable semantics defined on a set of intended data values comprising at least $\{\surd_1, \surd_2\}$.

4.4.3 Set is different from multiset

In order to prove that the set approach is different from the multiset approach, we consider two different examples, one for the undelayed architectures and one for the globally and locally delayed.

For the undelayed architectures we can extend the reasoning adopted for the local delete operator using the following process

$$P = lo(a).lo(a).gd(a{:}s).gd(a{:}t).go(\surd{:}u)\,.$$

For the other two architectures we consider the following three processes

$$P = lo(a).lo(a).go(b{:}s).tp(d).tp(a).go(\surd_1{:}t)\,,$$
$$Q = tp(b).lo(a).gd(a{:}u).go(c{:}v).tp(d).tp(a).go(\surd_2{:}z)\,,$$

$$R = tp(c).go(d{:}u)\,.$$

Observe that the global delete of a and the global output of c adopt the same sort u. For this reason, the message $\bar{a}{:}u$ is received before $d{:}u$ by all processes.

If we embed the above processes in an initial configuration of a delayed architecture (either globally or locally) with data spaces as multisets, it is easy to see that the message $\surd_1$ may be produced, while $\surd_2$ may not. The message $\surd_1$ may be produced because two instances of a are introduced in the local data space of P and only one delete operation is allowed. The message $\surd_2$ cannot be produced by process Q because, at the moment it tests the presence of message d, it is ensured that the global delete operation on a has been already performed (this is because the messages $\bar{a}{:}u$ and $d{:}u$ use the same sort).

Suppose now to embed the above processes (even with different sorts) in a delayed architecture (either globally or locally) with data spaces as sets. We have to deal with two different cases: in the first case we suppose that the global delete of a and the global output of d adopt the same sort, while in the second case we consider two different sorts. In the first case we have that $\surd_1$ cannot be produced because when the first process tests the presence of d, it is ensured that the consumption of a has already been performed. Thus, as we are dealing with sets, no a is available and the process blocks trying to perform the test for presence of a. On the other hand, in the second case, $\surd_2$ may be performed because the global consumption of a can be arbitrarily delayed.

Thus, we can conclude that the embedding cannot preserve any reasonable semantics defined on a set of intended data values comprising at least $\{\surd_1, \surd_2\}$.

5 Three coordination models

We have introduced our framework as a tool for the definition and comparison of coordination models. In order to show how to use the famework, we exploit the coordination architectures presented in the previous sections in order to describe and discuss well-known coordination models taken from both the data-driven and the control-driven families. A description of these two families can be found in [22].

Here, we only recall that in the data-driven approach coordination is obtained via the exchange of data through shared data spaces, while in the control-driven family the emphasis is on the internal state of the active components and not on passive shared data. For example, many control-driven languages adopt an event communication mechanism: when a process reaches a state that could be of interest to other components of the system, it raises an event; when the processes receive the communication of the occurred event, they may react by activating new processes or executing specific reactions.

Even if it is common to think that the two families of coordination models embody very different features and characteristics, here we show that the framework we have introduced permits to capture common features, at least at the level of the adopted coordination architectures. For example, we show that two abstract representations of the data-driven model Splice [6] and the control-driven language Manifold [4] embody the same coordination architecture.

Another interesting discussion is related to the adoption of our framework in order to prove the possibility to implement the considered coordination models onto different architectures; for example, we present a new kind of globally delayed architectures which can be seen as an alternative target architecture for the coordination language Linda.

5.1 The shared data space model of Linda

We start with a coordination model embedding the coordination primitives of Linda [15]. It is inspired by a more general Linda based calculus presented in [12]. Linda uses the abstraction of a shared multiset of data; this common space can be accessed via primitives which permits the introduction, the consumption, and the test for presence of a datum. Furthermore, also two non-blocking input and read operators are considered, which can terminate also if no interesting data are actually available in the data space. In this case, a test-for-absence of the considered datum is realized.

Formally, we define a Linda-like language L_1 using the following grammar

$$T ::= U \mid T \parallel T$$
$$U ::= \mathit{end} \mid \mathit{out}(a).U \mid \mathit{rd}(a).U \mid \mathit{in}(a).U \mid \mathit{rdp}(a)?U\_U \mid \mathit{inp}(a)?U\_U$$

where a is an element of an abstract set Val of values. Note that we only allow non-nested programs, in which the parallel composition does not occur after an action prefix.

The coordination primitives $\mathit{out}(a)$, $\mathit{rd}(a)$, and $\mathit{in}(a)$ are used for the production, for the test for presence, and for the consumption of datum a, respectively. The other primitives $\mathit{rdp}(a)$ and $\mathit{inp}(a)$ require the indication of two possible continuations; the first is chosen if the test for presence or consumption operation can be performed on a, while the second is activated if no a is available in the data space. For this reason, we say that these non-blocking read and input operations embody a test for absence mechanism.

The corresponding Linda-like coordination model is obtained by mapping the language L_1 to the undelayed architecture with multisets. Even if our architecture considers local data spaces and not a shared common one, we can see that the local data spaces are distributed consistent copies of the same shared space. Indeed, it is not hard to see that in the undelayed architecture, because we are dealing with a synchronous broadcast, all the local data spaces are kept consistent provided that no local operations, for consuming or producing data, are executed.

We define our Linda-like coordination model as the function $\mathcal{S}_1 : L_1 \rightarrow \mathit{Conf}(U_\mu)$ inductively given by

$$\mathcal{S}_1(U) = [\llbracket U \rrbracket, \emptyset] \qquad \mathcal{S}_1(T_1 \parallel T_2) = \mathcal{S}_1(T_1) \parallel \mathcal{S}_1(T_2)\,,$$

where

$$\begin{array}{ll} \llbracket \mathit{end} \rrbracket \quad = 0 & \llbracket \mathit{out}(a).U \rrbracket = \mathit{put}(a{:}s).\llbracket U \rrbracket \\ \llbracket \mathit{rd}(a).U \rrbracket = \mathit{tp}(a).\llbracket U \rrbracket & \llbracket \mathit{in}(a).U \rrbracket \ = \mathit{gd}(a{:}s).\llbracket U \rrbracket \end{array}$$

$$\begin{aligned} \llbracket \mathit{rdp}(a)?U_1\_U_2 \rrbracket &= \mathit{tp}(a).\llbracket U_1 \rrbracket + \mathit{ta}(a).\llbracket U_2 \rrbracket \\ \llbracket \mathit{inp}(a)?U_1\_U_2 \rrbracket &= \mathit{gd}(a{:}s).\llbracket U_1 \rrbracket + \mathit{ta}(a).\llbracket U_2 \rrbracket\,. \end{aligned}$$

In the definition of this coordination model we have used a fixed sort s in each broadcast action, but, as we have already discussed, this has no importance in the undelayed architecture.

In the previous section we proved that the considered coordination architectures are different if global consumption is adopted; for this reason we can conclude that we have no direct implementation of Linda in terms of the other architectures.

However, if we do not consider the non-blocking primitives *inp* and *rdp*, we could implement the language on a new kind of architecture presenting an asymmetric treatment of input (which remains undelayed) and output (which becomes globally delayed). This architecture has been inspired us by an interpretation of Linda presented in [13], where the Linda based calculus introduced in [12] is equipped with an alternative *unordered* semantics. Under the unordered approach, an output is composed of two separate phases: first the emission of the new datum (corresponding to the execution of the output operation) and then the rendering (the actual introduction of the datum in the shared data space). The unordered approach contrasts with the *ordered* one, according to which the emission and the rendering form a unique atomic action. The output operation under the unordered approach corresponds to the one modeled in the undelayed architecture.

In our framework, the unordered version of the calculus can be obtained by exploiting the globally delayed architecture with a different semantics for the input operator. We denote this new style of coordination architecture with G^A_γ where the index A represents the asymmetry of the architecture. The set of configurations $Conf(G^A_\gamma)$ is defined as in the globally delayed architecture; while in the transition specification we have to substitute rule (G12) with the following one:

(G12') $[gd(a{:}s).P, d], q \xrightarrow[\bar{a}]{\bar{a}} [P, d \oplus \bar{a}], q$ if $in(d, a) = tt$

The synchrony of the global consumption follows from the fact that the datum $\bar{a}$ is atomically introduced in all the local data spaces; this is ensured by adopting the label $\bar{a}$ not only as observable but also as effect of the transition. Observe also that the emitted datum is not introduced in the common queue but it is directly introduced in the data spaces.

The new architecture G^A_γ has the interesting property that it is equivalent to the undelayed architecture U_γ, provided that the data spaces are multisets (thus the equivalence holds only for $\gamma = \mu$) and the global input and output, and the test for presence. We do not give here the formal definition of the universal embeddings between the architectures, and the formal proof of the equivalence result, because they are similar to those presented in the previous section.

This new result allows us to state that the Linda-like coordination model of Linda defined above can be implemented also in terms of the new asymmetric globally delayed architecture G^A_μ. This is true only if we do not consider the *inp* and *rdp* operators embodying a test for absence mechanism. Thus, we can conclude that, in the absence of the non-blocking operations, the ordered and the unordered approach are interchangeable. On the other hand, in the presence of these operators, a strong discrimination between the ordered and the unordered semantics holds: in [14] it is proved that a simple Linda based calculus is Turing powerful under the first approach while this is not the case under the second one.

5.2 *The distributed data space model of Splice*

As a second example we discuss the distributed data space model adopted in the coordination language Splice [6]. In our presentation, we have been inspired by a Splice based calculus presented in [8].

The main differences between Linda and Splice are that Splice adopts sets instead of multisets, and only local consumption is allowed. Moreover, no operators able to test the absence of data are considered.

Splice has been developed originally in order to implement an information store for systems like radar control systems. If two radar systems communicate the actual position of an airplane, it is not necessary to maintain two copies of this piece of information; for this reason sets are adopted instead of multisets. Moreover, deletion is permitted to realize garbage collection of information which is no more interesting. As a process cannot know if the same information is no more of interest to the other processes, the consumption is executed only locally.

Formally, we define a Splice-like language L_2 using the following grammar

$$T ::= U \mid T \parallel T$$

$$U ::= end \mid put(a).U \mid read(a).U \mid get(a).U$$

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

Splice adopts a particular broadcast discipline: data of the same kind are received by all processes in the same order as they were broadcast, while data of different kinds can commute during the broadcast communication. This is represented in our framework by assuming the existence of a function $sort(a)$ which, given a datum a, returns a sort representing the kind of the datum a. This function is used when a new datum is broadcast in order to know the corresponding sort.

As Splice adopts sets instead of multisets and permits local consumption, we can conclude that the choice of the architecture to adopt is important. Indeed, we have proved in the previous section that, under these assumptions, the three coordination architectures that we have defined are all different. In this presentation, we adopt the locally delayed architecture, which is closer to the implementation of the Splice system running on a distributed system, even if there exists abstract representations of Splice which consider a shared data space with local views (see, e.g., [8]).

We define our Splice-like coordination model as the function $\mathcal{S}_2 : L_2 \rightarrow Conf(L_\sigma)$

inductively given by

$$\mathcal{S}_2(U) = [\llbracket U \rrbracket, \emptyset, \varepsilon] \qquad \mathcal{S}_2(T_1 \parallel T_2) = \mathcal{S}_2(T_1) \parallel \mathcal{S}_2(T_2)\,,$$

where

$$\begin{array}{llll} \llbracket end \rrbracket & = 0 & \llbracket put(a).U \rrbracket & = put(a{:}sort(a)).\llbracket U \rrbracket \\ \llbracket read(a).U \rrbracket & = tp(a).\llbracket U \rrbracket & \llbracket get(a).U \rrbracket & = ld(a).\llbracket U \rrbracket\,. \end{array}$$

5.3 The event-driven model of Manifold

We finish our presentation by moving to the family of control-driven coordination languages. In particular, we consider Manifold [4], one of the main representatives of this family.

Manifold [4] is a coordination language which permits to write software components (called coordinators) which have the responsibility to coordinate other computing processes (called workers). A worker receives data from input ports, performs a computation on them, and produces new data which are emitted through output ports. If, during the computation, a worker reaches an internal state which could be of interest to other components of the system, it raises an event which describes the reached internal state. On the other hand, coordinators do not perform computation, but simply manages the structure of the system by connecting the ports of the workers using ordered channels called streams. Upon the reception of events, a coordinator may change the actual structure of the system by creating or removing workers, coordinators, and streams, or by changing the endings of the available streams reconnecting them to other ports.

A complete presentation of the operational semantics of Manifold can be found in [5]. Here, we simply consider the description of the coordinators (and not the workers). The following language L_3 is essentially a simplification of the syntax of the Manifold language as presented in [5]

$$\begin{aligned} T &::= \textstyle\prod_{p=1}^{k}[p, U] \\ U &::= (e,p)?V \mid (*e,*p)?V \mid (*e,p)?V \mid (e,*p)?V \mid U+U \\ V &::= end \mid U \mid raise(e).V \mid post(e).V \end{aligned}$$

where $p \in PName$ and $e \in EName$ denote process names and event names, respectively. A Manifold system T is the parallel composition of n components $\langle p, U\rangle$ where p is the name of the component and U the corresponding program. The program U is a choice among possible behaviors. Each behavior is composed of an event pattern, e.g. (e, p) or $(*e, *p)$, and a reaction V. The event pattern describes the kind of events that are able to activate the corresponding reaction: for example, (e, p) represents the raising of an event e by process p. When we put a $*$ in front of e or p, it means that we are not interested in a particular event e or a source process p, but a generic event or process name. In this case, the name e or p preceded by the $*$ is a formal name, which is substituted by the actual name when the corresponding reaction is activated.

The actions that a reaction V may perform are the broadcast of a new event e (via the $raise(e)$ primitive) or the introduction of the event e inside the local event memory (via the $post(e)$ primitive). Moreover, a reaction V may be also a new program (i.e., the choice among other possible behaviors).

Each component has an event memory (which is a set) that contains all the received events represented by pairs $\langle e, p\rangle$ where e is the name of the event and p is the name of the program which produced the event. When an event activates a reaction, it is removed from the event memory.

In Manifold, the communication of events is realized via a broadcast mechanism which preserves the order of emission of events which have the same source; in other words, two events raised by the same process are received by all processes in the same order as they were produced. This is realized in our framework by associating to each broadcast datum a sort representing the identity of the source process.

Our Manifold-like coordination model is defined by the following function $\mathcal{S}_3{:}L_3 \rightarrow Conf(L_\sigma)$ inductively given by

$$\mathcal{S}_3(\Pi_{p=1}^k[p, U]) = \Pi_{p=1}^k[[\![U]\!]_p, \emptyset, \varepsilon] \ ,$$

where

$$\begin{array}{llll}
[\![U_1 + U_2]\!]_p & = [\![U_1]\!]_p + [\![U_2]\!]_p & [\![end]\!]_p & = 0 \\
[\![(e, q)?\,V]\!]_p & = ld(\langle e, q\rangle).[\![V]\!]_p & & \\
[\![(*e, *q)?\,V]\!]_p & = \sum_{x\in EName, y\in PName} ld(\langle x, y\rangle).([\![V]\!]_p\, x/e\; y/q) & & \\
[\![(*e, q)?\,V]\!]_p & = \sum_{x\in EName} ld(\langle x, q\rangle).([\![V]\!]_p\, x/e) & & \\
[\![(e, *q)?\,V]\!]_p & = \sum_{y\in PName} ld(\langle e, y\rangle).([\![V]\!]_p\, y/q) & & \\
[\![raise(e).\,V]\!]_p & = go(\langle e, p\rangle{:}p).[\![V]\!]_p & [\![post(e).\,V]\!]_p & = lo(\langle e, p\rangle).[\![V]\!]_p
\end{array}$$

where $P\ a/b$ corresponds to P where all instances of b are substituted with a.

6 Conclusions and Future Work

In this paper we have provided a framework for the formal representation of coordination models. The main contribution of our proposal consists of the identification and formalization of two separated levels: the *coordination language*, which is the linguistic support for describing the needed inter-process interaction, and the *coordination architecture*, describing the mechanisms adopted in a structured system to realize the intended interactions. In this scenario, a *coordination model* is simply a function mapping a coordination language to a coordination architecture.

Besides the introduction of a new approach for the formal definition of coordination models, we refined the notion of language embedding introduced by Shapiro [23] in order to deal with coordination architectures. This provides a useful framework for the relative comparison of coordination languages, models, and architectures; for example, it is possible to compare architectures in isolation, i.e., independently of the coordination models embedding them.

The proposed framework is used to described three commonly adopted coordination architectures, to prove equivalence and difference results among them, and by adopting the described architectures for the description of well known coordination models.

Future work may move in two different directions: the introduction of new coordination primitives and the representation of new architectures. For example, in related papers [9,10] we have considered a global delete operator, which deletes synchronously in the local data space and asynchronously in the remote spaces, and also an operator for the dynamic creation of new processes. Regarding process creation, there are several issues to be addressed; for example, should the data space to be associated to a newly created process be either empty or equal to the data space of the process which performed the creation operation? There are more interesting architectures that we want to model in our framework, for example architectures that adopt other kinds of broadcast mechanisms (e.g., a causal broadcast) or architectures based on a different relation between active processes and their data repository. Another example is the modeling of agents: we might associate an identifier to pairs composed of a process and a data space, and allow inter-agent communication exploiting names of agents.

A Proofs of Theorems 2 and 3

To prove the theorems we need several preliminary results.

The following fact describes an alternative definition of the encoding function $\mathcal{E}_{LG}$ which is useful in the following.

Fact 8 *Consider a configuration $C \in Conf(L_\gamma)$ and its encoding $\mathcal{E}_{LG}(C)$. Let q_C be the greatest lower bound among all the local queues in C; we have that there exist indexed data queues q_i such that*

$$C = \prod_i [P_i, d_i, q_C \odot q_i] \quad \textit{and} \quad \mathcal{E}_{LG}(C) = \prod_i ([P_i, d_i] \Leftarrow q_i), q_C \, .$$

Here we use $\prod_i [P_i, d_i, q_i]$ and $\prod_i [P_i, d_i]$ to denote the parallel composition of the indexed terms $[P_i, d_i, q_i]$ or $[P_i, d_i]$, respectively.

The following lemma considers the configurations of the globally delayed architectures and describes properties of their possible transitions. In this case we consider the effects of the transitions and not the observables.

The first item of the lemma shows that if a transition with effect τ is performed without altering the global queue, then the same transition can be executed in a configuration in which we change the common queue and we flush new data inside the local data spaces. As we consider no consumption operators we assume that the new flushed data are not of the kind $\bar{a}$. This property intuitively holds because the operators that we consider are monotonic, i.e., if they can be performed with the data spaces in a particular state, they can be performed also if new data are added to the data space.

The second item considers the case of the execution of a global output operation; also this operation can be performed even if the data spaces and the common queue are changed.

Finally, the third item states that a delivery action has a unique possible behavior corresponding to the introduction of the new message inside all the local data spaces.

Lemma 9 *Given a configuration $A, q \in Conf(G_\gamma)$ we have that*

(1) if $A, q \xrightarrow[\tau]{o} A', q$ then $(A \Leftarrow q', q'') \xrightarrow[\tau]{o} (A' \Leftarrow q', q'')$ for any queue q' and q'' such that q' does not contain any message $\bar{a}$,

(2) if $A, q \xrightarrow[\tau]{o} A', m{:}s \odot q$ then $(A \Leftarrow q', q'') \xrightarrow[\tau]{o} (A' \Leftarrow q', m{:}s \odot q'')$ for any queue q' and q'',

(3) *if* $A, q \xrightarrow[m]{o} A', q'$ *then there exists* s *such that* $q = q' \odot m{:}s$ *and* $A' = (A \Leftarrow m{:}s)$.

PROOF. The three assertions can be independently proved by induction on the length of the derivation of the considered transition. $\square$

The following two propositions take into account pairs of configurations of the kind $(\mathcal{E}_{LG}(C), C)$ where $C \in Conf(L_\gamma)$. The first proposition assumes that the configuration $\mathcal{E}_{LG}(C)$ performs a transition, while the second considers the case in which a transition is performed by C. In both cases, we show that the opposite configuration may perform a (weak) transition which mimics the considered transition.

Proposition 10 *Consider* $C \in Conf(L_\gamma)$ *and its encoding* $\mathcal{E}_{LG}(C) = A_C, q_C$*; we have that*

(1) *if* $A_C, q_C \xrightarrow[\ell]{m} A'_C, q'_C$ *then there exists an effect* ℓ' *such that* $C \xrightarrow[\ell']{m} C'$ *where* $\mathcal{E}_{LG}(C') = A'_C, q'_C$,

(2) *if* $A_C, q_C \xrightarrow[\ell]{} A'_C, q'_C$ *then* $C \Longrightarrow C'$ *where* $\mathcal{E}_{LG}(C') = A'_C, q'_C$.

PROOF. We consider only item (2); the other case is treated similarly.

Given $A_C, q_C \xrightarrow[\ell]{} A'_C, q'_C$ we consider two cases.

If $\ell = \tau$, we can prove by induction on the length of the derivation of $A_C, q_C \longrightarrow A'_C, q'_C$ that also $C \longrightarrow C'$ with $\mathcal{E}_{LG}(C') = A'_C, q'_C$.

If $\ell \neq \tau$, the configuration C may require the execution of more than one step. By (3) of Lemma 9 we have that there exists a broadcastable message $m{:}s$ such that $q_C = q'_C \odot m{:}s$ and $(A'_C = A_C \Leftarrow m{:}s)$.

By Fact 8 there exist a data queue q_C and data queues q_i such that $C = \prod_i [P_i, d_i, q_C \odot q_i]$ and $A_C = \prod_i([P_i, d_i] \Leftarrow q_i)$. As $q_C = q'_C \odot m{:}s$, thus also $C = \prod_i [P_i, d_i, q'_C \odot m{:}s \odot q_i]$. The configuration C can perform a sequence of unobservable transitions corresponding to the local delivery of all the data in the part of the queues $m{:}s \odot q_i$. Let C' be the reached configuration; we have that $C' = \prod_i [P_i, d_i \leftarrow (m{:}s \odot q_i), q'_C]$. We have that, $\mathcal{E}_{LG}(C') = \prod_i [P_i, d_i \leftarrow (m{:}s \odot q_i)], q'_C$ which is the same as $(\prod_i([P_i, d_i] \Leftarrow q_i)) \Leftarrow m{:}s, q'_C$. This term is equal to A'_C, q'_C because we proved that $A'_C = (A_C \Leftarrow m{:}s)$. $\square$

Proposition 11 *Consider* $C \in Conf(L_\gamma)$ *and its encoding* $\mathcal{E}_{LG}(C) = A_C, q_C$*; we have that*

(1) if $C \xrightarrow[\ell]{m} C'$ then there exists a sort s such that $\ell = m{:}s$ and $A_C, q_C \xrightarrow{m} A'_C, m{:}s \odot q_C$ with $\mathcal{E}_{LG}(C') = A'_C, m{:}s \odot q_C$,
(2) if $C \xrightarrow[\ell]{} C'$ then $\ell = \tau$ and one of the following holds:
- $\mathcal{E}_{LG}(C) = \mathcal{E}_{LG}(C')$,
- $A_C, q_C \xrightarrow[\tau]{} A'_C, q_C$ with $\mathcal{E}_{LG}(C') = A'_C, q_C$,
- there exists a broadcastable message $m{:}s$ such that $A_C, q_C \xrightarrow[m]{} (A_C \Leftarrow m{:}s, q'_C)$ with $q_C = q'_C \odot m{:}s$ and $\mathcal{E}_{LG}(C') = (A_C \Leftarrow m{:}s, q'_C)$.

PROOF. The proof is by induction on the length of the derivation of the considered transition $C \xrightarrow[\ell]{m} C'$ or $C \xrightarrow[\ell]{} C'$.

In the base case an axiom is used to derive the considered transition; the thesis can be trivially proved. In the inductive case we proceed by case analysis on the last rule applied. We present only the case of rules (L7) and (L8) (the cases (L6) and (L9) are trivial: we are reasoning up to structural congruence).

If the last applied rule is (L7), then we have $C = [P, d, q] \parallel C_1$ and $C' = [P, d, q] \parallel C'_1$ with $C_1 \xrightarrow{o} C'_1$. Observe that $o = \tau$, otherwise the effect of the transition should be different from τ. Thus, the item (2) of the proposition is considered.
If $\mathcal{E}_{LG}(C) = \mathcal{E}_{LG}(C')$ then the thesis is proved. Let $\mathcal{E}_{LG}(C) \neq \mathcal{E}_{LG}(C')$; this implies also $\mathcal{E}_{LG}(C_1) \neq \mathcal{E}_{LG}(C'_1)$. As the effect of the transition from C_1 to C'_1 is τ, then the item (2) of inductive hypothesis should be considered; only two cases remain to be analyzed.

- $\mathcal{E}_{LG}(C_1) = A_{C_1}, q_{C_1}$ with $A_{C_1}, q_{C_1} \xrightarrow{o} A_{C'_1}, q_{C'_1}$ and $\mathcal{E}_{LG}(C'_1) = A_{C'_1}, q_{C_1}$.
  By definition we have $\mathcal{E}_{LG}(C) = ([P, d] \Leftarrow q_P \parallel A_{C_1} \Leftarrow q'), q \sqcap q_{C_1}$ where $(q \sqcap q_{C_1}) \odot q_P = q$ and $(q \sqcap q_{C_1}) \odot q' = q_{C_1}$. By (1) of Lemma 9 we have that $(A_{C_1} \Leftarrow q', q \sqcap q_{C_1}) \longrightarrow (A_{C'_1} \Leftarrow q', q \sqcap q_{C_1})$.
  Hence, by rule (G7) we have also $([P, d] \Leftarrow q_P \parallel A_{C_1} \Leftarrow q'), q \sqcap q_{C_1} \longrightarrow ([P, d] \Leftarrow q_P \parallel A_{C'_1} \Leftarrow q'), q \sqcap q_{C_1}$ which is the same as $\mathcal{E}_{LG}(C')$.
- $\mathcal{E}_{LG}(C_1) = A_{C_1}, q_{C_1}$ and there exists a broadcastable message $m{:}s$ such that $A_{C_1}, q_{C_1} \xrightarrow[m]{} (A_{C_1} \Leftarrow m{:}s, q'_{C_1})$ with $q_{C_1} = q'_{C_1} \odot m{:}s$ and $\mathcal{E}_{LG}(C'_1) = (A_{C_1} \Leftarrow m{:}s, q'_{C_1})$.
  By definition of $\mathcal{E}_{LG}$ we have $\mathcal{E}_{LG}(C) = ([P, d] \Leftarrow q_P \parallel A_{C_1} \Leftarrow q'), q \sqcap q_{C_1}$ where $(q \sqcap q_{C_1}) \odot q_P = q$ and $(q \sqcap q_{C_1}) \odot q' = q_{C_1}$.
  There are two cases to analyze.
  - $q \sqcap q_{C_1} = q \sqcap q'_{C_1}$.
    As $(q \sqcap q_{C_1}) \odot q' = q_{C_1} = q'_{C_1} \odot m{:}s$, then $q' = q'' \odot m{:}s$ for some q''. Thus $(q \sqcap q_{C_1}) \odot q'' = q'_{C_1}$ and also $(A_{C_1} \Leftarrow q') = ((A_{C_1} \Leftarrow m{:}s) \Leftarrow q'')$.
    Hence, we have that $\mathcal{E}_{LG}(C) = ([P, d] \Leftarrow q_P \parallel (A_{C_1} \Leftarrow m{:}s) \Leftarrow q''), q \sqcap q'_{C_1}$ which, by definition of $\mathcal{E}_{LG}$, is equal to $\mathcal{E}_{LG}(C')$ (because $(q \sqcap q'_{C_1}) \odot q_P = q$ and $(q \sqcap q'_{C_1}) \odot q'' = q'_{C_1}$).

- $q \sqcap q_{C_1} \neq q \sqcap q'_{C_1}$.
  As $q'_{C_1} \odot m{:}s = q_{C_1}$ we have $q'_{C_1} \sqsubseteq q_{C_1}$. It is not difficult to see that, in order to have two different greatest lower bounds, $q_{C_1} \sqsubseteq q$; i.e., $q = q_{C_1} \odot q''$ for some q''. Thus $q \sqcap q_{C_1} = q_{C_1}$ and $q \sqcap q'_{C_1} = q'_{C_1}$.
  Hence, we have that $\mathcal{E}_{LG}(C) = ([P, d] \Leftarrow q_P \,\|\, A_{C_1} \Leftarrow q'), q_{C_1}$. By definition of $\mathcal{E}_{LG}$, we have that $q_P = q''$ and $q' = \varepsilon$. Moreover, as $q_{C_1} = q'_{C_1} \odot m{:}s$, the last message $m{:}s$ can be delivered to all the data spaces via the transition $([P, d] \Leftarrow q'' \,\|\, A_{C_1}), q'_{C_1} \odot m{:}s \xrightarrow[m]{} (([P, d] \Leftarrow q'') \Leftarrow m{:}s \,\|\, A_{C_1} \Leftarrow m{:}s), q'_{C_1}$. The last term is the same as $\mathcal{E}_{LG}(C')$ because $(q \sqcap q'_{C_1}) \odot m{:}s \odot q'' = q$ and also $(q'_{C_1} \sqcap q'_{C_1}) \odot \varepsilon = q'_{C_1}$.

If the last applied rule is (L8) we have $C = [P, d, q] \,\|\, C_1$ and $C' = [P, d, m{:}s \odot q] \,\|\, C'_1$ with $C_1 \xrightarrow[m:s]{o} C'_1$. By the induction hypothesis we have that $o = m$ and $A_{C_1}, q_{C_1} \xrightarrow{m} A'_{C_1}, m{:}s \odot q_{C_1}$ with $\mathcal{E}_{LG}(C'_1) = A'_{C_1}, m{:}s \odot q_{C_1}$.
By definition of $\mathcal{E}_{LG}$ we have that $\mathcal{E}_{LG}(C) = ([P, d] \Leftarrow q_P \,\|\, A_{C_1} \Leftarrow q'), q \sqcap q_{C_1}$ where $(q \sqcap q_{C_1}) \odot q_P = q$ and $(q \sqcap q_{C_1}) \odot q' = q_{C_1}$.
By (2) of Lemma 9 we have that also $A_{C_1} \Leftarrow q', q \sqcap q_{C_1} \xrightarrow{m} A'_{C_1} \Leftarrow q', m{:}s \odot (q \sqcap q_{C_1})$. Hence, by rule (G7) we have also $([P, d] \Leftarrow q_P \,\|\, A_{C_1} \Leftarrow q'), q \sqcap q_{C_1} \xrightarrow{m} [P, d] \Leftarrow q_P \,\|\, A'_{C_1} \Leftarrow q'), m{:}s \odot (q \sqcap q_{C_1})$.
It remains to show that the last term is the same as $\mathcal{E}_{LG}(C')$. This follows from the two following three observations.
First, $(m{:}s \odot q) \sqcap (m{:}s \odot q_{C_1}) = m{:}s \odot (q \sqcap q_{C_1})$. Second, $m{:}s \odot (q \sqcap q_{C_1}) \odot q_P = m{:}s \odot q$. Third, $m{:}s \odot (q \sqcap q_{C_1}) \odot q' = m{:}s \odot q_{C_1}$. $\square$

We are now ready to present the proof of Theorem 2 stating:

> Let $C \in \mathit{Conf}(G_\gamma)$; its transition system $(\mathit{Conf}(G_\gamma), C, \longrightarrow)$ is weakly bisimilar to the rooted transition system $(\mathit{Conf}(L_\gamma), \mathcal{E}_{GL}(C), \longrightarrow)$.

In order to prove this, consider the relation

$$\mathcal{R} = \{(\mathcal{E}_{LG}(C), C) \;\mid\; C \in \mathit{Conf}(L_\gamma)\}.$$

As a corollary of the Propositions 10 and 11, we have that the relation $\mathcal{R}$ is a weak bisimulation. For each $C \in \mathit{Conf}(G_\gamma)$ we have that $\mathcal{E}_{LG}(\mathcal{E}_{GL}(C)) = C$, thus $(C, \mathcal{E}_{GL}(C)) \in \mathcal{R}$, hence the rooted transition system $(\mathit{Conf}(G), C, \longrightarrow)$ is weakly bisimilar to $(\mathit{Conf}(L), \mathcal{E}_{GL}(C), \longrightarrow)$.

In a similar way we can prove Theorem 3.

B Proofs of Theorems 4 and 5

We first present two propositions which take into account pairs of configurations of the kind $(\mathcal{E}_{GU}(C), C)$ where $C \in Conf(G_\gamma)$. The first proposition assumes that the configuration $\mathcal{E}_{GU}(C)$ performs a transition, while the second considers a transition performed by C. In both cases, the propositions show that the opposite configuration may perform a (weak) transition which mimics the considered behavior.

Proposition 12 *Consider $A_C, q_C \in Conf(G_\gamma)$ and its encoding $\mathcal{E}_{GU}(A_C, q_C) = C$; we have that*

if $C \xrightarrow[\ell]{m} C'$ then there exists ℓ' such that $A_C, q_C \xrightarrow[\ell']{m} A'_C, q'_C$ where $\mathcal{E}_{GU}(A'_C, q'_C) = C'$.

PROOF. By case analysis on the last rule applied to derive the transition $C \xrightarrow[\ell]{m} C'$.

The only interesting case is the one of rule (U7). In this case, it is not difficult to see that inside the configuration C there exists an active process $go(m{:}s).P$ which performs its output operation. Thus, we have $C \xrightarrow[\ell]{m} C'$ with $C' = (C_1 \Leftarrow m{:}s)$ where C_1 is obtained by substituting in C the process $go(m{:}s).P$ with P.
The process $go(m{:}s).P$ is present also in the configuration A_C, q_C and it may perform its output operation: $A_C, q_C \xrightarrow{m} A'_C, m{:}s \odot q_C$, where A'_C is obtained by substituting in A_C the process $go(m{:}s).P$ with P.
By definition of $\mathcal{E}_{GU}$ we have $C = (A_C \Leftarrow q_c)$. Finally, we observe that $C' = (A'_C \Leftarrow q_c) \Leftarrow m{:}s$ which is the same as $\mathcal{E}_{GU}(A'_C, m{:}s \odot q_C)$. $\square$

Proposition 13 *Consider $A_C, q_C \in Conf(G_\gamma)$ and its encoding $\mathcal{E}_{GU}(A_C, q_C) = C$; we have that*

(1) if $A_C, q_C \xrightarrow[\ell]{m} A'_C, q'_C$ then also $C \xrightarrow[m]{m} C'$ such that $\mathcal{E}_{GU}(A'_C, q'_C) = C'$ and $q'_C = m{:}s \odot q_C$, for some sort s,
(2) if $A_C, q_C \xrightarrow[\ell]{} A'_C, q'_C$ then one of the following holds:
- $\mathcal{E}_{GU}(A'_C, q'_C) = C$ and $q'_C = q_C$,
- $C \longrightarrow C'$ with $\mathcal{E}_{GU}(A'_C, q'_C) = C'$ and $q'_C = q_C,$.

PROOF. The proof is by induction on the length of the derivation of the considered transition $A_C, q_C \xrightarrow[\ell]{m} A'_C, q'_C$ or $A_C, q_C \xrightarrow[\ell]{} A'_C, q'_C$.

In the base case an axiom is used to derive the considered transition; the thesis can be trivially proved. In the inductive case we proceed by case analysis on the last rule applied. We present only the case of rule (G7); the other rules are treated similarly.

If the last applied rule is (G7) we have $A_C, q_C = [P,d] \parallel A_{C_1}, q_C$ and $A'_C, q'_C = [P,d] \parallel A'_{C_1}, q'_C$ with $A_{C_1}, q_C \xrightarrow{o} A'_{C_1}, q'_C$. By definition of $\mathcal{E}_{GU}$ we have $C = [P,d] \Leftarrow q_C \parallel C_1$ with $C_1 = A_{C_1} \Leftarrow q_C$. By inductive hypothesis we have two possible cases.

If $o = m$ then $C_1 \xrightarrow[m]{m} C'_1$ such that $\mathcal{E}_{GU}(A'_{C_1}, q'_C) = C'_1$ and $q'_C = m{:}s \odot q_C$, for some sort s. Hence, by rule (U6) we have also $[P, (d \leftarrow q_C)] \parallel C_1 \xrightarrow[m]{m} [P, (d \leftarrow q_C) \oplus m] \parallel C'_1$. It remains to show that the last term is the same as $\mathcal{E}_{GU}(A'_C, q'_C)$. This simply follows by the fact that $\mathcal{E}_{GU}(A'_C, q'_C) = [P,d] \Leftarrow q'_C \parallel \mathcal{E}_{GU}(A'_{C_1}, q'_C) = [P,d] \Leftarrow (m{:}s \odot q_C) \parallel C'_1$.

If $o = \tau$ then we have two different cases to analyze.

- $\mathcal{E}_{GU}(A'_{C_1}, q'_C) = C_1$ and $q'_C = q_C$.
  The thesis directly follows from the fact that $\mathcal{E}_{GU}(A'_C, q'_C) = [P,d] \Leftarrow q'_C \parallel \mathcal{E}_{GU}(A'_{C_1}, q'_C) = [P,d] \Leftarrow q_C \parallel C_1 = C$.
- $C_1 \longrightarrow C'_1$ with $\mathcal{E}_{GU}(A'_{C_1}, q'_C) = C'_1$ and $q'_C = q_C$.
  By rule (U6) we have also $[P, d \leftarrow q_C] \parallel C_1 \longrightarrow [P, d \leftarrow q_C] \parallel C'_1$. It remains to show that the last term is the same as $\mathcal{E}_{GU}(A'_C, q'_C)$. This simply follows by the fact that $\mathcal{E}_{GU}(A'_C, q'_C) = [P,d] \Leftarrow q'_C \parallel \mathcal{E}_{GU}(A'_{C_1}, q'_C) = [P, d \leftarrow q_C] \parallel C'_1$. $\square$

We are now ready to present the proof of Theorem 4 stating:

> Let $C \in \mathit{Conf}(U_\gamma)$; its transition system $(\mathit{Conf}(U_\gamma), C, \longrightarrow)$ is weakly bisimilar to the rooted transition system $(\mathit{Conf}(G_\gamma), \mathcal{E}_{UG}(C), \longrightarrow)$.

The proof is the same as the proof of Theorem 2 when we consider the relation

$$\mathcal{R} = \{(\mathcal{E}_{GU}(C), C) \mid C \in \mathit{Conf}(G_\gamma)\}.$$

Theorem 5 can be proved in a similar way.

C Proofs of Theorems 6 and 7

Also in this case the equivalence result is a consequence of two theorems which require, in order to be proved, the following result.

Proposition 14 *Consider $C_M \in Conf(L_\mu)$ and its encoding $\mathcal{E}_{MS}(C_M) = C_S$; we have that*

(1) if $C_M \xrightarrow[\ell]{m} C'_M$ then also $C_S \xrightarrow[\ell]{m} \mathcal{E}_{MS}(C'_M)$,
(2) if $C_S \xrightarrow[\ell]{m} C'_S$ then also $C_M \xrightarrow[\ell]{m} C'_M$ where $C'_S = \mathcal{E}_{MS}(C'_M)$.

PROOF. The proof is by case analysis on the last rule applied to derive the transition $C_M \xrightarrow[\ell]{m} C'_M$ or $C_S \xrightarrow[\ell]{m} C'_S$. $\square$

Theorem 6 states:

> Let $C \in Conf(L_\sigma)$; its transition system $(Conf(L_\sigma), C, \longrightarrow)$ is weakly bisimilar to the rooted transition system $(Conf(L_\mu), \mathcal{E}_{SM}(C), \longrightarrow)$.

The proof is the same as the proof of Theorem 2 when we consider the relation

$$\mathcal{R} = \{(\mathcal{E}_{MS}(C), C) \ \mid \ C \in Conf(L_\mu)\} .$$

Theorem 7 can be proved in a similar way.

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 *Lecture Notes in Computer Science*, pages 182–198, Springer-Verlag, 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. What do you mean, coordination? *Bulletin of the Dutch Association for Theoretical Computer Science, NVTI*, pages 11–22, 1998. Available on-line at `http://www.cwi.nl/NVTI/Nieuwsbrief/nieuwsbrief.html`.

[4] F. Arbab, C. Blom, F. Burger, and C. Everaars. Reusable Coordination Modules for Massively Concurrent Applications. *Software: Practice and Experience*, 28(7):703–735, 1998.

[5] F. Arbab, J.W. de Bakker, M.M. Bonsangue, J.J.M.M. Rutten, A. Scutellà, and G. Zavattaro. A transition system semantics for the control-driven coordination language MANIFOLD. *Theoretical Computer Science*, 240(1): 3-47, 2000.

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

[7] F.S. de Boer and C. Palamidessi. Embedding as a tool for language comparison. *Information and Computation*, 108(1):128–157, 1994.

[8] 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 SAC'98*, pages 159–168. ACM Press, 1998.

[9] M.M. Bonsangue, J.N. Kok, and G. Zavattaro. Comparing coordination models based on shared distributed replicated data. In *Proc. of SAC'99*, pages 156–165. ACM Press, 1999.

[10] M.M. Bonsangue, J.N. Kok, and G. Zavattaro. Comparing Software Architectures for Coordination Languages. In *Proc. of Coordination'99*, volume 1594 of *Lecture Notes in Computer Science*, pages 150–165. Springer-Verlag, 1999.

[11] A. Brogi and J.-M. Jaquet. On the expressiveness of coordination models. In *Proc. of Coordination 99*, volume 1594 of *Lecture Notes in Computer Science*, pages 134–149, Springer-Verlag, 1999.

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

[13] N. Busi, R. Gorrieri, and G. Zavattaro. Comparing Three Semantics for Linda-like Languages. *Theoretical Computer Science*, 240(1):49-90, 2000.

[14] N. Busi, R. Gorrieri, and G. Zavattaro. On the Expressiveness of Linda Coordination Primitives. *Information and Computation*, 156(1/2):90-121, 2000.

[15] N. Carriero and D. Gelernter. Linda in context. *Communications of the ACM*, 32(4):444–458, ACM Press, 1989.

[16] A. Engels, S. Mauw and M.A. Reniers. A Hierarchy of Communication Models for Message Sequence Charts. In *Proc. of FORTE X and PSTV XVII*, pages 75-90, Chapman & Hall, 1997.

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

[18] A. Mazurkiewicz. Trace theory. In *Petri Nets, Applications, and Relationship to other models of Concurrency*, volume 255 of *Lecture Notes in Computer Science*, pages 279–324, Springer-Verlag, 1987.

[19] R. Milner. *A Calculus of Communication Systems.* Volume 92 of *Lecture Notes in Computer Science*, Springer-Verlag, 1980

[20] R. Milner. *Communication and Concurrency.* Prentice Hall, 1989.

[21] C. Palamidessi. Comparing the expressive power of the synchronous and the asynchronous pi-calculus. In *Proc. of the 24th ACM Symposium on Principles of Programming Languages (POPL)*, pages 256–265, ACM, 1997.

[22] G.A. Papadopoulos and F. Arbab. Coordination Models and Languages. *Advances in Computers*, 46:329–400, 1998.

[23] E.Y. Shapiro. The family of concurrent logic programming languages. *ACM Computing Surveys*, 21(3):412–510, 1989.

[24] M. Shaw and D. Garlan. *Software architecture: perspective on an emerging discipline*. Prentice-Hall, 1996.

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